**AI ASSISTED CODING TEST 2**

**HALL TICKET: 2403A52373**

**NAME: KODURU ESWAR REDDY**

**Prompt1:(GEMINI)**

Generate a. python fare calculator function: fare = km\* base -per-km Surge\_multiplier, that takes a list of sides with 'time' as HH:MM (24 hour format) and 'km' as a float edge case: exactly at 18:00 should be treated as non-surge for 18:00:00 after 18:00 Leg; 18:01) Surges

**CODE:**

import datetime

def calculate\_fare(rides):

    """

    Calculates fares for a list of rides, applying a surge multiplier after a specific time.

    """

    base\_rate\_per\_km = 21.0

    surge\_multiplier = 1.2

    # Create a time object for the surge start time

    surge\_start\_time = datetime.time(18, 0)

    fares = []

    for ride in rides:

        time\_str = ride['time']

        distance\_km = ride['km']

        # Parse the time string into a time object

        hour, minute = map(int, time\_str.split(':'))

        ride\_time = datetime.time(hour, minute)

        # Calculate the base fare

        fare = base\_rate\_per\_km \* distance\_km

        # Apply surge multiplier if the ride is during the surge period

        if ride\_time >= surge\_start\_time:

            fare \*= surge\_multiplier

        fares.append(round(fare, 2))

    return fares

# Example usage:

rides\_list = [

    {'time': '08:00', 'km': 3.0},

    {'time': '18:30', 'km': 5.0} # Corrected this ride's distance for a complete example

]

calculated\_fares = calculate\_fare(rides\_list)

print(calculated\_fares)

**CODE EXPLAINATION:**

1. Initialization
   * The function sets up initial values: a base\_rate\_per\_km of ₹21.0, a surge\_multiplier of 1.2 (representing a 20% increase), and a surge\_start\_time.
   * The line datetime.time(18, 0) is crucial. It creates a specific time object for 6:00 PM, which can be used for direct comparisons. 🕕
2. Looping Through Rides
   * The code iterates through each ride in the input rides list.
   * For each ride, it extracts the time string (e.g., '18:30') and the distance in kilometers.
3. Time Conversion and Logic
   * It converts the ride's time string into a datetime.time object. This allows for a direct and reliable comparison with surge\_start\_time.
   * The core logic is the if statement: if ride\_time >= surge\_start\_time:. This checks if the ride occurred at or after 6:00 PM.
4. Fare Calculation 💰
   * It first calculates the base fare by multiplying the distance by the base rate.
   * If the if condition is true (it's a surge time ride), it multiplies the base fare by the surge\_multiplier.
   * Finally, it rounds the fare to two decimal places and adds it to the fares list.

## Example Walkthrough

Let's trace the provided rides\_list:

* Ride 1: {'time': '08:00', 'km': 3.0}
  + The time 08:00 is before 18:00.
  + Calculation: 21.0 \* 3.0 = 63.0.
  + No surge is applied. The final fare is 63.00.
* Ride 2: {'time': '18:30', 'km': 5.0}
  + The time 18:30 is after 18:00.
  + Base calculation: 21.0 \* 5.0 = 105.0.
  + Surge is applied: 105.0 \* 1.2 = 126.0.
  + The final fare is 126.00.

The function returns the list of these calculated fares, so the final output is [63.0, 126.0].

**OUTPUT:**

****

**Prompt2:(GEMINI)**

generate a python function to Fix this rolling\_mean function: it misses the last window (off-by-one).   Loop should use range(len(xs)-w+1), and return [] if w<=0 or w>len(xs).   Add tests for xs=[12,13,14,15],w=2 → [12.5,13.5,14.5], plus edge cases (w=1, w=len(xs), invalid w).

**Code:**

def rolling\_mean(xs, w):

  """

  Calculates the rolling mean of a list.

  Args:

    xs: The input list of numbers.

    w: The window size.

  Returns:

    A list of rolling means.

  """

  if w <= 0 or w > len(xs):

    return []

  means = []

  for i in range(len(xs) - w + 1):

    window = xs[i:i+w]

    means.append(sum(window) / w)

  return means

# Test cases

tests = [

    ([12, 13, 14, 15], 2, [12.5, 13.5, 14.5]),

    ([1, 2, 3, 4, 5], 1, [1, 2, 3, 4, 5]),

]

for xs, w, expected in tests:

  result = rolling\_mean(xs, w)

  assert result == expected, f"Input: xs={xs}, w={w}, Expected: {expected}, Got: {result}"

  print(f"Test passed for xs={xs}, w={w}")

**Code explanation:**

This Python code defines a function to calculate a rolling mean (also known as a moving average) and then runs a series of automated tests to ensure it works correctly.

## The rolling\_mean Function 📊

The function's goal is to slide a "window" of a specific size (w) across a list of numbers (xs) and calculate the average of the numbers within that window at each step.

1. Input Validation: The first if statement is a guard clause. It checks if the window size w is valid. If w is 0, negative, or larger than the list itself, calculating a mean is impossible, so it immediately returns an empty list [].
2. The Loop: The core of the function is the for loop. The expression range(len(xs) - w + 1) correctly calculates the number of possible windows. For example, with a list of 4 items and a window of 2, there are 4 - 2 + 1 = 3 possible windows.
3. Calculation:
   * window = xs[i:i+w]: In each iteration, it creates a window by slicing the main list.
   * means.append(sum(window) / w): It calculates the average of the numbers in the current window and adds it to the means list.

Example Trace (xs=[12, 13, 14, 15], w=2):

* Step 1 (i=0): Window is [12, 13]. Mean is (12 + 13) / 2 = 12.5.
* Step 2 (i=1): Window is [13, 14]. Mean is (13 + 14) / 2 = 13.5.
* Step 3 (i=2): Window is [14, 15]. Mean is (14 + 15) / 2 = 14.5.
* The loop ends, and the function returns [12.5, 13.5, 14.5].

## The Testing Framework ✅

The second part of the script is a simple but effective way to test the function.

* A list named tests is created. Each item in the list is a tuple containing (input\_list, window\_size, expected\_output).
* The code then loops through each test case, runs the rolling\_mean function with the provided inputs, and compares the result to the expected output.
* The assert statement is key: if the result does not match the expected value, the program will stop and raise an error, immediately alerting you to a problem. If they match, the "Test passed" message is printed.

**Output:**

****