

AI ASSISTED CODING

LABTEST-02

NAME:CH.RAMCHARAN

ENROLL.NO:2403A52069

BATCH:04

TASK-01:

Use regex to extract @ mentions and # hashtags (case-insensitive) and return lower case list.

PROMPT:

Generate a python code which accepts a input string as input and performs the operation like segregating the words which starts with the hashtags “#” and mentions “@” and place them in a list in lower case.

CODE:

```
test-2.1.py X test-2.2.py dijkstra.py
labtest_02 > test-2.1.py > ...
1 import re
2
3 def extract_tags(text: str) -> dict[str, list[str]]:
4     """
5     Extracts @mentions and #hashtags from a given text string.
6
7     Args:
8         text: The input string to process.
9
10    Returns:
11        A dictionary with two keys:
12        'mentions': A list of lowercase mentions (without the '@').
13        'hashtags': A list of lowercase hashtags (without the '#').
14    """
15    # Regex to find @mentions. It looks for an '@' followed by one or more
16    # word characters (alphanumeric + underscore).
17    mention_regex = r"@(\w+)"
18    mentions = re.findall(mention_regex, text, re.IGNORECASE)
19
20    # Regex to find #hashtags. It looks for a '#' followed by one or more
21    # word characters.
22    hashtag_regex = r"#(\w+)"
23    hashtags = re.findall(hashtag_regex, text, re.IGNORECASE)
24
25    return {
26        "mentions": [mention.lower() for mention in mentions],
27        "hashtags": [tag.lower() for tag in hashtags],
28    }
29
30 # --- Testing Edge Cases ---
31
32 def run_tests():
33     """Runs a series of tests to validate the extract_tags function."""
34     test_cases = [
35         {
36             "name": "Simple case",
37             "input": "Hello @alice check #AI and #Python with @Bob",
38             "expected": {"mentions": ["alice", "bob"], "hashtags": ["ai", "python"]},
39         },
40     ]
41
42
43
44
45
46
47
48
49
50
51
52
53
54
55
56
57
58
59
60
61
62
63
64
65
66
67
68
69
70
71
72
73
74
75
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

```
labtest_02 > test-2.1.py > ...
32 def run_tests():
33     """Runs a series of tests to validate the extract_tags function."""
34     test_cases = [
35         {
36             "name": "Simple case",
37             "input": "Hello @alice check #AI and #Python with @Bob",
38             "expected": {"mentions": ["alice", "bob"], "hashtags": ["ai", "python"]},
39         },
40         {
41             "name": "Case-insensitivity",
42             "input": "Hi @USER1 and #HelloWorld, how is @user2?",
43             "expected": {"mentions": ["user1", "user2"], "hashtags": ["helloworld"]},
44         },
45         {
46             "name": "Tags with punctuation",
47             "input": "Check this out: @dave, #awesome!",
48             "expected": {"mentions": ["dave"], "hashtags": ["awesome"]},
49         },
50         {
51             "name": "Tags at start/end of string",
52             "input": "@start #middle @end",
53             "expected": {"mentions": ["start", "end"], "hashtags": ["middle"]},
54         },
55         {
56             "name": "No tags",
57             "input": "Just a regular sentence.",
58             "expected": {"mentions": [], "hashtags": []},
59         },
60         {
61             "name": "Tags with numbers and underscores",
62             "input": "Contact @support_team_1 for #issue_42.",
63             "expected": {"mentions": ["support_team_1"], "hashtags": ["issue_42"]},
64         },
65         {
66             "name": "Invalid tags (e.g., in email, with spaces)",
67             "input": "My email is test@example.com. This is not a #bad tag.",
68             "expected": {"mentions": ["example"], "hashtags": ["bad"]},
69         },
70         {
71             "name": "Empty input",
72             "input": "",
73             "expected": {"mentions": [], "hashtags": []},
74         },
75     ]
76
77
78
79
80
81
82
83
84
85
86
87
88
89
90
91
92
93
94
95
96
97
98
99
100
```

```

32 def run_tests():
76
77     all_passed = True
78     for test in test_cases:
79         result = extract_tags(test["input"])
80         if result == test["expected"]:
81             print(f"✅ PASSED: {test['name']}")
82         else:
83             all_passed = False
84             print(f"❌ FAILED: {test['name']}")
85             print(f"    Input:    '{test['input']}'")
86             print(f"    Expected: {test['expected']}")
87             print(f"    Got:      {result}")
88
89     if all_passed:
90         print("\nAll tests passed successfully!")
91     else:
92         print("\nSome tests failed.")
93
94 if __name__ == "__main__":
95     # --- Dynamic Input Usage ---
96     # You can still run the tests by calling run_tests() here if you want.
97     # run_tests()
98
99     # Ask the user for input dynamically
100    user_input = input("Enter a sentence with @mentions and #hashtags: ")
101    extracted_data = extract_tags(user_input)
102    print("\n--- Extracted Tags ---")
103    print(f"Mentions: {extracted_data['mentions']}")
104    print(f"Hashtags: {extracted_data['hashtags']}")
105

```

OUTPUT:

```

PS C:\Users\ramch\OneDrive\Desktop\ai> & C:/Users/ramch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ramch/OneDrive/Desktop/ai/labtest_02/test-2.1.py
Enter a sentence with @mentions and #hashtags: Hello @alice check #AI and #Python with @Bob

--- Extracted Tags ---
Mentions: ['alice', 'bob']
Hashtags: ['ai', 'python']
PS C:\Users\ramch\OneDrive\Desktop\ai>

```

OBSERVATION:

This Python script acts like a social media tag finder for any piece of text. It's designed to read a sentence and intelligently pull out all the words that start with a # (hashtags) or an @ (mentions). The script is smart enough to ignore punctuation attached to the tags and isn't case-

sensitive. After you provide a sentence, it neatly organizes the findings into two separate, lowercase lists—one for all the mentions and one for all the hashtags—and displays them.

TASK-02:

Implement Dijkstra from a source node 'A' to all nodes using a priority queue

PROMPT:

Generate a python script which finds the shortest distance by taking the input as nodes which are graphs and asks from which node you want the shortest path and give the result in the structured dictionary format.

CODE:

```
labtest_02 > djikstra.py > build_graph_dynamically
1 import heapq
2 import math
3
4 def dijkstra_shortest_path(graph: dict[str, dict[str, int]], start_node: str) -> dict[str, int]:
5     """
6     Calculates the shortest paths from a source node to all other nodes in a
7     weighted graph using Dijkstra's algorithm with a priority queue.
8
9     Args:
10         graph: An adjacency list representation of the graph.
11             e.g., {'A': {'B': 1, 'C': 4}, 'B': {'C': 2}, ...}
12         start_node: The node from which to calculate the shortest paths.
13
14     Returns:
15         A dictionary mapping each node to its shortest distance from the start_node.
16         Unreachable nodes will have a distance of infinity.
17     """
18     # 1. Initialization
19     # Initialize distances to all nodes as infinity
20     distances = {node: math.inf for node in graph}
21     # The distance to the start node is 0
22     distances[start_node] = 0
23
24     # Priority queue will store tuples of (distance, node)
25     priority_queue = [(0, start_node)]
26
27     # This implementation doesn't strictly need a 'visited' set because of the
28     # check 'if current_distance > distances[current_node]:', which handles
29     # outdated entries in the priority queue efficiently.
30
31     # 2. Processing Nodes
32     while priority_queue:
33         # Get the node with the smallest distance
34         current_distance, current_node = heapq.heappop(priority_queue)
35
36         # If we've already found a shorter path to this node, skip
37         if current_distance > distances[current_node]:
38             continue
39
40         # Relax edges
41         for neighbor, weight in graph[current_node].items():
42             distance = current_distance + weight
43             if distance < distances[neighbor]:
44                 distances[neighbor] = distance
45                 heapq.heappush(priority_queue, (distance, neighbor))
46
47     return distances
```

```

4 def dijkstra_shortest_path(graph: dict[str, dict[str, int]], start_node: str) -> dict[str, int]:
39
40     # 3. Edge Relaxation
41     for neighbor, weight in graph[current_node].items():
42         new_distance = current_distance + weight
43
44         # If a shorter path to the neighbor is found
45         if new_distance < distances[neighbor]:
46             distances[neighbor] = new_distance
47             # Push the updated path to the priority queue
48             heapq.heappush(priority_queue, (new_distance, neighbor))
49
50     return distances
51
52 def build_graph_dynamically():
53     """Interactively builds a graph from user input."""
54     graph = {}
55     import ast # Use ast for safe literal evaluation
56
57     print("--- Build Your Graph ---")
58     print("You can either:")
59     print(" 1. Paste a full graph dictionary (e.g., {'A':{'B':1}, ...}) and press Enter.")
60     print(" 2. Enter edges one by one in the format 'Node1 Node2 Weight'.")
61     print("Type 'done' when finished with manual entry.")
62
63     while True:
64         entry = input("Enter edge (or 'done'): ").strip()
65         if entry.lower() == 'done':
66             break
67
68         # Try to parse the input as a dictionary literal
69         if entry.startswith('{') and entry.endswith('}'):
70             try:
71                 return ast.literal_eval(entry)
72             except (ValueError, SyntaxError):
73                 print("Invalid dictionary format. Please check your syntax.")
74                 continue
75
76         parts = entry.split()
77
78         print("Invalid format. Please use 'Node1 Node2 Weight'.")
79         continue
80
81         node1, node2, weight_str = parts
82         try:
83             weight = int(weight_str)
84             if weight < 0:
85                 print("Warning: Dijkstra's algorithm may not work correctly with negative weights.")
86
87             # Add nodes to graph if they don't exist
88             if node1 not in graph:
89                 graph[node1] = {}
90             if node2 not in graph:
91                 graph[node2] = {}
92
93             # Add the directed edge
94             graph[node1][node2] = weight
95
96         except ValueError:
97             print("Invalid weight. Please enter an integer.")
98
99     return graph
100
101 if __name__ == "__main__":
102     # Build the graph from user input
103     network_graph = build_graph_dynamically()
104
105     if not network_graph and isinstance(network_graph, dict):
106         print("Graph is empty. Exiting.")
107     else:
108         source = input("Enter the source node: ").strip()
109         if source not in network_graph:
110             print(f"Error: Source node '{source}' not found in the graph.")
111         else:
112             shortest_paths = dijkstra_shortest_path(network_graph, source)
113             print(f"\n--- Results ---")
114             print(f"Graph: {network_graph}")
115             print(f"Shortest paths from node '{source}':")
116             print(shortest_paths)

```

OUTPUT:

```
PS C:\Users\ranch\OneDrive\Desktop\ai> & C:/Users/ranch/AppData/Local/Programs/Python/Python312/python.exe c:/Users/ranch/OneDrive/Desktop/ai/labtest_02/dijkstra.py
--- Build Your Graph ---
You can either:
  1. Paste a full graph dictionary (e.g., {'A':{'B':1}, ...}) and press Enter.
  2. Enter edges one by one in the format 'Node1 Node2 Weight'.
Type 'done' when finished with manual entry.
Enter edge (or 'done'): {'A':{'B':1,'C':4},'B':{'C':2,'D':5},'C':{'D':1},'D':{}}
Enter the source node: A

--- Results ---
Graph: {'A': {'B': 1, 'C': 4}, 'B': {'C': 2, 'D': 5}, 'C': {'D': 1}, 'D': {}}
Shortest paths from node 'A':
{'A': 0, 'B': 1, 'C': 3, 'D': 4}
PS C:\Users\ranch\OneDrive\Desktop\ai>
```

OBSERVATION:

This Python script acts like a GPS for a network you define. It uses Dijkstra's algorithm, a classic method to find the shortest path from a single starting point to all other locations. The script first asks you to build the network map, either by pasting a full dictionary or by adding connections one by one. After you specify your starting "source" node, it calculates the minimum "distance" (or cost) to every other node. The results show a simple list of each destination and the shortest possible route cost to get there from your start point.