**AI Assisted Coding**

**Lab Test - 2**

**Name:** Adepu Goutham Sri Bhargav **Id.no:** 2403A510C4

**Batch:** 05 **Date:** 17-09-2025

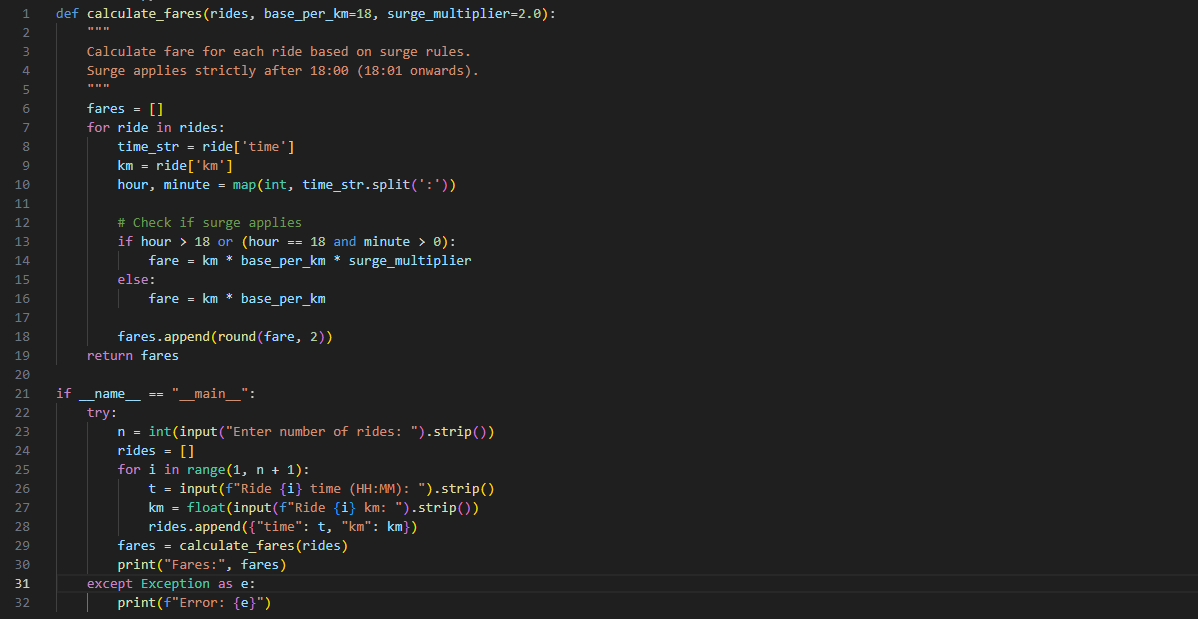
**Q.no-1: B.1 — [S09B1] Apply surge/penalty rules (conditionals)**  
Scenario (sports analytics):  
**Context:**

Pricing in the sports analytics app uses a base per-km rate and time-based surge after business  
peaks. Product wants a deterministic calculator for receipts and audits.  
**Your Task:**  
Implement a fare function: fare = km \* base\_per\_km \* surge Multiplier, 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. Edge case: exactly at 18:00  
should be treated as non-surge for 18:00:00; after 18:00 (e.g., 18:01) surges.  
**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': '08:00', 'km': 3.0}, {'time': '18:30', 'km': 5.0}]  
*Sample Output*  
[54.0, 108.0]  
Acceptance Criteria: Correct surge threshold and rounding

**Prompt:**

“Write a Python script with a function calculates fares(rides, base per km=18, surge multiplier=2.0) that computes fare = km \* base\_per\_km \* surge multiplier with surge applying strictly after 18:00 (18:01+). rides is a list of dicts: {'time': 'HH:MM', 'km': float}. Round each fare to 2 decimals, do not mutate input, no external libraries. In \_\_main\_\_, prompt the user for number of rides, then for each ride ask time (HH:MM) and km, compute fares, and print the list. Treat exactly 18:00 as non-surge and parse HH:MM safely.”

**Code:**



**Output:**

****

**Observation:**

* Correct surge rule: strictly after 18:00; 18:00 is non-surge.
* Rounds to 2 decimals; does not mutate input.
* CLI collects rides and prints fares; base\_per\_km/surge multiplier are fixed defaults.
* Time parsing lacks validation (e.g., 24:00, 18:60, non-numeric) and will crash on bad input.
* Python round uses banker's rounding; may differ from “round half up” expectations.

**Q.no: B.2 — [S09B2] Debug rolling mean (off-by-one)**  
Scenario (sports analytics):  
**Context:**A team in sports analytics 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=[9, 10, 11, 12] 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=[9, 10, 11, 12], 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*  
[9.5, 10.5, 11.5]  
Acceptance Criteria: All valid windows included; passes tests; no index errors

**Prompt:**

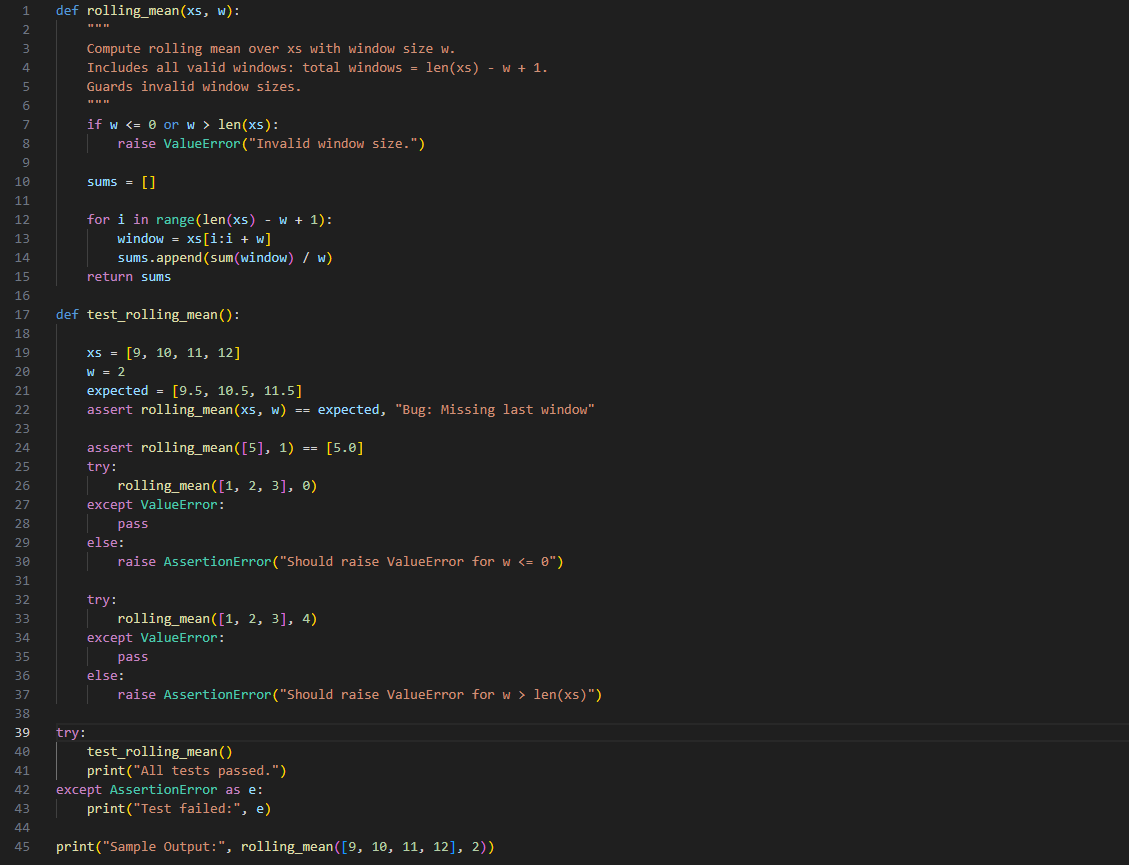
“Give me a Python code to fix the bug in a rolling mean function.

- All valid windows must be included: number of windows = len(xs) - w + 1.

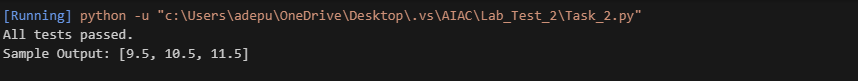
- If w <= 0 or w > len(xs), raise ValueError.

- Example: xs = [9,10,11,12], w = 2 → [9.5,10.5,11.5].”

**Code:**

****

**Output:**

****

**Observation:**

* Correct: computes all windows; raises Value Error for w <= 0 or w > Len(xs); returns floats.
* Complexity: O(n\*w) due to slicing; can be O(n) with a sliding window.
* Memory: slicing allocates new lists per window.
* Robustness: assumes numeric xs and int w; non-numeric will fail implicitly.
* Tests: good basics; consider adding w == Len(xs), empty xs, float/negative values, and float tolerance.