Coding: Game Class Functionality

Adding Functionality to the Game Class

We will use the interface for adversarial search problems defined in Chapter 5.1 of Artificial Intelligence: a Modern Approach, which includes the following five functions:

- `player()`: return the active player in the current state
- `actions()`: return a list of the legal actions in the current state
- `result(action)`: return a new state that results from applying the given action in the current state
- `terminal_test()`: return True if the current state is terminal, and False otherwise
- `utility(player)`: return +inf if the game is terminal and the specified player wins, return -inf if the game is terminal and the specified player loses, and return 0 if the game is not terminal (**NOTE:** You do **not** need to implement this function now)

We will extend that interface with one additional domain specific method, which will simplify implementing several techniques later in the lesson:

- `liberties(loc)`: return a list of cells in the neighborhood of the specified location that are open

Implement these functions for your game class below. If you get stuck, flip over to the solution to see one possible implementation.

Note: Don't be afraid to add additional functions to the class or module to help complete the two required tasks.

gamestate.py | testcode.py | solution.py

```
from copy import deepcopy

x_cells, y_cells = 3, 2
RAYS = [(1, 0), (1, -1), (0, -1), (-1, -1), (-1, 0), (-1, 1), (0, 1), (1, 1)]


class GameState:
    """
    Attributes
    ----------
    TODO: Copy in your implementation from the previous quiz
    """
    def __init__(self):
        # TODO: Copy in your implementation from the previous quiz
        self.board = [[0]*y_cells for i in range(x_cells)]
        self.board[-1][-1] = 1
        self.player_locations = [None, None]
        self.parity = 0

    def actions(self):
        """ Return a list of legal actions for the active player

        You are free to choose any convention to represent actions,
        but one option is to represent actions by the (row, column)
        of the endpoint for the token. For example, if your token is
        in (0, 0), and your opponent is in (1, 0) then the legal
        actions could be encoded as (0, 1) and (0, 2).
        """
```

The example solution is intended to be simple to reason about; it is not designed for high performance. Returning copies of the game state when forecasting moves has significant overhead (especially in Python), and returning a complete list of all legal moves is inefficient (a generator would be better), but both conventions simplify the underlying implementation & interface for the minimax algorithm. We will use a similar interface in the project.

When performance matters, it is typical to use **bitboards** (which *are* used in the project library).

NEXT