Daily Coding Problem #302

Problem

This problem was asked by Uber.

You are given a 2-d matrix where each cell consists of either `/`, `\`, or an empty space. Write an algorithm that determines into how many regions the slashes divide the space.

For example, suppose the input for a three-by-six grid is the following:

```
\    /
 \  /
  \/
```

Considering the edges of the matrix as boundaries, this divides the grid into three triangles, so you should return 3.

Solution

To make things clearer, we can examine what it means for a set of slashes to "divide the space". One way of dividing the space, for example, would be to form a series of slashes, each one starting where the last one ended, that goes from the top to the bottom of the matrix. This sounds like something that a graph algorithm could help with, so let us try turning our input into a graph.

More specifically, we will create an undirected graph whose edges correspond to slashes in our input. We can represent this graph with an adjacency list, where each key is a point where a slash begins, and each value is a list of adjacent points.

```
from collections import defaultdict

def create_graph(matrix):
    graph = defaultdict(list)

    for i, row in enumerate(matrix):
        for j, col in enumerate(row):
            if matrix[i][j] == '/':
                graph[(i, j + 1)].append((i + 1, j))
                graph[(i + 1, j)].append((i, j + 1))
            elif matrix[i][j] == '\\':
                graph[(i, j)].append((i + 1, j + 1))
                graph[(i + 1, j + 1)].append((i, j))

    return graph
```

Now, starting with a given vertex, we can traverse this graph to find all its connected vertices. But how do we know when a new region has formed along the way? There are two cases where this can happen:

- First, if we return to a vertex we have already visited, creating a loop.
- Second, if we create a line joining two separate boundary (or "wall") locations.

The second case requires some care when implementing. In particular, whenever we reach a vertex on a wall, we will increment a counter for the number of wall hits. If this counter is two or greater, then we know we have just created a new division in our matrix.

With these cases in place, we can perform a depth first search from a given start node, counting the regions formed as we proceed.

```
def traverse(start, graph, walls, walls_hit, visited, regions):
    visited.add(start)

    neighbors = graph[start]
    for neighbor in neighbors:
        graph[neighbor].remove(start)
```

```
            graph[neighbor].remove(start)

        for neighbor in neighbors:
            if neighbor in visited:
                regions += 1
            else:
                if neighbor in walls:
                    walls_hit += 1
                    if walls_hit > 1:
                        regions += 1
                regions, walls_hit = traverse(neighbor, graph, walls, walls_hit, visited,
regions)

        return regions, walls_hit
```

Our main code, then, will mostly involve gluing these two pieces together.

We first create the edges and vertices, and make a set of walls that represent the boundary points of the matrix.

Then, we repeatedly choose an unvisited vertex as our start node, and traverse the graph from this vertex, updating our region count with the result. Once we have visited each point, we return the overall number of regions.

```
from itertools import product

def get_regions(matrix):
    graph = create_graph(matrix)
    vertices = list(graph.keys())

    m, n = len(matrix), len(matrix[0])
    walls = set(list(product((0, m), range(n + 1))) + list(product(range(1, m), (0, n))))

    regions = 1
    visited = set()

    while vertices:
        start = vertices[0]
        walls_hit = 1 if start in walls else 0
        regions, _ = traverse(start, graph, walls, walls_hit, visited, regions)
        vertices = [v for v in vertices if v not in visited]
```

```
    return regions
```

To create the graph, we must traverse each row and column of the input matrix, and possibly add a new key in our dictionary for each cell. This will take `O(M * N)` time and space, where `M` is the number of rows and `N` is the number of columns.

In the worst case, every lattice point in the matrix will be a vertex, and there will be a corresponding number of edges. As a result, the time to run our depth first search, and the overall run time of our algorithm, will be `O(M * N)` as well.