Connect N Game: Homework Assignment

Objective:

Design and implement a multi-round console-based game called "Connect N". The goal for players is to align their respective markers in N consecutive positions in any direction (horizontally, vertically, or diagonally) on a board.
Related reading:
https://jellis18.github.io/post/2022-01-15-access-modifiers-python/
Notice, that the column number should be 0 indexed.
Notice, that the class, method, and variable name need to exactly match the information in the doc.
For example:

```
Class Player:
      def display():
            …
```

Class Specifications:

a) Notation Class:

- Objective: Define an enumeration to represent the three possible notations on the board.
  - `EMPTY`: Represents an unoccupied spot on the board.
    The default value should be **0**.
  - `PLAYER1`: Represents Player 1's marker on the board.
    The default value should be **1**.
  - `PLAYER2`: Represents Player 2's marker on the board.
    The default value should be **2**.

b) Player Class:

- Objective: Represent each player in the game, keeping track of their details and performance.
  Attributes:
  - `__playerName`: a private string representing the name of the player.
  - `__playerNotation`: a private enumeration value of Notation representing the marker of the player.
    For example `Notation.PLAYER1`
  - `__curScore`: a private integer representing the current score of the player(The number of rounds the current player has won).
- Methods:

- `__init__(self, playerName: str, playerNotation: Notation, curScore: int) -> None`: Initialize the player object.
- `display() -> str`: Display and return the player's details in a string format. Display the player name, current Score, and player Notation
- `addScoreByOne() -> None`: Increment the player's score by one.
- `getScore() -> int`: Return the player's current score.
- `getName() -> str`: Return the player's name.
- `getNotation() -> Notation`: Return the player's marker notation.

c) Board Class:

- Objective: Handle the game board operations and check for winning conditions.
  Attributes:
  - `__rowNum`: a private integer representing the number of rows on the board.
  - `__colNum`: a private integer representing the number of columns on the board.
  - `__grid`: a private 2D list representing the current state of the board.
- Methods:
  - `__init__(self, rowNum: int, colNum: int) -> None`: Initialize the board with empty notations.
    For example:
    If temp = Board(2,2)
    temp. `__grid is`
    `[[EMPTY Notation,EMPTY Notation ],`
    `[EMPTY Notation, EMPTY Notation]]`
    The empty notation is what you define in the Notation Class.
  - `initGrid() -> None`: Reset the grid to its initial empty state.
  - `getColNum() -> int`: Return the number of columns on the board.
  - `placeMark(colNum: int, mark: Notation) -> bool`: Attempt to place a player's mark in a column, returning success status.
    1) Check if the provided colNum is valid.
       - It should be within the range of the board's columns .
       - It should not be a negative integer.

       If not, print an error message and return False.
    2) Check if the top cell of the specified column is already occupied. If it is, print "column is full" and return False.
    3) Validate if the mark provided is not Notation.EMPTY. If it is, print "invalid marker" and return False.
    4) Iterate through the cells of the specified column from bottom to top. Place the mark in the first empty cell found and return True.
  - `checkFull() -> bool`: Check if the board is completely filled.
    If Full returns true, otherwise false.
  - `display() -> None`: Display the current state of the board.
    1) Initialize an empty string, `boardStr`, to store the board representation.

2) Loop through each row of the board.
3) For each cell in the row: Append 'O' to `boardStr` if the cell contains Notation.EMPTY.
4) Append 'R' to `boardStr` if the cell contains `Notation.PLAYER1`.
5) Append 'Y' to `boardStr` if the cell contains `Notation.PLAYER2`.
6) After processing each row, append a newline character to `boardStr`.
7) Print "Current Board is" followed by the generated `boardStr`.

- `__checkWinHorizontal(target: int) -> Optional[Notation]`: Check and return winner Notation based on horizontal alignment.
  `target represents the number of markers needed to become a winner`
  If there is a winner return the `winner notation(eg: Notation.PLAYER2)`, if not return `None`.
- `__checkWinVertical(target: int) -> Optional[Notation]`: Check and return winner based on vertical alignment.
  `target represents the number of markers needed to become a winner`
  If there is a winner return the `winner notation`, if not return `None`.
- `__checkWinDiagonal(target: int) -> Optional[Notation]`: Check and return winner based on diagonal alignments.
  `target represents the number of markers needed to become a winner`
  If there is a winner return the `winner notation`, if not return `None`.
  Notice you also need to check anti Diagonal in this function, and please write some other private methods to achieve it.
  For example:
  ```
  def __checkWinOneDiag(self,target,rowNum,colNum)->
  Optional[Notation]
  def__checkWinAntiOneDiag(self,target,rowNum,colNum)->
  Optional[Notation]
  ```

- `checkWin(target: int) -> Optional[Notation]`: Check for any winning condition and return the `winner's notation` or `None`. The `checkWin` should call the other checker functions and return the winner. Return `None` if there is not.

d) Game Class:

- Objective: Manage the game's logic, including player turns, rounds, and overall game progression.
  Notice player1 always goes first
  Attributes:
  - `__board`: an instance of the Board class.

- `__connectN`: an integer representing the number of consecutive markers required to win.
- `__targetScore`: an integer representing the score target to win the entire game.
- `__playerList`: a list of two Player instances.
- `__curPlayer`: a reference to the current player instance whose turn it is.
  Hint: First elements in `__playerList`

- Methods:
  - `__init__(self, rowNum: int, colNum: int, connectN: int, targetScore: int, playerName1: str, playerName2: str) -> None`: Initialize the game.
    1) Initialize a Borad instance with (`rowNum,colNum`) and set it to `__board` attribute.
    2) Initialize the first player instance with `Notation.Player1` and `playerName1.`
    3) Initialize the Second player instance with `Notation.Player2` and `playerName2.`
    4) Store the instances in `__playerList` attribute.
    5) Set the `__curPlayer` to the first player instance reference.

  - `__playBoard(curPlayer: Player) -> None`: Handle a player's turn to play on the board.
    1) Using a local variable `isPlaced: bool` and a while loop to control the following behavior
    2) Prompt the player to input a column number.
    3) Validate the input: check if it's a valid integer and within the allowed range.
       - It should not be any English letter.
       - It should not be an integer with leading 0, such as 007.
    4) If valid, attempt to place the mark in the column.
    5) If placement is successful, set isPlaced to True to stop the while loop. If invalid or the column is full, re-prompt the player.
  - `__changeTurn() -> None`: Switch the active player by switching the `__curPlayer` attribute.
  - `playRound()-> None`: Handle a full round of gameplay.
    1) Start by initializing `curWinnerNotation` to None. This variable will be used to keep track of the winner of the current round, if any.
       It also serves the control the flow of the following step:
       `For example: while not curWinnerNotation:`
    2) Initialize the game board by invoking the `initGrid` method on `self.__board`.
    3) Set the current player (`self.__curPlayer`) to the first player in `self.__playerList.`

4) Display messages indicating the start of a new round.
   For example: `Starting a new round`
5) Enter a loop that continues until there is a winner (`curWinnerNotation` is not None).
    1) Display the current player's details using the `display` method of the player object.
    2) Display the current state of the board using the `display` method of the board object.
    3) Let the current player play their turn using the `__playBoard` method.
    4) Check if there is a winner using the `checkWin` method on `self.__board`. If there's a winner, update curWinnerNotation with the winner's notation.
    5) If a winner is detected:
        - Announce the winner by printing their name.
        - Display the board's final state for this round.
        - Increase the winner's score by one.
        - Exit the round (end the loop).

6) If no winner is detected and the board is full (checked using `checkFull` method), announce that the board is full and that there's no winner for this round. Then, exit the round(end the loop).

7)If the round hasn't ended, switch to the other player using the `__changeTurn` method.

- `play() -> None`: Manage the entire game until a player reaches the target score or quits.
    1) Use a `while` loop to continue playing rounds until one of the players reaches or exceeds the target score `(self._targetScore)`. You can obtain a player's score using the `getScore` method on the player object.
    2) Inside the loop, invoke the `playRound` method to facilitate a single round of gameplay. This method handles the turn-by-turn gameplay and updates scores as needed.
    3) Once a player reaches the `__targetScore` exit the loop.
    4) Display the message "Game Over" to indicate the end of the game.
    5) Finally, display the details (including the final scores) of both players using the `display` method of each player object.

Homework Instruction: Unit Testing (if needed)

Objective: Implement unit tests to ensure the functionality and reliability of your methods.

For each method you've implemented in your classes, write a corresponding unit test. Each test should cover both typical use cases and edge cases. Consider scenarios where inputs might be at their limits or where you might expect potential errors.

Use assertions to validate that the output of your methods matches the expected outcomes. For instance, if a method is supposed to return True, ensure your test checks for this.

Ensure you have tests that handle exceptional situations. If a method is expected to raise an error under certain conditions, your test should confirm that it does.

After writing your tests, run them to confirm that all tests pass. If any tests fail, revisit your methods to troubleshoot and refine your implementation.

Evaluation: Your submission will be graded based on the completeness and thoroughness of your tests, as well as how well they validate the functionality of the corresponding methods. Successful unit tests should give confidence that your methods operate correctly across a wide range of scenarios.