

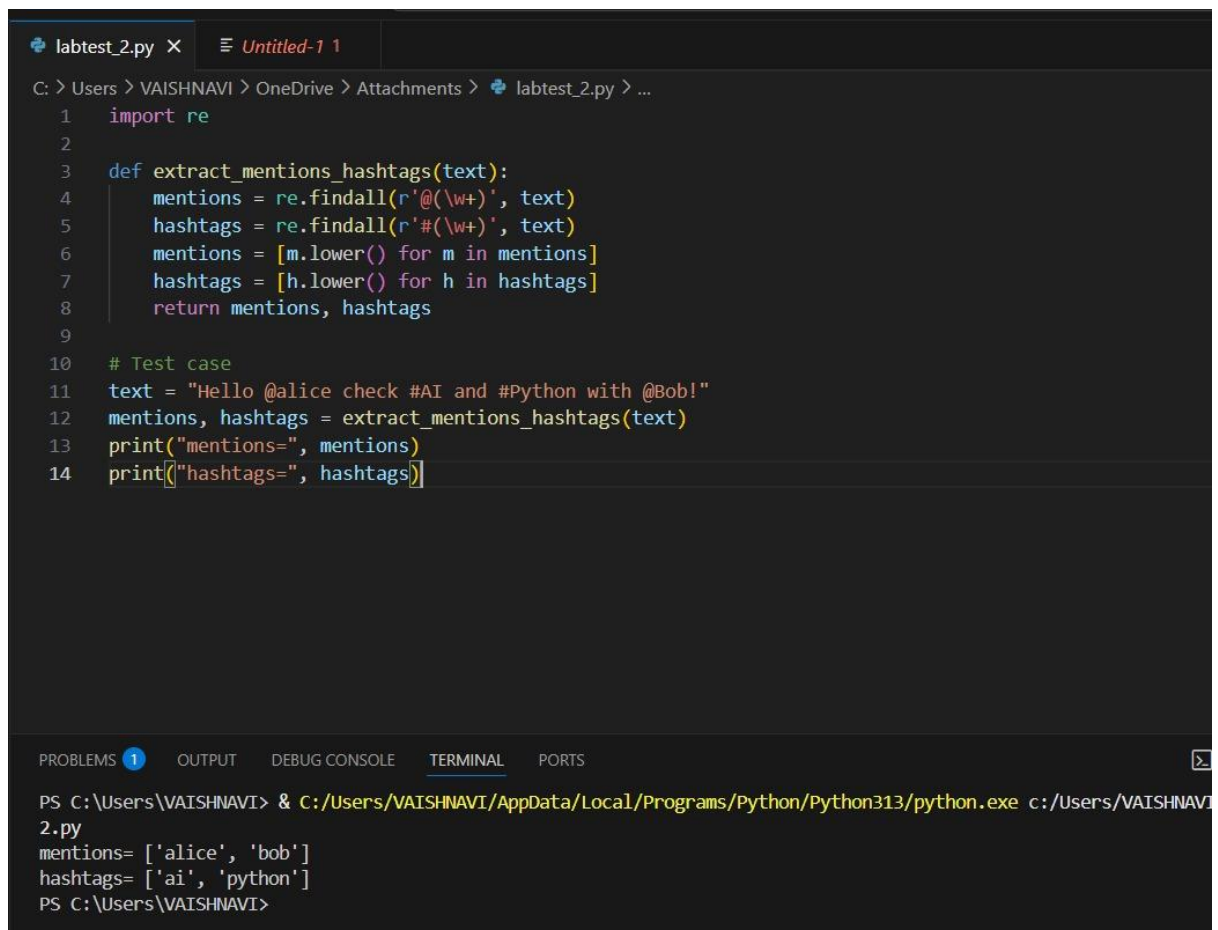
AI ASSISTED CODING LAB TEST – 2

(SET - H)

1.TASK : Regex extract mentions/hashtags, lowercase lists.

PROMPT : Write a Python function to extract all @mentions and #hashtags from a string, ignoring punctuation, and return two lowercase lists

CODE:



```
labtest_2.py x  Untitled-1 1
C: > Users > VAISHNAVI > OneDrive > Attachments > labtest_2.py > ...
1  import re
2
3  def extract_mentions_hashtags(text):
4      mentions = re.findall(r'@(\w+)', text)
5      hashtags = re.findall(r'#(\w+)', text)
6      mentions = [m.lower() for m in mentions]
7      hashtags = [h.lower() for h in hashtags]
8      return mentions, hashtags
9
10 # Test case
11 text = "Hello @alice check #AI and #Python with @Bob!"
12 mentions, hashtags = extract_mentions_hashtags(text)
13 print("mentions=", mentions)
14 print("hashtags=", hashtags)

PROBLEMS 1 OUTPUT DEBUG CONSOLE TERMINAL PORTS
PS C:\Users\VAISHNAVI> & C:/Users/VAISHNAVI/AppData/Local/Programs/Python/Python313/python.exe c:/Users/VAISHNAVI/labtest_2.py
mentions= ['alice', 'bob']
hashtags= ['ai', 'python']
PS C:\Users\VAISHNAVI>
```

OBSERVATION :

- The function `extract_mentions_hashtags(text)` uses regular expressions to find all words that follow @ (mentions) and # (hashtags) in the input string.
- `re.findall(r'@(\w+)', text)` finds all mentions, and `re.findall(r'#(\w+)', text)` finds all hashtags.
- Both lists are converted to lowercase for normalization.
- The function returns two lists: one for mentions and one for hashtags.
- The test cases check that the function works with typical input, punctuation, and edge cases (like no tags).

2 .TASK : Dijkstra from 'A' using heapq.

PROMPT :

Given a graph as a positive-weight adjacency dictionary, implement Dijkstra's algorithm using heapq to find the shortest distances from node 'A'. Return a dictionary of shortest distances. AI should outline the relaxation step.

CODE :

```
import heapq.py > dijkstra
1 import heapq
2
3 def dijkstra(graph, start):
4     # Initialize all distances as infinity, except the start node
5     distances = {node: float('inf') for node in graph}
6     distances[start] = 0
7     heap = [(0, start)]
8     while heap:
9         current_dist, current_node = heapq.heappop(heap)
10        # AI-outlined relaxation: Only process if this is the best known distance
11        if current_dist > distances[current_node]:
12            continue
13        for neighbor, weight in graph[current_node].items():
14            distance = current_dist + weight
15            # Relaxation: If a shorter path is found, update and push to heap
16            if distance < distances[neighbor]:
17                distances[neighbor] = distance
18                heapq.heappush(heap, (distance, neighbor))
19    return distances
20
21 # Example usage
22 graph = {'A':{'B':1,'C':4},'B':{'C':2,'D':5},'C':{'D':1},'D':{}}
23 print(dijkstra(graph, 'A')) # {'A': 0, 'B': 1, 'C': 3, 'D': 4}
```

OBSERVATION :

All nodes start with infinite distance except 'A' (0).

The heap always pops the node with the smallest known distance.

For each neighbor, the algorithm checks if the path through the current node is shorter (relaxation).

If so, it updates the distance and adds the neighbor to the heap.

Continues until all nodes are processed.

The output matches the sample.