TicTacToe

Complete the implementations of player, actions, result, winner, terminal, utility, and minimax.

The player function should take a board state as input, and return which player's turn it is (either X or O).

- In the initial game state, X gets the first move. Subsequently, the player alternates with each additional move.
- Any return value is acceptable if a terminal board is provided as input (i.e., the game is already over).

```
def player(board):
    """
    Returns player who has the next turn on a board.
    """
    X_count = 0
    O_count = 0
    for row in board[:]:
        for unit in row:
            X_count = X_count + int(unit=="X")
            O_count = O_count + int(unit=="O")

    if X_count == O_count:
        return "X"
    elif X_count > O_count:
        return "O"
```

The code defines a function named `player` that determines which player's turn it is in the game based on the current state of the board.

- `def player(board):`: This line defines a function named `player` that takes a `board` as an argument. The purpose of this function is to determine which player has the next turn based on the current state of the board.
- `""" ... """`: This is a docstring, providing a brief description of the function's purpose.
- `X_count = 0` and `O_count = 0`: These lines initialize two variables, `X_count` and `O_count`, to keep track of the number of X and O marks on the board.
- `for row in board[:]:`: This loop iterates over each row in the `board`.
- `for unit in row:`: This nested loop iterates over each cell (unit) within the current row.

- `X_count = X_count + int(unit=="X")`: This line increments the `X_count` variable if the current cell contains an "X". The `int(unit=="X")` expression evaluates to 1 if the condition is true and 0 otherwise.
- `O_count = O_count + int(unit=="O")`: Similarly, this line increments the `O_count` variable if the current cell contains an "O".
- After iterating through all cells of the board, the counts of X and O marks are calculated.
- The following `if` conditions determine which player's turn it is:
  - `if X_count == O_count:`: If the counts are equal, it means it's X's turn since X goes first.
  - `elif X_count > O_count:`: If X's count is greater than O's count, it means it's O's turn.
- Depending on the counts, the function returns the corresponding player's mark ("X" or "O").

This function calculates the player's turn by comparing the counts of X and O marks on the board. If X's count is greater than O's count, it's O's turn, and if the counts are equal, it's X's turn.

The actions function should return a set of all of the possible actions that can be taken on a given board.

- Each action should be represented as a tuple (i, j) where i corresponds to the row of the move (0, 1, or 2) and j corresponds to which cell in the row corresponds to the move (also 0, 1, or 2).
- Possible moves are any cells on the board that do not already have an X or an O in them.
- Any return value is acceptable if a terminal board is provided as input.

The code defines a function named `actions` that determines all possible actions (i.e., valid moves) that can be taken on the current state of the Tic-Tac-Toe board.

```
def actions(board):
    """
    Returns set of all possible actions (i, j) available on the board.
    """
    possible_actions = []
    for row_no, row in enumerate(board):
        for col_no, unit in enumerate(row):
            if unit == None:
                possible_actions.append((row_no, col_no))

    return possible_actions
```

- `def actions(board):`: This line defines a function named `actions` that takes a `board` as an argument. The purpose of this function is to find all the possible actions (valid moves) that can be taken on the current state of the board.
- `""" ... """`: This is a docstring, providing a brief description of the function's purpose.

- `possible_actions = []`: This initializes an empty list to store the possible actions as tuples of `(row_no, col_no)`.
- `for row_no, row in enumerate(board):`: This loop iterates over each row in the `board`, while keeping track of the row number using the `enumerate` function.
- `for col_no, unit in enumerate(row):`: This nested loop iterates over each cell (unit) within the current row, while keeping track of the column number using the `enumerate` function.
- `if unit == None:`: This line checks if the current cell is empty, indicated by `None`.
- `possible_actions.append((row_no, col_no))`: If the cell is empty, a tuple `(row_no, col_no)` representing the coordinates of the empty cell is appended to the `possible_actions` list.
- After iterating through all cells of the board, the function returns the list of `possible_actions`, which represents all the valid moves that can be made on the current state of the board.

In summary, the `actions` function identifies all the empty cells on the board and returns a list of tuples, where each tuple contains the row and column indices of an empty cell. These tuples represent all possible valid moves that can be made in the current game state.

The result function takes a board and an action as input, and should return a new board state, without modifying the original board.

- If action is not a valid action for the board, your program should raise an exception.
- The returned board state should be the board that would result from taking the original input board, and letting the player whose turn it is make their move at the cell indicated by the input action.
- Importantly, the original board should be left unmodified: since Minimax will ultimately require considering many different board states during its computation. This means that simply updating a cell in board itself is not a correct implementation of the result function. You'll likely want to make a deep copy of the board first before making any changes.

The code defines a function named `result` that calculates the resulting state of the Tic-Tac-Toe board after making a specific move (action).

```
def result(board, action):
    """
    Returns the board that results from making move (i, j) on the board.
    """
    (x, y) = action
    dummy = deepcopy(board)
    if dummy[x][y] == None:
        dummy[x][y] = player(board)
    else:
        raise Exception("Invalid Move")
```

```
    return dummy
```

- `def result(board, action):`: This line defines a function named `result` that takes a `board` and an `action` (a tuple representing the row and column indices of a move) as arguments. The purpose of this function is to compute the new board state resulting from the given move.
- `""" ... """`: This is a docstring, providing a brief description of the function's purpose.
- `(x, y) = action`: This line unpacks the `action` tuple into two variables, `x` and `y`, which represent the row and column indices of the move.
- `dummy = deepcopy(board)`: This line creates a deep copy of the input `board` using the `deepcopy` function from the `copy` module. This copy is made to ensure that the original board remains unmodified while calculating the result of the move.
- `if dummy[x][y] == None:`: This condition checks if the cell at the specified row `x` and column `y` is empty (None). If it is, it means the move is valid.
- `dummy[x][y] = player(board)`: If the cell is empty, the code assigns the current player's mark (determined by the `player` function) to the specified cell in the copied board.
- `else:`: If the cell is not empty, meaning the move is invalid, this block is executed.
- `raise Exception("Invalid Move")`: This line raises an exception with the message "Invalid Move" to indicate that the specified move is not valid.
- `return dummy`: After performing the move, the function returns the new board state represented by the `dummy` board.

In summary, the `result` function takes a board and an action, makes the specified move on a copy of the board, and returns the new board state after the move. If the move is invalid (the specified cell is already occupied), an exception is raised.

The winner function should accept a board as input, and return the winner of the board if there is one.

- If the X player has won the game, your function should return X. If the O player has won the game, your function should return O.
- One can win the game with three of their moves in a row horizontally, vertically, or diagonally.
- You may assume that there will be at most one winner (that is, no board will ever have both players with three-in-a-row, since that would be an invalid board state).
- If there is no winner of the game (either because the game is in progress, or because it ended in a tie), the function should return None.
  Certainly! The provided code defines a function named `winner` that determines the winner of the Tic-Tac-Toe game based on the current state of the board.

```
def winner(board):
    """
```

```
    Returns the winner of the game, if there is one.
    """
    for row in board:  # Checking row wins
        if sum(unit == "X" for unit in row) == 3:
            return "X"
        elif sum(unit == "O" for unit in row) == 3:
            return "O"

    for i in [0, 1, 2]:  # Checking column wins
        if sum(row[i] == "X" for row in board) == 3:
            return "X"
        if sum(row[i] == "O" for row in board) == 3:
            return "O"

    if sum([board[0][0] == "X", board[1][1] == "X", board[2][2] == "X"]) ==
3:  # Checking Diagonal Wins
        return "X"

    if sum([board[0][2] == "X", board[1][1] == "X", board[2][0] == "X"]) ==
3:  # Checking Diagonal Wins
        return "X"

    if sum([board[0][0] == "O", board[1][1] == "O", board[2][2] == "O"]) ==
3:
        return "O"

    if sum([board[0][2] == "O", board[1][1] == "O", board[2][0] == "O"]) ==
3:  # Checking Diagonal Wins
        return "O"

    return None
```

- `def winner(board):`: This line defines a function named `winner` that takes a `board` as an argument. The purpose of this function is to determine the winner of the game based on the current state of the board.
- `""" ... """`: This is a docstring, providing a brief description of the function's purpose.
- The function checks for various win conditions: rows, columns, and diagonals.
    - `for row in board:`: This loop iterates over each row in the board to check for row wins.
        - `sum(unit == "X" for unit in row) == 3`: This condition checks if there are three "X" marks in the current row, indicating an "X" win.
        - Similarly, the `sum(unit == "O" for unit in row) == 3` condition checks for an "O" win in the current row.

- `for i in [0, 1, 2]:`: This loop iterates over the indices [0, 1, 2] to check for column wins.
  - `sum(row[i] == "X" for row in board) == 3`: This condition checks if there are three "X" marks in the current column, indicating an "X" win.
  - Similarly, the `sum(row[i] == "O" for row in board) == 3` condition checks for an "O" win in the current column.
- The next two conditions, with `sum([...])`, check for diagonal wins.

- Finally, if none of the above conditions are satisfied, the function returns `None`, indicating there is no winner yet.

In summary, the `winner` function analyzes the board to determine if there is a winner (either "X" or "O") based on rows, columns, and diagonals. If no winner is found, it returns `None`.

The terminal function should accept a board as input, and return a boolean value indicating whether the game is over.

- If the game is over, either because someone has won the game or because all cells have been filled without anyone winning, the function should return True.
- Otherwise, the function should return False if the game is still in progress.

The code defines a function named `terminal` that determines whether the Tic-Tac-Toe game is over or still in progress based on the current state of the board.

```
def terminal(board):
    """
    Returns True if game is over, False otherwise.
    """
    None_count = 0
    for row in board:
        None_count = None_count + sum(unit == None for unit in row)

    if None_count == 0 or winner(board) != None:
        return True
    else:
        return False
```

- `def terminal(board):`: This line defines a function named `terminal` that takes a `board` as an argument. The purpose of this function is to determine whether the game is over or still in progress based on the current state of the board.
- `""" ... """`: This is a docstring, providing a brief description of the function's purpose.
- `None_count = 0`: This initializes a variable `None_count` to keep track of the number of empty cells (None) on the board.
- `for row in board:`: This loop iterates over each row in the board.

    - `None_count = None_count + sum(unit == None for unit in row)`: This line calculates the sum of the number of empty cells (None) in the current row and adds it to the `None_count`.
- `if None_count == 0 or winner(board) != None:`: This line checks two conditions:
    - `None_count == 0`: If there are no empty cells on the board, the game is over due to a tie.
    - `winner(board) != None`: If there is a winner (either "X" or "O") based on the `winner` function, the game is over.
- If either of the above conditions is true, the function returns `True`, indicating that the game is over.
- `else:`: If neither of the above conditions is true, the function returns `False`, indicating that the game is still in progress.

In summary, the `terminal` function checks if the game is over by either all cells being filled (a tie) or there being a winner. If the game is over, it returns `True`; otherwise, it returns `False`.

The utility function should accept a terminal board as input and output the utility of the board.

- If X has won the game, the utility is 1. If O has won the game, the utility is -1. If the game has ended in a tie, the utility is 0.
- You may assume utility will only be called on a board if terminal(board) is True.

The code defines a function named `utility` that calculates the utility value of the Tic-Tac-Toe game for the current state of the board.

```
def utility(board):
    """
    Returns 1 if X has won the game, -1 if O has won, 0 otherwise.
    """
    if winner(board) == "X":
        return 1
    elif winner(board) == "O":
        return -1
    else:
        return 0
```

- `def utility(board):`: This line defines a function named `utility` that takes a `board` as an argument. The purpose of this function is to determine the utility value of the game for the current state of the board.
- `""" ... """`: This is a docstring, providing a brief description of the function's purpose.
- The function uses the `winner` function to check if there is a winner on the board.
- `if winner(board) == "X":`: If the winner is "X," the function returns 1, indicating that the utility value for player X is 1.

- `elif winner(board) == "O":`: If the winner is "O," the function returns -1, indicating that the utility value for player O is -1.
- `else:`: If there is no winner (a tie), the function returns 0, indicating a utility value of 0.

In summary, the `utility` function calculates the utility value of the game for the current state of the board. It returns 1 if "X" has won, -1 if "O" has won, and 0 for a tie.

The minimax function should take a board as input, and return the optimal move for the player to move on that board.

- The move returned should be the optimal action (i, j) that is one of the allowable actions on the board. If multiple moves are equally optimal, any of those moves is acceptable.
- If the board is a terminal board, the minimax function should return None.

The code defines a minimax algorithm for finding the optimal move in a Tic-Tac-Toe game based on the current state of the board. The algorithm is implemented using two helper functions: `max_value` and `min_value`.

```
def minimax(board):
    """
    Returns the optimal action for the current player on the board.
    """
    if terminal(board):
        return None
    else:
        if player(board) == X:
            value, move = max_value(board)
            return move
        else:
            value, move = min_value(board)
            return move


def max_value(board):
    if terminal(board):
        return utility(board), None

    v = float('-inf')
    move = None
    for action in actions(board):
        aux, act = min_value(result(board, action))
        if aux > v:
            v = aux
```

```
            move = action
            if v == 1:
                return v, move

    return v, move


def min_value(board):
    if terminal(board):
        return utility(board), None

    v = float('inf')
    move = None
    for action in actions(board):
        aux, act = max_value(result(board, action))
        if aux < v:
            v = aux
            move = action
            if v == -1:
                return v, move

    return v, move
```

Here's how the code works:

- `def minimax(board):`: This function is the main minimax algorithm. It determines the optimal action for the current player based on the current state of the board. It alternates between calling `max_value` for player "X" and `min_value` for player "O" to evaluate the potential outcomes of each move.
- `def max_value(board):` and `def min_value(board):`: These functions implement the max-value and min-value components of the minimax algorithm. They recursively explore the possible moves and their outcomes for the current player ("X" or "O") by evaluating the utility of terminal boards or making the best possible moves.
- Both `max_value` and `min_value` functions return a tuple containing two values: the utility value of the current board state and the corresponding move (action). If a move leads to a win, the function immediately returns that move, as the best outcome is already achieved.
- `aux, act = max_value(result(board, action))` and `aux, act = min_value(result(board, action))`: These lines call the corresponding min-value or max-value functions recursively to explore the outcomes of the opponent's possible moves.
- `v == 1` and `v == -1`: These conditions check if a winning move has been found. If a move leads to a win, there's no need to explore further, and the function immediately returns the winning move.

In summary, the minimax algorithm, along with its helper functions, recursively explores all possible moves on the Tic-Tac-Toe board to find the optimal move for the current player, taking into account both the player's and opponent's strategies.
