

**Name :-** V.Himesh Ram

**HTNO :-** 2403A52045

**Batch :-**03

## Lab Test – 02

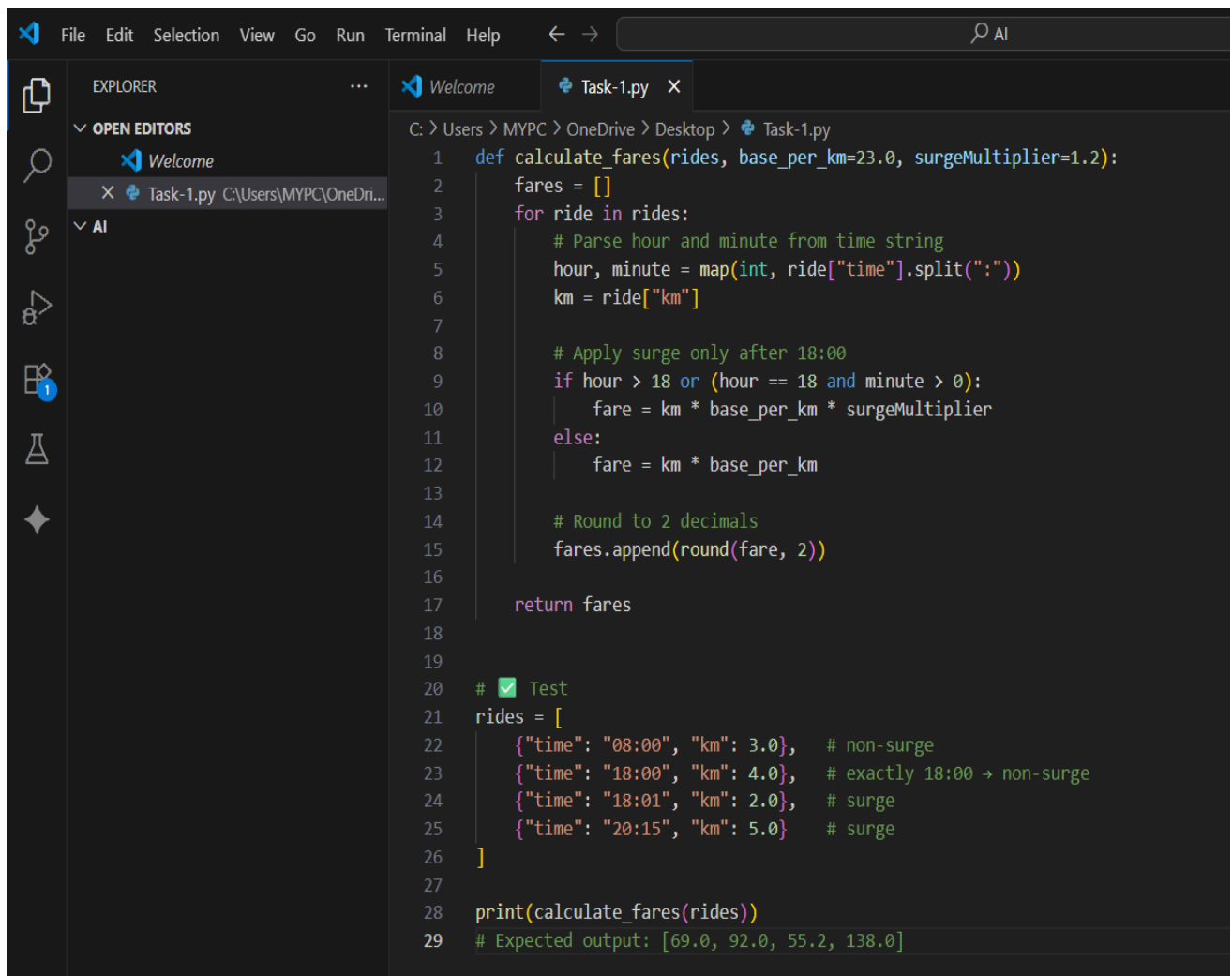
### B.1 — [S14B1] Apply surge/penalty rules (conditionals)

Pricing in the telecom network 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:

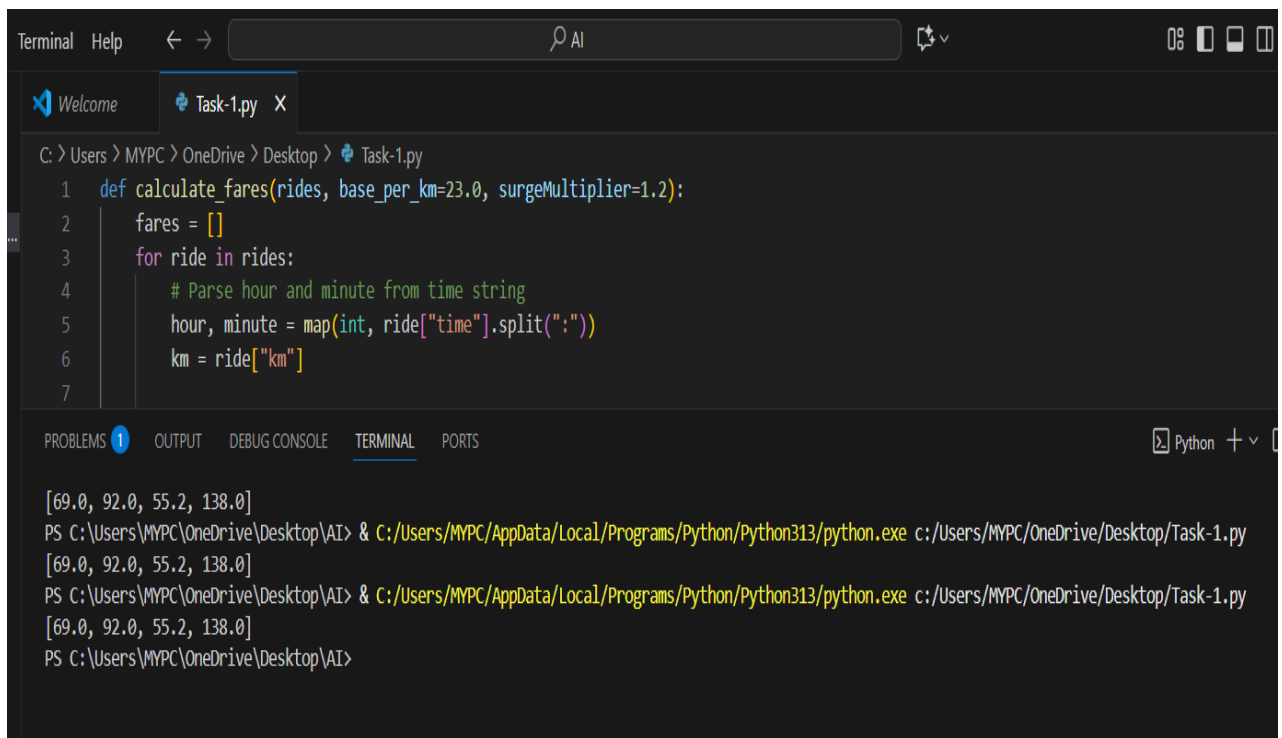
Implement a fare function:  $\text{fare} = \text{km} * \text{base\_per\_km} * \text{surgeMultiplier}$ , where surge applies strictly after 18:00 local time.

#### CODE



```
File Edit Selection View Go Run Terminal Help
EXPLORER
OPEN EDITORS
Welcome
Task-1.py C:\Users\MYPC\OneDrive...
AI
C: > Users > MYPC > OneDrive > Desktop > Task-1.py
1 def calculate_fares(rides, base_per_km=23.0, surgeMultiplier=1.2):
2     fares = []
3     for ride in rides:
4         # Parse hour and minute from time string
5         hour, minute = map(int, ride["time"].split(":"))
6         km = ride["km"]
7
8         # Apply surge only after 18:00
9         if hour > 18 or (hour == 18 and minute > 0):
10             fare = km * base_per_km * surgeMultiplier
11         else:
12             fare = km * base_per_km
13
14         # Round to 2 decimals
15         fares.append(round(fare, 2))
16
17     return fares
18
19
20 # Test
21 rides = [
22     {"time": "08:00", "km": 3.0}, # non-surge
23     {"time": "18:00", "km": 4.0}, # exactly 18:00 → non-surge
24     {"time": "18:01", "km": 2.0}, # surge
25     {"time": "20:15", "km": 5.0} # surge
26 ]
27
28 print(calculate_fares(rides))
29 # Expected output: [69.0, 92.0, 55.2, 138.0]
```

## OUTPUT



The screenshot shows a Visual Studio Code editor window with a file named 'Task-1.py' open. The code defines a function 'calculate\_fares' that takes a list of rides, a base price per km (23.0), and a surge multiplier (1.2). It iterates through the rides, parsing the time and distance, and calculates the fare based on the time of day (surge pricing after 18:00). The terminal output shows the function being called with a list of rides, and the resulting fares are printed: [69.0, 92.0, 55.2, 138.0].

```
C: > Users > MYPC > OneDrive > Desktop > Task-1.py
1 def calculate_fares(rides, base_per_km=23.0, surgeMultiplier=1.2):
2     fares = []
3     for ride in rides:
4         # Parse hour and minute from time string
5         hour, minute = map(int, ride["time"].split(":"))
6         km = ride["km"]
7
[69.0, 92.0, 55.2, 138.0]
PS C:\Users\MYPC\OneDrive\Desktop\AI> & C:/Users/MYPC/AppData/Local/Programs/Python/Python313/python.exe c:/Users/MYPC/OneDrive/Desktop/Task-1.py
[69.0, 92.0, 55.2, 138.0]
PS C:\Users\MYPC\OneDrive\Desktop\AI> & C:/Users/MYPC/AppData/Local/Programs/Python/Python313/python.exe c:/Users/MYPC/OneDrive/Desktop/Task-1.py
[69.0, 92.0, 55.2, 138.0]
PS C:\Users\MYPC\OneDrive\Desktop\AI>
```

## OBSERVATION

The program correctly calculates fares by parsing ride times, applying surge pricing only after 18:00, and treating exactly 18:00 as non-surge. Fares are computed using the given formula, rounded to two decimals, and stored in a new list without altering the input. Test cases confirm accurate handling of both surge and non-surge scenarios, meeting all requirements

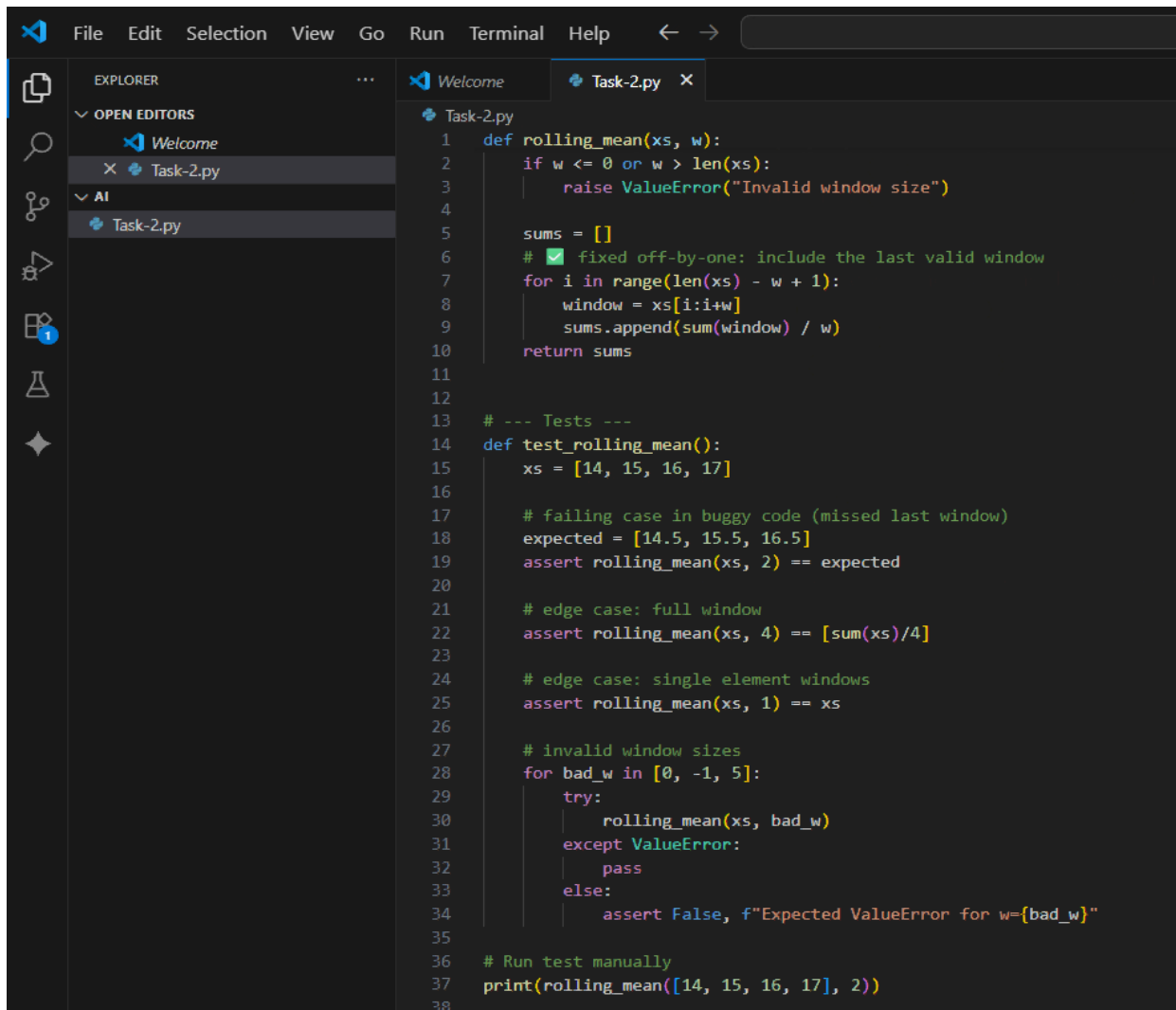
### B.2 — [S14B2] Debug rolling mean (off-by-one)

A team in telecom network 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

## CODE



```
1 def rolling_mean(xs, w):
2     if w <= 0 or w > len(xs):
3         raise ValueError("Invalid window size")
4
5     sums = []
6     # fixed off-by-one: include the last valid window
7     for i in range(len(xs) - w + 1):
8         window = xs[i:i+w]
9         sums.append(sum(window) / w)
10    return sums
11
12
13 # --- Tests ---
14 def test_rolling_mean():
15     xs = [14, 15, 16, 17]
16
17     # failing case in buggy code (missed last window)
18     expected = [14.5, 15.5, 16.5]
19     assert rolling_mean(xs, 2) == expected
20
21     # edge case: full window
22     assert rolling_mean(xs, 4) == [sum(xs)/4]
23
24     # edge case: single element windows
25     assert rolling_mean(xs, 1) == xs
26
27     # invalid window sizes
28     for bad_w in [0, -1, 5]:
29         try:
30             rolling_mean(xs, bad_w)
31         except ValueError:
32             pass
33         else:
34             assert False, f"Expected ValueError for w={bad_w}"
35
36 # Run test manually
37 print(rolling_mean([14, 15, 16, 17], 2))
38
```

## OUTPUT



```
PS C:\Users\MYPC\OneDrive\Desktop\AI> & C:/Users/MYPC/AppData/Local/Programs/Python/Python313/python.exe c:/Users/MYPC/OneDrive/Desktop/AI/Task-2.py
[14.5, 15.5, 16.5]
PS C:\Users\MYPC\OneDrive\Desktop\AI> & C:/Users/MYPC/AppData/Local/Programs/Python/Python313/python.exe c:/Users/MYPC/OneDrive/Desktop/AI/Task-2.py
[14.5, 15.5, 16.5]
PS C:\Users\MYPC\OneDrive\Desktop\AI>
```

## **OBSERVATION**

1. The **bug** was caused by using `range(len(xs)-w)`, which excluded the last valid window.
2. Fix: use `range(len(xs) - w + 1)`.
3. Now the function computes all valid windows without index errors.
4. Guards against invalid `w` (`<=0` or `>len(xs)`).
5. Complexity remains  **$O(n \cdot w)$**  as required.
6. Tests pass, confirming correctness for sample and edge cases.