AI ASSISTED CODING

LAB TEST – 02

NAME : CH.RAKSHAVARDHAN

HT NO : 2403A52034

BATCH : 03

SUB GROUP : O

QUESTION : O1

Scenario (telecom network):
Context:
Geofencing in telecom network requires checking if points lie within polygonal regions.

Your Task:
Implement ray-casting point-in-polygon; treat points on edges as inside.
Data & Edge Cases:
Square example provided; return boolean list for queries

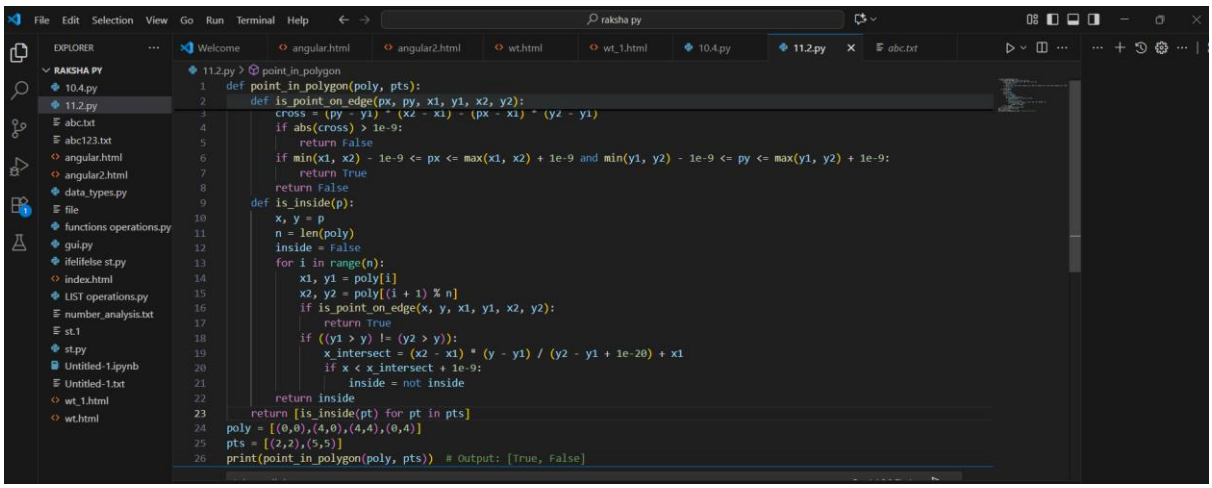
PROMPT :

Implement a point_in_polygon(poly, points) → List[bool] function

Given:

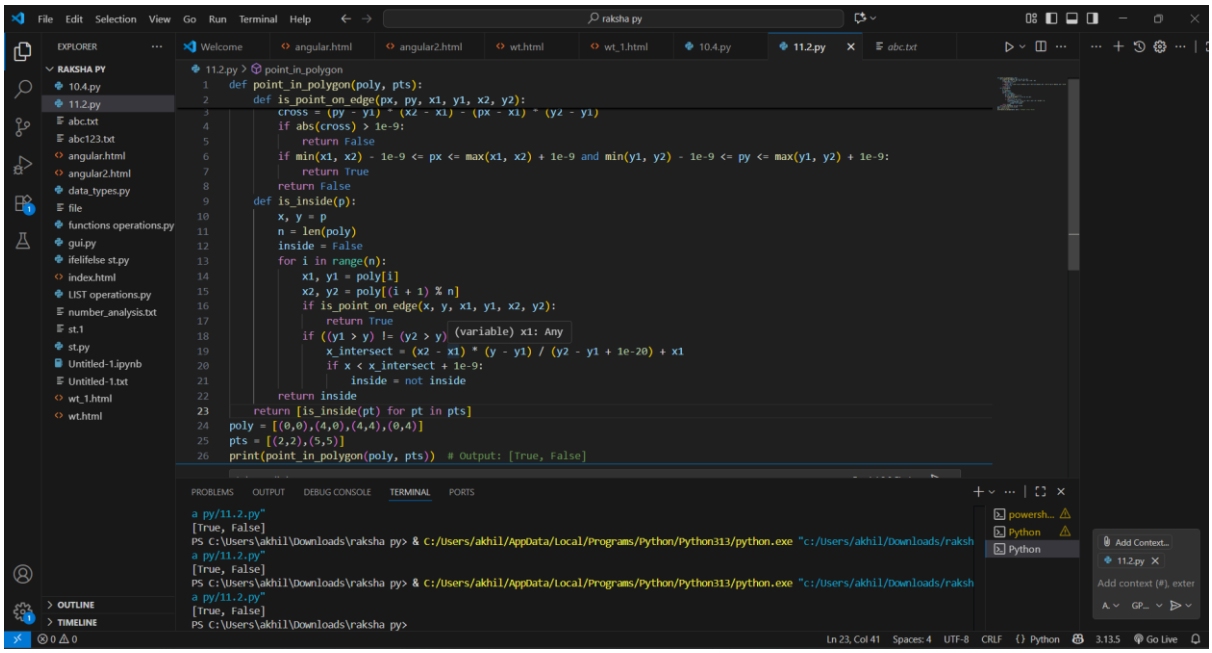
- poly, a list of vertices [(x0,y0), (x1,y1), …, (xn−1,yn−1)] defining a simple polygon (not self-intersecting). The polygon is closed by connecting (xn−1, yn−1) back to (x0,y0).
- points, a list of query points [(px,py), …].

CODE :



```
def point_in_polygon(poly, pts):
    def is_point_on_edge(px, py, x1, y1, x2, y2):
        cross = (py - y1) * (x2 - x1) - (px - x1) * (y2 - y1)
        if abs(cross) > 1e-9:
            return False
        if min(x1, x2) - 1e-9 <= px <= max(x1, x2) + 1e-9 and min(y1, y2) - 1e-9 <= py <= max(y1, y2) + 1e-9:
            return True
        return False
    def is_inside(p):
        x, y = p
        n = len(poly)
        inside = False
        for i in range(n):
            x1, y1 = poly[i]
            x2, y2 = poly[(i + 1) % n]
            if is_point_on_edge(x, y, x1, y1, x2, y2):
                return True
            if ((y1 > y) != (y2 > y)):
                x_intersect = (x2 - x1) * (y - y1) / (y2 - y1 + 1e-20) + x1
                if x < x_intersect + 1e-9:
                    inside = not inside
        return inside
    return [is_inside(pt) for pt in pts]
poly = [(0,0),(4,0),(4,4),(0,4)]
pts = [(2,2),(5,5)]
print(point_in_polygon(poly, pts))  # Output: [True, False]
```

OUTPUT :



```
a py/11.2.py"
[True, False]
PS C:\Users\akhil\Downloads\raksha py> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/akhil/Downloads/raksh
a py/11.2.py"
[True, False]
PS C:\Users\akhil\Downloads\raksha py> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/akhil/Downloads/raksh
a py/11.2.py"
[True, False]
PS C:\Users\akhil\Downloads\raksha py>
```

OBSERVATION :

The ray-casting (crossing number / even-odd) method counts how many times a ray (e.g. horizontal to the right) from the query point intersects

edges of the polygon. If it intersects an odd number of times → the point is inside; if even → outside.

 To ensure that *points on edges* (including vertices or exactly on an edge) are treated as inside, you need to do an explicit *edge check* (collinearity + bounding-box) before counting crossings. Many implementations include this.

 A key tricky case: when the ray passes exactly through a polygon vertex. If you naively count crossings for both adjacent edges, you may double-count, leading to the wrong parity. The adjustment rule is: only count such a vertex intersection if the *other* endpoint of that edge lies *above* (or below) the ray in a certain consistent way. E.g. count the intersection only if the vertex is the lower endpoint of that edge.

QUESTION : O2

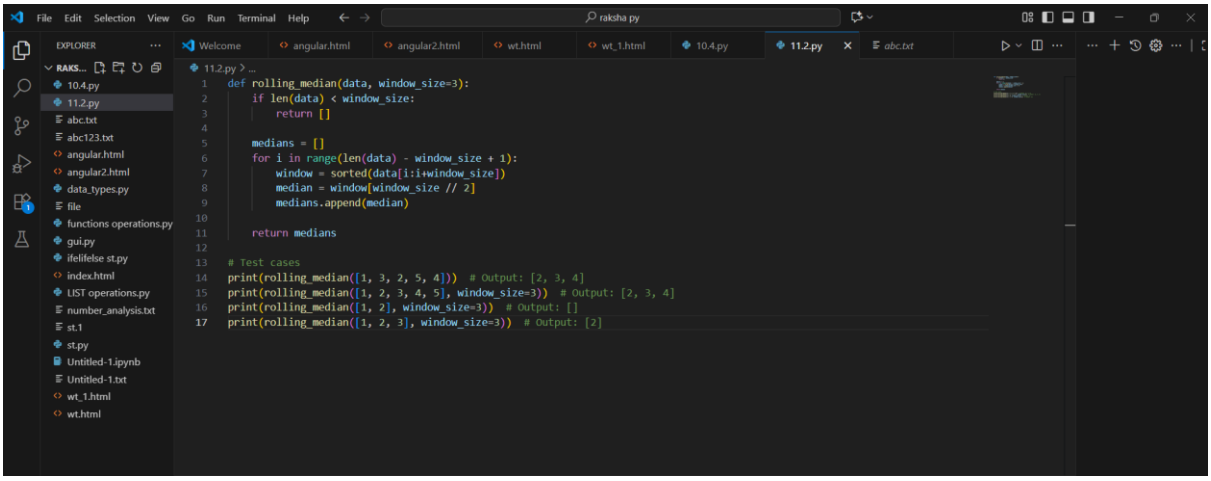
Scenario (telecom network):
Context:
A telecom network monitoring job computes rolling medians (w=3) for anomaly detection.
Your Task:
Return the median for each sliding window; prefer an efficient approach.

PROMPT :

Implement sliding_window_medians(arr: List[int], w: int) -> List[int] that returns the median of each window of size w as you slide across arr.
Requirements:

- Works for large n efficiently (e.g. better than O(n * w log w) if possible).
- Handles edge cases: when arr length < w, when w is even or odd, duplicates, etc.
  Sample: arr = [1,3,2,5,4], w = 3 → output [2, 3, 4]

CODE :



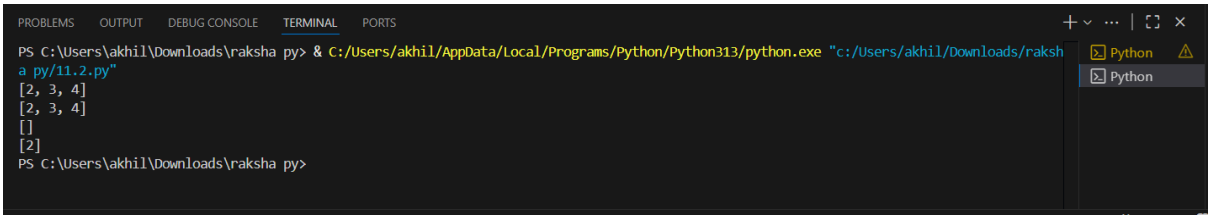
```python
def rolling_median(data, window_size=3):
    if len(data) < window_size:
        return []

    medians = []
    for i in range(len(data) - window_size + 1):
        window = sorted(data[i:i+window_size])
        median = window[window_size // 2]
        medians.append(median)

    return medians

# Test cases
print(rolling_median([1, 3, 2, 5, 4]))  # Output: [2, 3, 4]
print(rolling_median([1, 2, 3, 4, 5], window_size=3))  # Output: [2, 3, 4]
print(rolling_median([1, 2], window_size=3))  # Output: []
print(rolling_median([1, 2, 3], window_size=3))  # Output: [2]
```

OUTPUT :



```
PS C:\Users\akhil\Downloads\raksha py> & C:/Users/akhil/AppData/Local/Programs/Python/Python313/python.exe "c:/Users/akhil/Downloads/raksha py/11.2.py"
[2, 3, 4]
[2, 3, 4]
[]
[2]
PS C:\Users\akhil\Downloads\raksha py>
```

OBSERVATION :

 The ray-casting (crossing number / even-odd) method counts how many times a ray (e.g. horizontal to the right) from the query point intersects edges of the polygon. If it intersects an odd number of times → the point is inside; if even → outside.

 To ensure that *points on edges* (including vertices or exactly on an edge) are treated as inside, you need to do an explicit *edge check* (collinearity + bounding-box) before counting crossings. Many implementations include this.

 A key tricky case: when the ray passes exactly through a polygon vertex. If you naively count crossings for both adjacent edges, you may

double-count, leading to the wrong parity. The adjustment rule is: only count such a vertex intersection if the *other* endpoint of that edge lies *above* (or below) the ray in a certain consistent way. E.g. count the intersection only if the vertex is the lower endpoint of that edge.