PE06: Programming Exercise

Instructions

1. island_counter_main.py
2. island_counter.py

Description:

This assignment is to gain knowledge on the Breadth-first search using a queue. Given a 2D grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water. Note that **"island_counter_main.py" with the "main" method and starter file "island_counter.py" with class has already been provided (download attachment)**.

As part of the assignment, describe how this problem can be solved using a recursive function instead of a queue. Keep in mind to always comment and document your class and methods.

Documentation reference:

- Mertz, J. (n.d.). Documenting Python Code: A Complete Guide. https://realpython.com/documenting-python-code/

Expected result:

island_counter_main.py:

- This is the Main python file that is already provided and contains the main and test procedure, which calls methods implemented on "islandcounter.py" (this is already provided, but please include this file in your submission).

island_counter.py

- This class contains such methods as init, print_grid_visited, count_island, and mark_visited_island. Please keep in mind the following notes for each method during implementation:
    - init(): this method has already been provided.
    - print_grid_visited(): this method has already been provided.
    - count_island(): this method returns the number of islands while marking already visited grounds.
    - mark_visited_island (x, y): this method marks the current location as visited first. After marking the current position, the method looks top/bottom/left/right first to see if it is a valid land and has never been visited **using a queue**. If the island has never been visited, the method marks the location as visited, and the function runs until the queue does not contain any coordinates. **Parameters:** x, y coordinate of the current location

Writeup

As part of the assignment, describe how this problem can be solved using a recursive function instead of a queue.

PE06: Programming Exercise

Instructions

- island_counter_main.py
- island_counter.py

Description

This assignment is to gain knowledge on the Breadth-first search using a queue. Given a 2D grid map of '1's (land) and '0's (water), count the number of islands. An island is surrounded by water and is formed by connecting adjacent lands horizontally or vertically. You may assume all four edges of the grid are all surrounded by water. Note that `island_counter_main.py` with the `main` method and starter file `island_counter.py` with class has already been provided.

Comment and document your class and methods.

Documentation

Mertz, J. (n.d.). Documenting Python Code: A Complete Guide.

Expected result

`island_counter_main.py`

- This is the Main python file that is already provided and contains the main and test procedure, which calls methods implemented on `islandcounter.py`.

```
"""island_counter_main.py

This script demonstrates the usage of the IslandCounterClass to count the number
of islands
in a grid representation of a map.

Given a 2d grid map of '1's (land) and '0's (water), count the number of islands.
An island is surrounded by water and is formed by connecting adjacent lands
horizontally or
vertically. You may assume all four edges of the grid are all surrounded by water.
```

```
    """

    from island_counter import IslandCounterClass

    grid_one_island = [
        ["1", "1", "1", "1", "0"],
        ["1", "1", "0", "1", "0"],
        ["1", "1", "0", "0", "0"],
        ["0", "0", "0", "0", "0"],
    ]

    grid_three_island = [
        ["1", "1", "0", "0", "0"],
        ["1", "1", "0", "0", "0"],
        ["0", "0", "1", "0", "0"],
        ["0", "0", "0", "1", "1"],
    ]


    def main():
        """runs the main function of the program"""
        print("One island test")
        ic_one = IslandCounterClass(grid_one_island)
        print("\n---Before visited---")
        ic_one.print_grid_visited()
        result = ic_one.count_island()
        print("\n---After visited---")
        ic_one.print_grid_visited()
        print(f"\nCounted: {result} \n")

        print("\n\nThree island test")
        ic_three = IslandCounterClass(grid_three_island)
        print("\n---Before visited---")
        ic_three.print_grid_visited()
        result = ic_three.count_island()
        print("\n---After visited---")
        ic_three.print_grid_visited()
        print(f"\nCounted: {result} \n")


    if __name__ == "__main__":
        main()
```

island_counter.py

This class contains such methods as:

1. init
2. print_grid_visited
3. count_island
4. mark_visited_island

Please keep in mind the following notes for each method during implementation:

- `init()`: this method has already been provided.
- `print_grid_visited()`: this method has already been provided.
- `count_island()`: this method returns the number of islands while marking already visited grounds.
- `mark_visited_island(x,y)`: this method marks the current location as visited first.

After marking the current position, the method looks top/bottom/left/right first to see if it is a valid land and has never been visited using a queue. If the island has never been visited, the method marks the location as visited, and the function runs until the queue does not contain any coordinates.

1. ***Parameters***: x, y coordinate of the current location

Outcome

```
"""island_counter.py

This module contains a class, IslandCounterClass, which is responsible for
counting the number of islands in a 2D grid. Each cell in the grid is either
land ('1') or water ('0'). An island is defined as a group of '1's (land)
connected vertically or horizontally. The grid can be of any size.

The IslandCounterClass has the following methods:
- print_grid_visited: Prints the input grid and the cells that have been visited
                                                        during the island count.
- count_island: Counts the number of islands in the grid. It marks cells as
                                     visited when they are part of an island.
- mark_visited_island: Marks all cells of an island as visited. It uses a
                                       Breadth-First Search (BFS) approach.
"""

from typing import List
from collections import deque

WATER = "0"
LAND = "1"
VISITED = True
NOT_VISITED = False
DIRECTIONS = [(0, 1), (1, 0), (0, -1), (-1, 0)]  # Right, Down, Left, Up


class IslandCounterClass:
    """
    A class used to count the number of islands in a 2D grid.

    Attributes:
        grid (List[List[str]]): A 2D list representing a map, where '1' represents
                                land and '0' represents water.
        visited (List[List[bool]]): A 2D list that keeps track of the cells that
                                    have been visited during the island count.

    Methods:
        print_grid_visited(): Prints the input grid and the cells that have been
                              visited during the island count.
        count_island() -> int: Counts and returns the number of islands in the
```

```
    grid.
                                Marks cells as visited when they are part of an
    island.
            mark_visited_island(x: int, y: int): Marks all cells of an island as
    visited.
                               It uses a Breadth-First Search (BFS) approach.
            is_valid_land(x: int, y: int) -> bool: Checks if a cell is a valid part of
    an
                island (i.e., it is within the grid boundaries, is land, and is
    unvisited).
            visit(x: int, y: int): Marks a cell as visited.
        """

        def __init__(self, grid: List[List[str]]) -> None:
            """
            Initializes an instance of the IslandCounterClass with the provided grid.

            Args:
                grid (List[List[str]]): A 2D list representing a map, where '1'
                represents land and '0' represents water.
            """
            self.grid = grid
            self.visited = [[NOT_VISITED for _ in row] for row in grid]

        def print_grid_visited(self) -> None:
            """
            Prints the input grid and the cells that have been visited during
            the island count.
            """
            print("Map:")
            print("\n".join(["\t".join(row) for row in self.grid]))
            print("Visited:")
            print("\n".join(["\t".join(str(cell) for cell in row) for row in
    self.visited]))

        def count_island(self) -> int:
            """
            Counts and returns the number of islands in the grid. Marks cells as
            visited when they are part of an island.

            Returns:
                int: The number of islands in the grid.
            """
            result = 0
            for x, row in enumerate(self.grid):
                for y, cell in enumerate(row):
                    if cell == LAND and not self.visited[x][y]:
                        result += 1
                        self.mark_visited_island(x, y)
            return result

        def mark_visited_island(self, xcrd, ycrd) -> None:
            """
            Marks all cells of an island as visited using a Breadth-First Search (BFS)
    approach.
```

```
        Args:
            x (int): The x-coordinate of the starting cell.
            y (int): The y-coordinate of the starting cell.
        """
        queue = deque([(xcrd, ycrd)])

        while queue:
            current_x, current_y = queue.popleft()
            if self.is_valid_land(current_x, current_y):
                self.visit(current_x, current_y)
                for dx, dy in DIRECTIONS:
                    queue.append((current_x + dx, current_y + dy))

    def is_valid_land(self, xcrd: int, ycrd: int) -> bool:
        """
        Checks if a cell is a valid part of an island (i.e., it is within the
            grid boundaries, is land, and is unvisited).

        Args:
            x (int): The x-coordinate of the cell.
            y (int): The y-coordinate of the cell.

        Returns:
            bool: True if the cell is a valid part of an island, False otherwise.
        """
        return (0 <= xcrd < len(self.grid)
            and 0 <= ycrd < len(self.grid[0])
            and self.grid[xcrd][ycrd] == LAND
            and not self.visited[xcrd][ycrd]
        )

    def visit(self, xcrd: int, ycrd: int) -> None:
        """
        Marks a cell as visited.

        Args:
            x (int): The x-coordinate of the cell.
            y (int): The y-coordinate of the cell.
        """
        self.visited[xcrd][ycrd] = VISITED
```

Using A Recursive Function

Question

Describe how this problem can be solved using a recursive function instead of a queue.

The problem of counting islands in a 2D grid can also be solved using a recursive Depth-First Search (DFS) approach instead of the Breadth-First Search (BFS) approach using a queue. The key difference is that DFS will explore as far as possible along each branch before backtracking, which makes it suitable for this problem where we need to explore all adjacent land cells ('1's) for each starting cell.

Here's an outline of how the mark_visited_island method might be modified to use DFS:

```
def mark_visited_island(self, x: int, y: int) -> None:
    """
    Marks all cells of an island as visited using a Depth-First Search (DFS)
approach.

    Args:
        x (int): The x-coordinate of the starting cell.
        y (int): The y-coordinate of the starting cell.
    """

    if not self.is_valid_land(x, y):
        return

    self.visit(x, y)

    # Recursively call mark_visited_island on neighboring cells
    self.mark_visited_island(x - 1, y)
    self.mark_visited_island(x + 1, y)
    self.mark_visited_island(x, y - 1)
    self.mark_visited_island(x, y + 1)
```

In this recursive approach, the mark_visited_island method first checks if the current cell is a valid part of an island using the is_valid_land method. If it is, it marks the cell as visited using the visit method, and then recursively calls mark_visited_island on the neighboring cells. This continues until all connected land cells have been visited, at which point the recursion ends and the method returns.

The main advantage of the recursive DFS approach over the BFS approach is its simplicity. The recursive method is more straightforward to implement and understand because it doesn't require managing a queue. However, DFS can consume more stack space than BFS if the maximum depth is large. This could potentially lead to a stack overflow for very large grids or very complex island shapes.