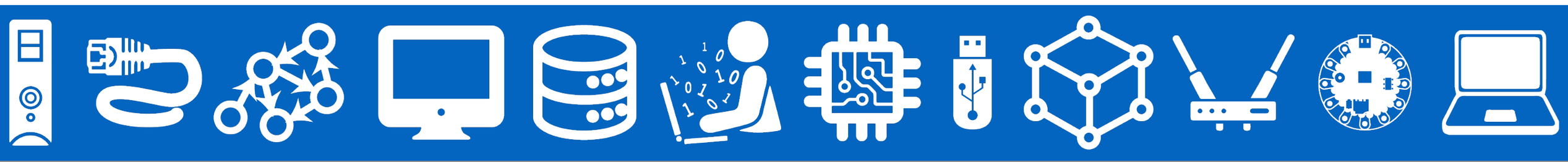
CS134:
Tic Tac Toe: `TTTBoard`



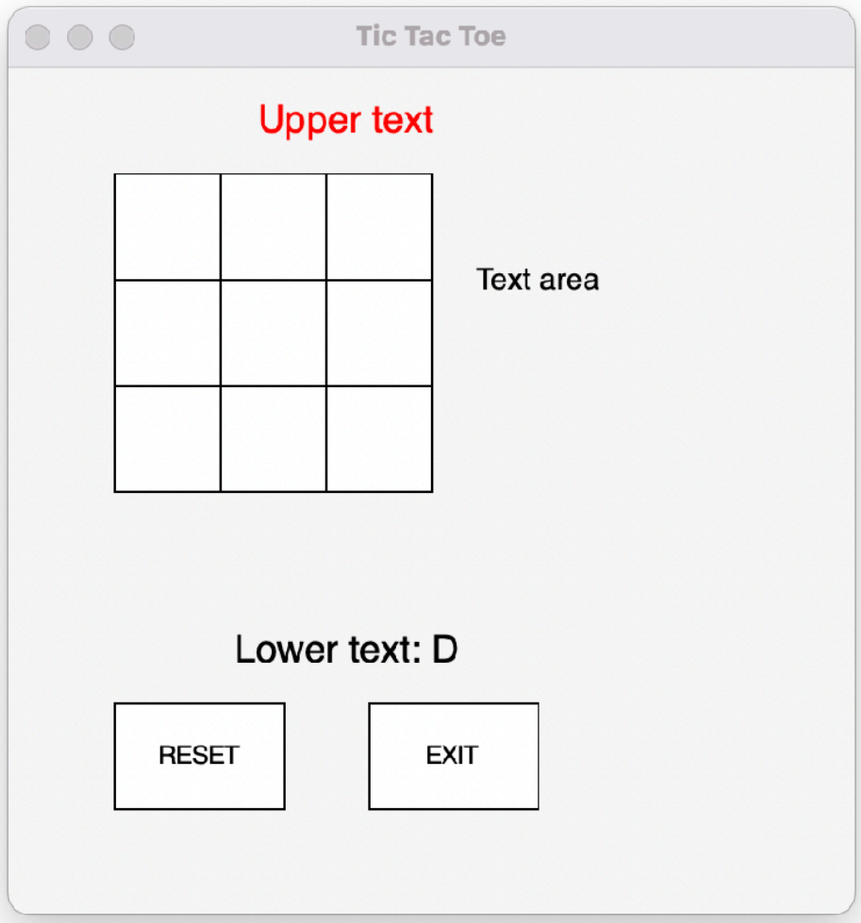
Announcements & Logistics

- **Lab 8** due Weds/Thurs @ 10 pm
- **Lab 9 Boggle**: two-week lab
  - **Part 1** due next Wed/Thur 10 pm
    - Feedback given; you are able to correct/fix Part 1 before it is graded in Part 2
  - **Part 2** due Nov 21/22

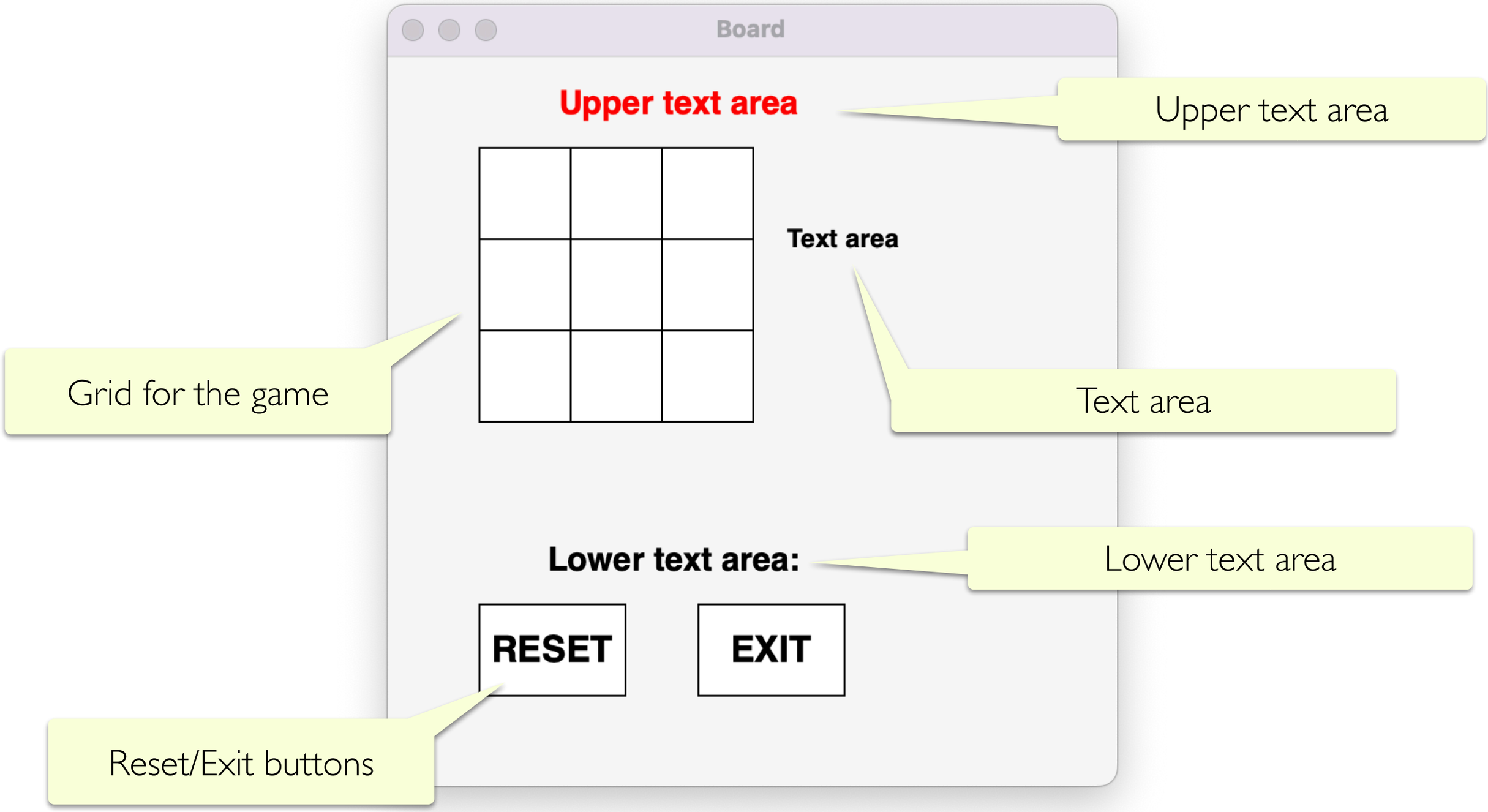
Do You Have Any Questions?

Last Time: Board class

- Basic features of our game board:
    - Text areas: above, below, right of grid
    - Grid of squares of set size: rows x cols
    - Reset and Exit buttons
        - React to mouse clicks (we'll discuss this)
- These are all **graphical** (GUI) components
    - Used graphics package to create rectangles/window/text
    - `object.draw(win)` draws object on graphical window `win`



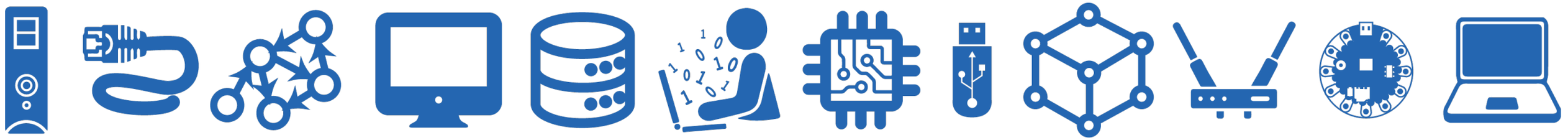
Board Class: All the Pieces



Today's Plan

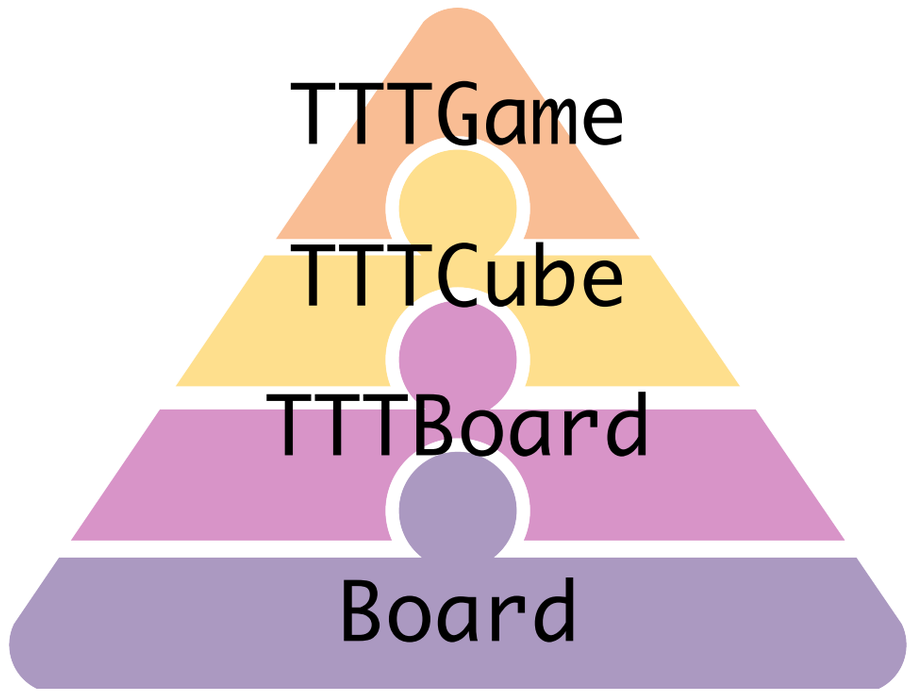
- Use our design for the **`Board`** class to build a Tic Tac Toe board that inherits from **`Board`**
    - How can we extend Board for a Tic Tac Toe (TTT) game?
    - What TTT-specific new methods/attributes do we need?
- Move up to the next layer: **`TTTCube`**
    - What attributes/methods can we use to implement functionality of a single Tic Tac Toe Cube (letter)?

TTTBoard Class



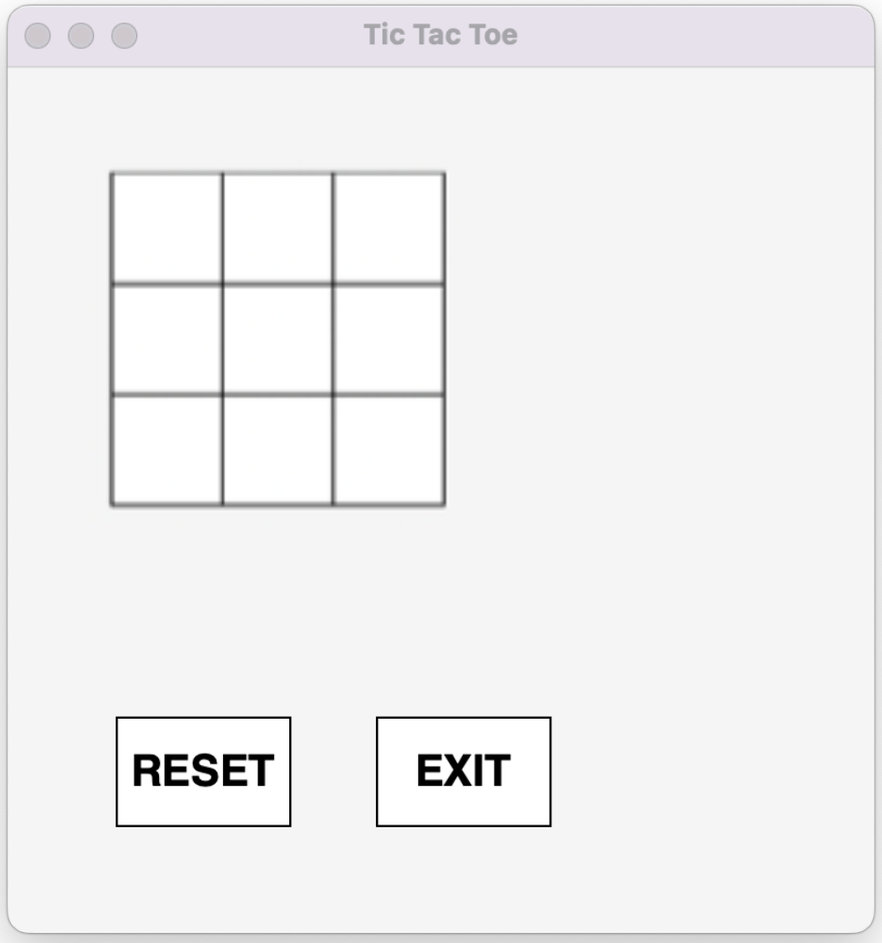
Moving up: TTTBoard

- Although our `Board` class provides a lot of useful functionality, there are some Tic Tac Toe specific features we need to support
- We can do this by ***inheriting*** from the `Board` class
- We can take advantage of all of the methods and attributes defined in `Board` and add any (specific) extras we may need for TTT
- What extra attributes and/or methods might be useful?



`TTTBoard`: Design

- Think of the grid composed of `TTTCubes`
  - Initially populate grid with `TTTCubes` that show "empty" spaces
- Let's think about the Board state in the "middle of the game"
- What are some helper methods that can help get/set the game state?
  - Reset/draw/update board
  - Get TTTCubes at specific grid locations



TTTCube

- To *use* TTTCube, we just need to know its documentation (not how it is implemented) — this is abstraction at work!
- We will explore the implementation later

```
CLASSES
    builtins.object
        TTTCube

    class TTTCube(builtins.object)
    |  TTTCube(letter='')
    |
    |  A TTT Cube has several attributes that define it:
    |  *  _letter:  denotes the letter 'X', 'O', or '-' on the face of a TTT Cube
    |
    |  Methods defined here:
    |
    |  __init__(self, letter='')
    |      Initializes the TTTCube with a visible letter
    |
    |  __repr__(self)
    |      Debugging representation of a TTTCube
    |
    |  __str__(self)
    |      String representation of a TTTCube
    |
    |  get_letter(self)
    |      Returns the visible letter on the cube
    |
    |  set_letter(self, char)
    |      Updates the visible letter on the cube if it is valid (X, O, or '')
```

TTTCube

- To *use* TTTCube, we just need to know its documentation (not how it is implemented) — this is abstraction at work!
- To use TTTCubes we need to know that they have:
    - a `letter` (string): Going to be "X", "O", or "" in TTT
    - methods for getting and setting `letter`

Initializing the TTT Board

- What attributes do we need?
  - Everything inherited from **`Board`** class
  - A grid: a list of lists of **`TTTCubes`**

Inherit from Board

Call parent's __init__ method

Populate grid with empty TTTCubes

```
class TTTBoard(Board):
    # inherits _grid made up of _rows and _cols and other graph
    # new attribute:  _cubes (list of TTTCubes)
    __slots__ = ['_cubes']

    def __init__(self, win):
        """Initializes a Tic-tac-toe board with an empty grid""
        # call Board init with appropriate TTT grid size
        super().__init__(win, rows=3, cols=3)

        # initialize new attribute to build 2-D list
        # of TTTCubes
        self._cubes = []
        for row in range(self._rows):
            cube_row = []
            for col in range(self._cols):
                # add TTTCube to row
                cube_row.append(TTTCube())

            # add column to grid
            self._cubes.append(cube_row)

        # display the cubes on the board
        self.place_cubes_on_board()
```

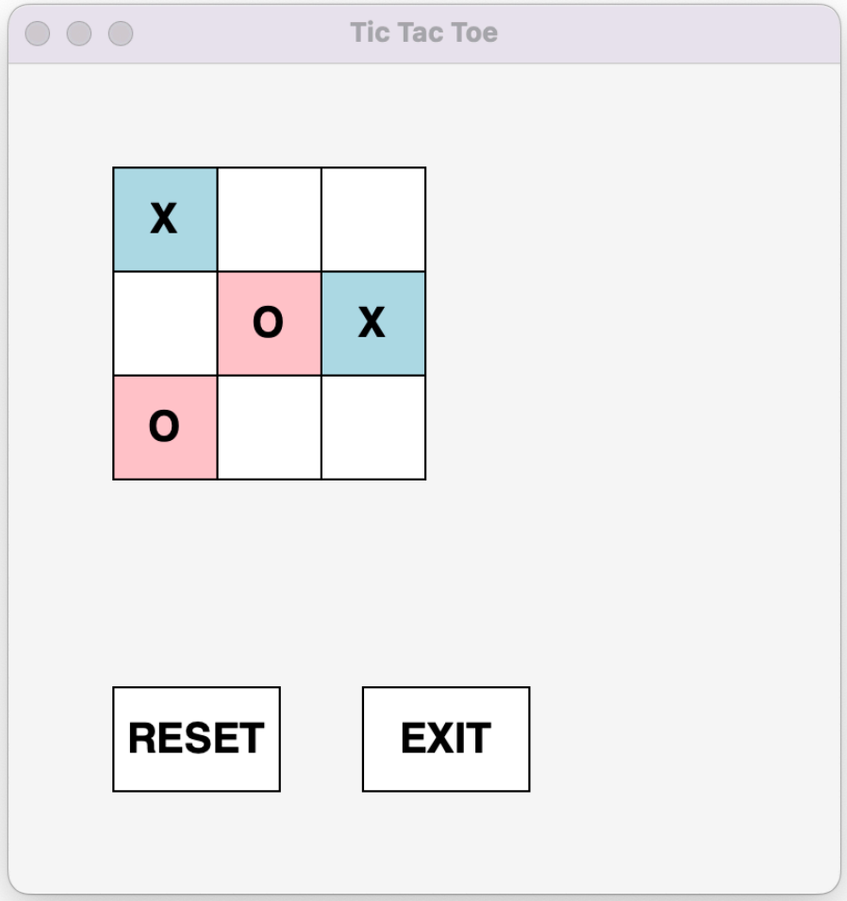
`TTTBoard`: Design

- `_cubes` mirrors the `Board` class's grid of `TextRect` objects
  - Initially populate board's `_grid` with `TTTCubes` that are "empty"
  - Then, update graphics objects stored in the `Board` class's _grid to reflect the state of the `TTTCubes`

```
def place_cubes_on_board(self):
        '''Updates the board to display the letters on TTTCubes'''
        for row in range(self._rows):
            for col in range(self._cols):
                cube = self._cubes[row][col]
                # method from board class to update the TextRect
                self.set_grid_cell(row, col, cube.get_letter()
```

`TTTBoard`: Design

- Let's think about the Board state in the "middle of the game"



- What are some helper methods that can help get/set the game state?
  - Check individual `TTTCubes` for X or O
- Setting individual `TTTCubes` to X or O
- Checking for wins (how?)
  - Need helper methods for row/column/diagonal checks

Accessing Letters on the Board

- Initially, our board is blank. To put some characters on the board, what do we need to do?
    - Change the `TTTCube` object from "" (empty) to "X" or "O"
    - Update the `Board`'s TextRect to include the new letter
- Let's write a getter method to help us get `TTTCube` objects from our grid

Works with screen "Points" from mouse clicks (such as (100, 200))

```
def get_ttt_cube_at_point(self, point):
        """Returns the TTTCube at point on window (a screen coord pair)"""
        if self.in_grid(point):
            # get_position returns grid coords as a (row,col) pair
            (row, col) = self.get_position(point)
            return self._cubes[row][col]
        return None
```

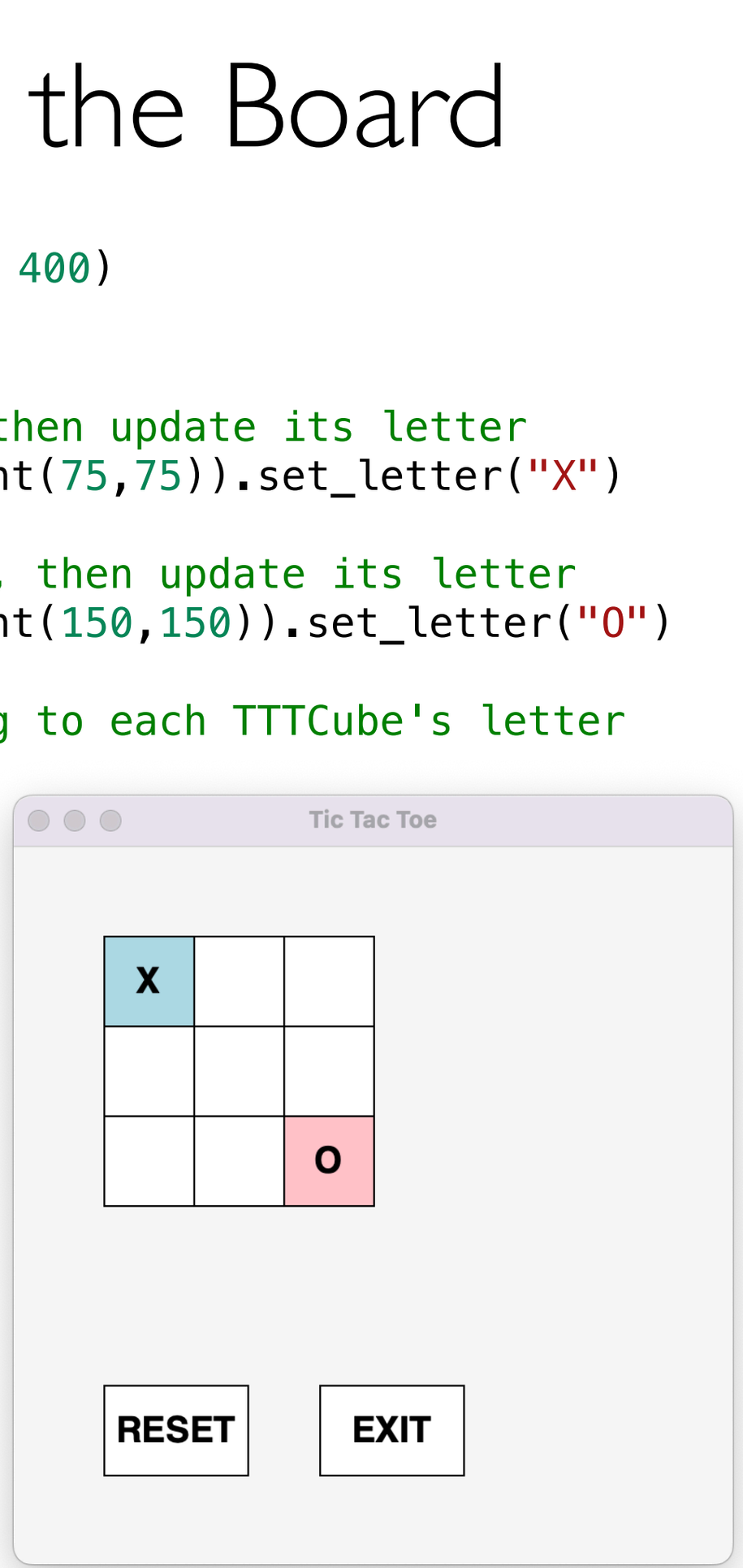
Setting Letters on the Board

- Once we have a **`TTTCube`** object, we can use the **`set_letter()`** method to change the character to an "X" or "O"

```
# get cube at pixel Point(75,75), then update its letter
tttboard.get_ttt_cube_at_point(Point(75,75)).set_letter("X")

# get cube at pixel Point(150,150), then update its letter
tttboard.get_ttt_cube_at_point(Point(150,150)).set_letter("O")
```

Setting Letters on the Board

```
win = GraphWin("Tic-Tac-Toe", 400, 400)

board = TTTBoard(win)
# get cube at pixel Point(75,75), then update its letter
tttboard.get_ttt_cube_at_point(Point(75,75)).set_letter("X")

# get cube at pixel Point(150,150), then update its letter
tttboard.get_ttt_cube_at_point(Point(150,150)).set_letter("O")

# update the TextRect corresponding to each TTTCube's letter
tttboard.place_cubes_on_board()
```

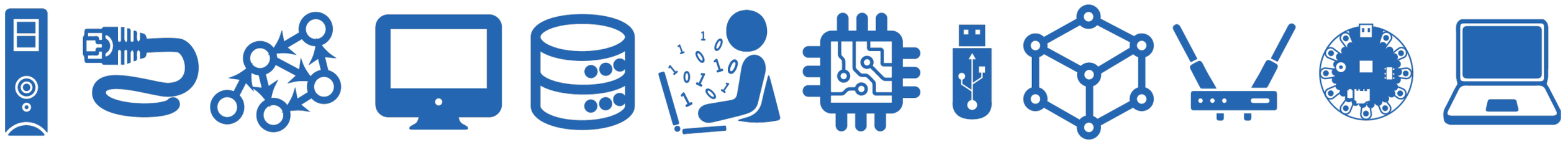


Resetting the TTTBoard

- As we are building the Board it would be helpful for us to have a way to reset the state of the board to be blank
- This, of course, is also helpful during play (if we hit the reset button or the game ends in Win/Draw and we want to restart)
- What do we need to change to reset the board?
  - Reset every **TTTCube** to an empty string

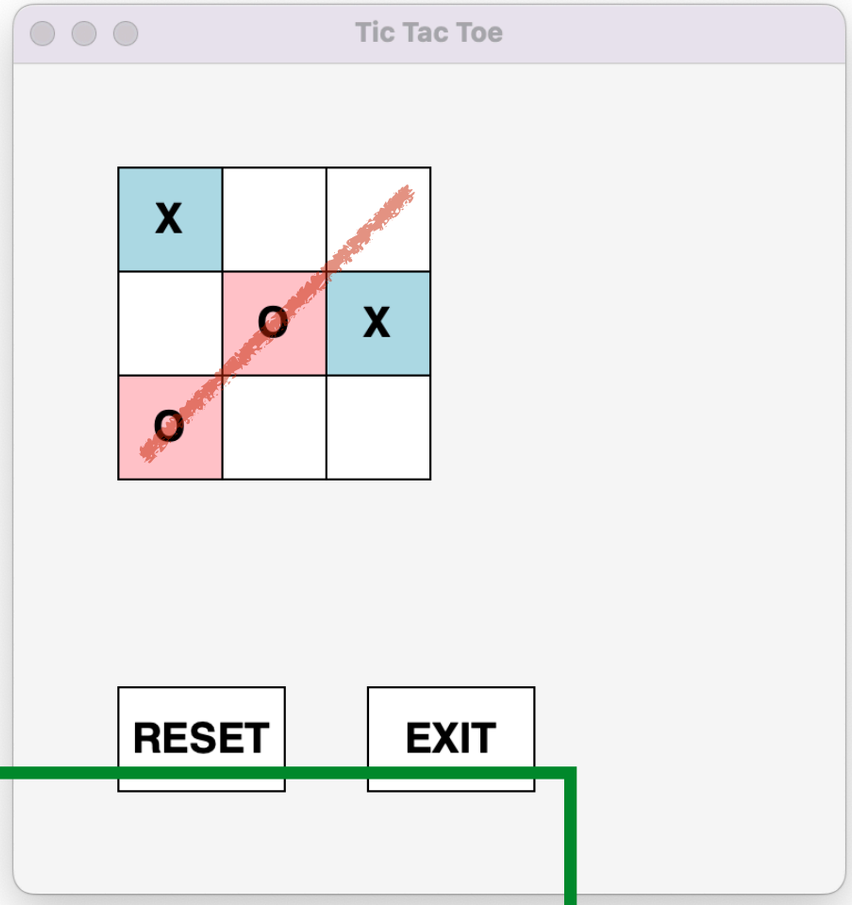
```
def reset(self):
        """Clears the TTT board by clearing letters and colors on grid"""

        # first update cube letters, then make the grid's graphics
        # reflect the state of the reset cubes
        for x in range(self._rows):
            for y in range(self._cols):
                self._cubes[x][y].set_letter("")
        self.place_cubes_on_board()
```

`TTTBoard` Helper Methods: Checking for Wins



`TTTBoard`: Design

- Let's think about the Board state in the "middle of the game"



- What are some helper methods that can help get/set the game state?
  - Check individual `TTTCubes` for X or O
- Setting individual `TTTCubes` to X or O
- Checking for wins (how?)
  - Need helper methods for row/column/diagonal checks

Getting Closer

- What other helper methods do we need?
    - Checking for wins for any player (either "X" or "O")
- A player ("X" or "O") wins if:
    - There exists a column filled with their letter, OR
    - There exists a row filled with their letter, OR
    - There exists a diagonal that is filled with their letter
- Let's break that down into separate private helper methods
    - `_check_rows`
    - `_check_cols`
    - `_check_diagonals`

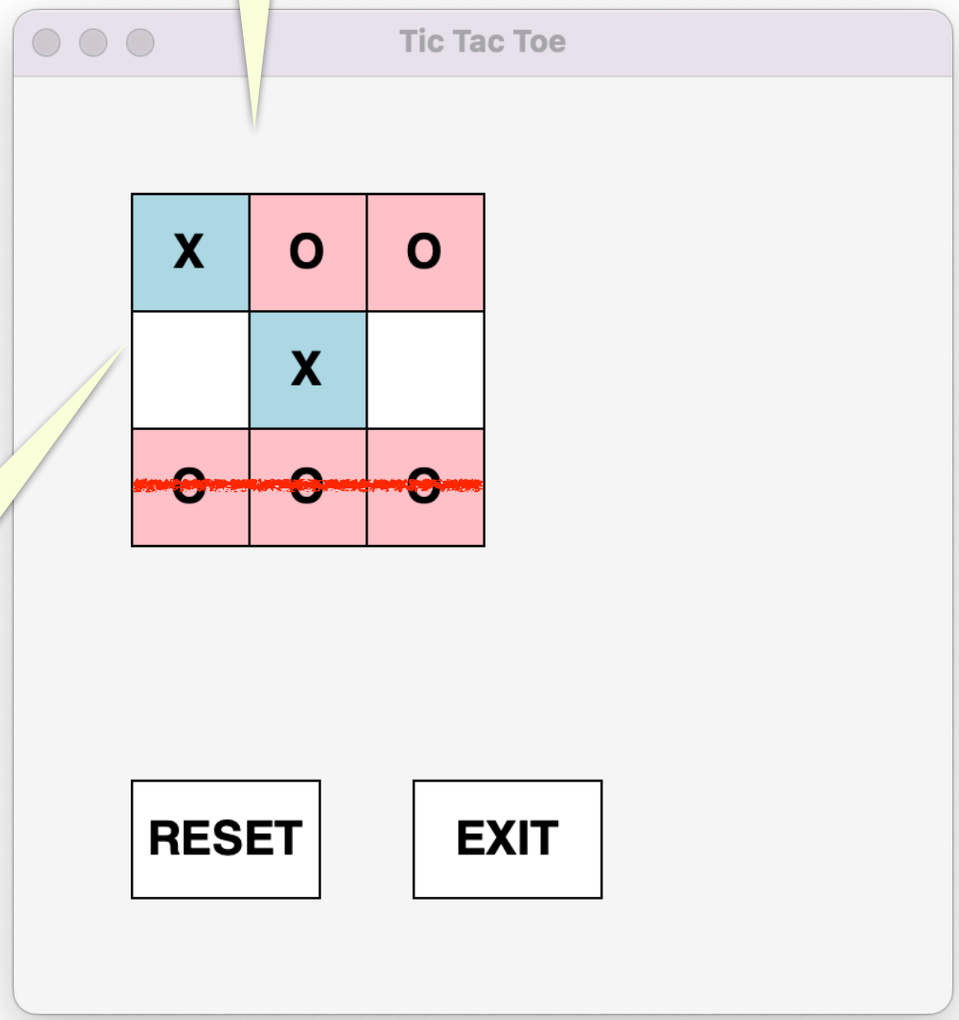
Checking the Rows

- For a given letter ("X" or "O"), we need to find if there is ANY row that is made of only **`letter`**
- How can we approach this?

```
def _checkRows(self, letter):
    pass
```

checkRows checks the board
horizontally

Grid positions are **`(col, row)`**



Checking the Rows

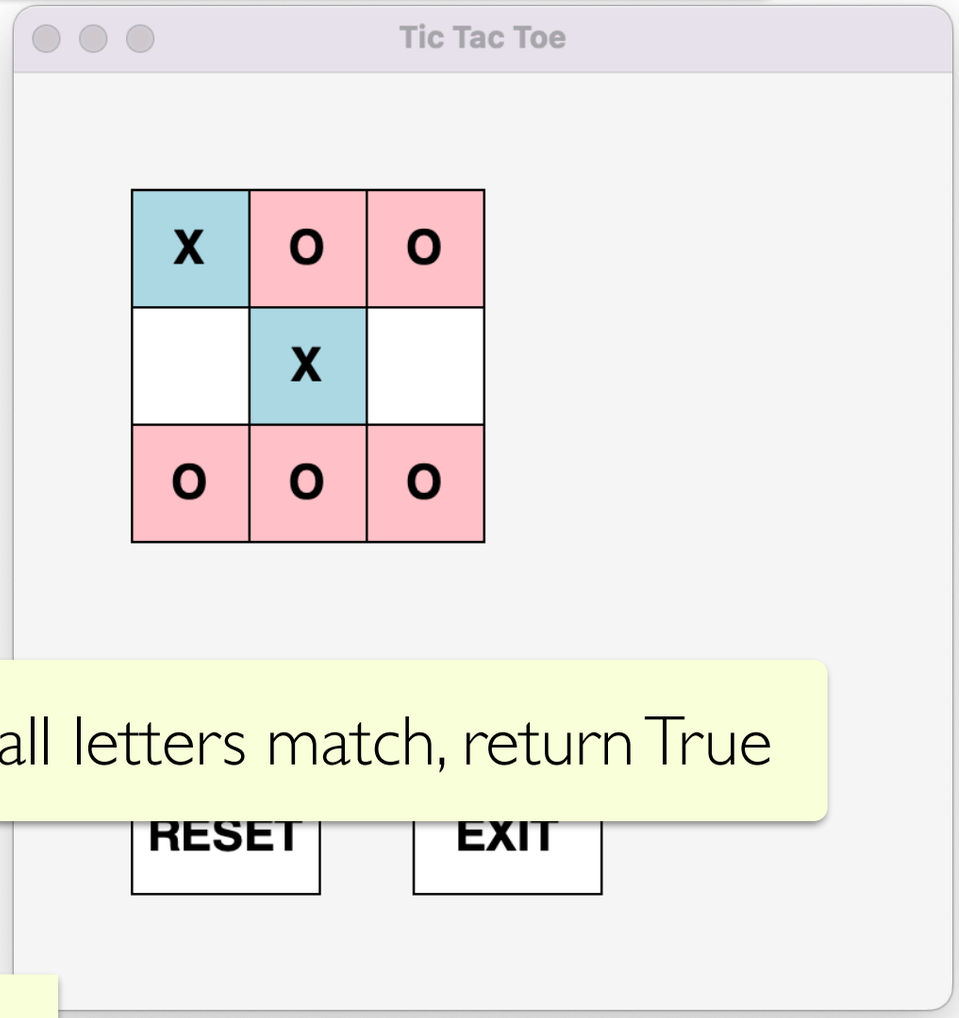
- For a given letter ("X" or "O"), we need to find if there is ANY row that is made of only **`letter`**
- Fix a row, go through each column

Why initialize **`count`** here?

```
def _check_rows(self, letter):
        """Check rows for a win (3 in a row)."""
        for row in range(self._rows):
            count = 0
            for col in range(self._cols):
                cube = self._cubes[row][col]

                # check how many times letter appears
                if cube.get_letter() == letter:
                    count += 1

            # if this is a winning row
            if count == self._cols:
                return True

        # no winning row found
        return False
```

If all letters match, return True

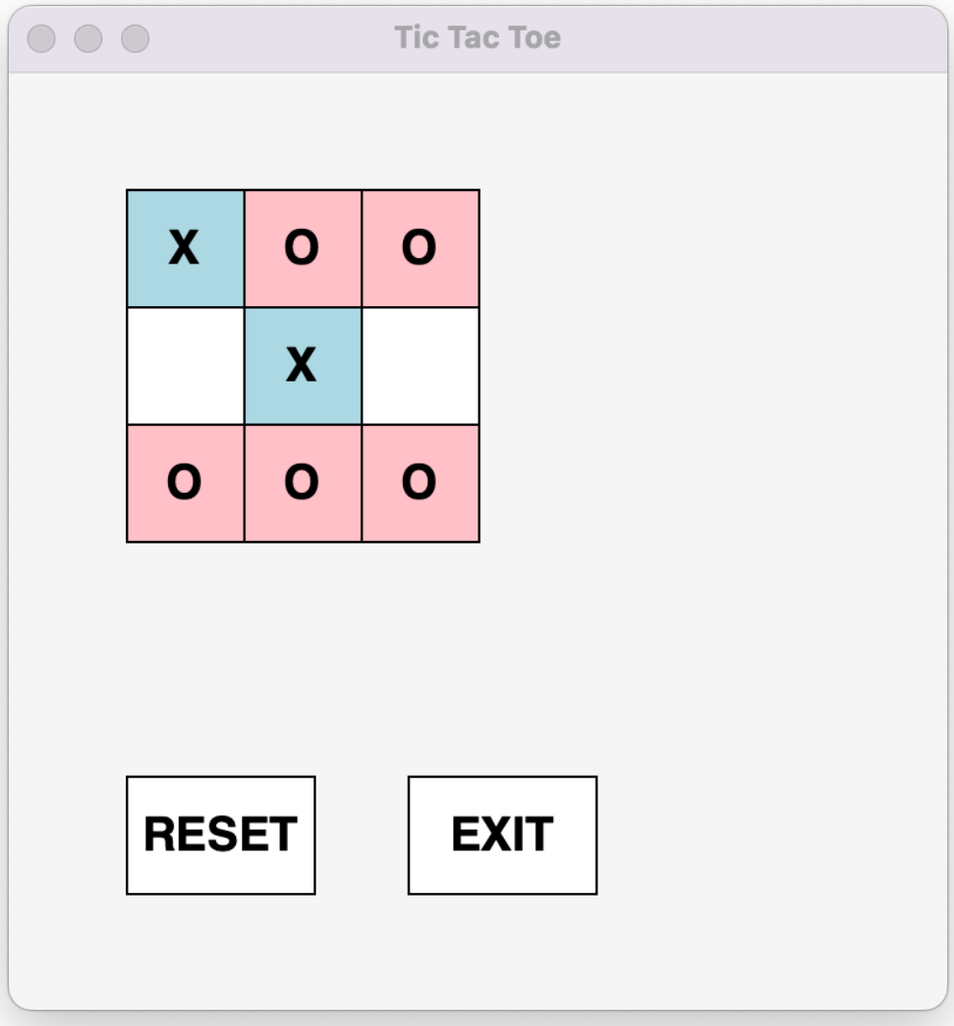
If no winning row, return False



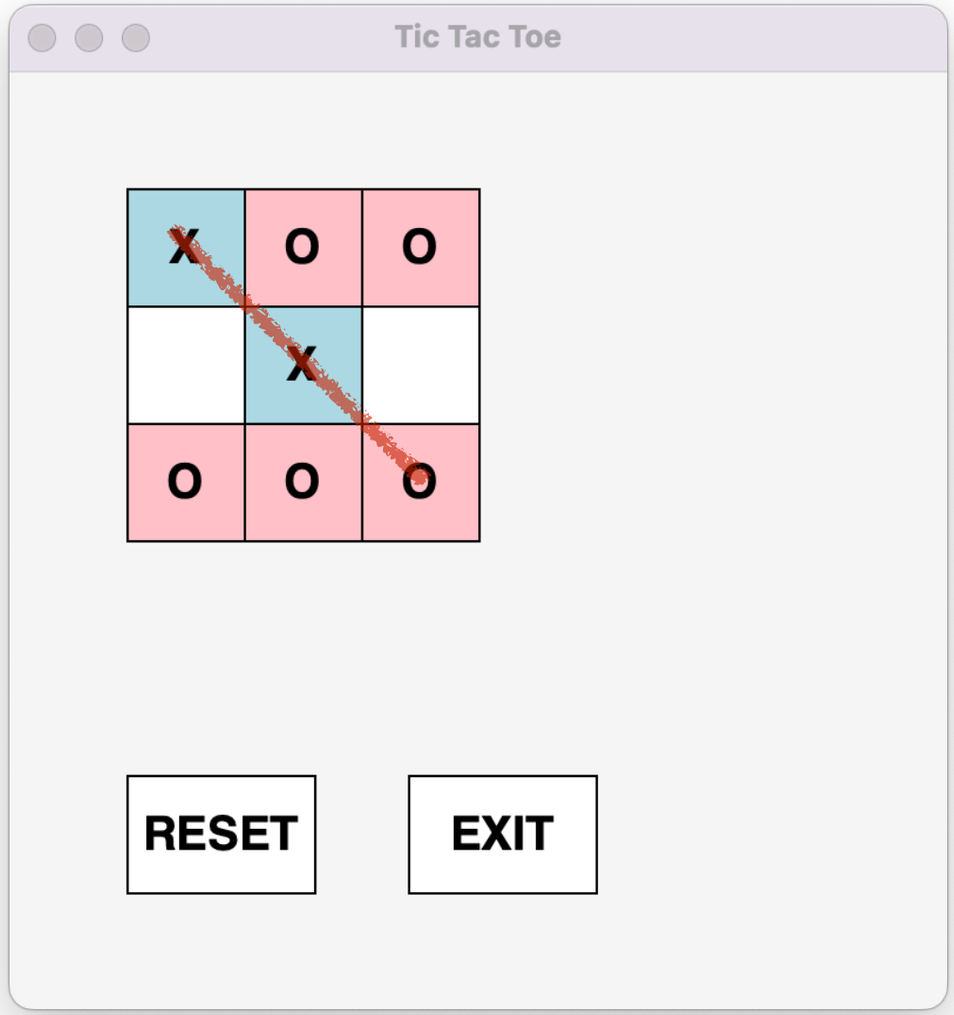
Similarly Check Columns

- We can similarly check a column for a win

```
def _check_cols(self, letter):
        """Check columns for a win (3 in a row)."""
        for col in range(self._cols):
            count = 0
            for row in range(self._rows):
                cube = self._cubes[row][col]

                # check how many times letter appears
                if cube.get_letter() == letter:
                    count +=1

            # if this is a winning col
            if count == self._rows:
                return True

        # if no winning cols
        return False
```



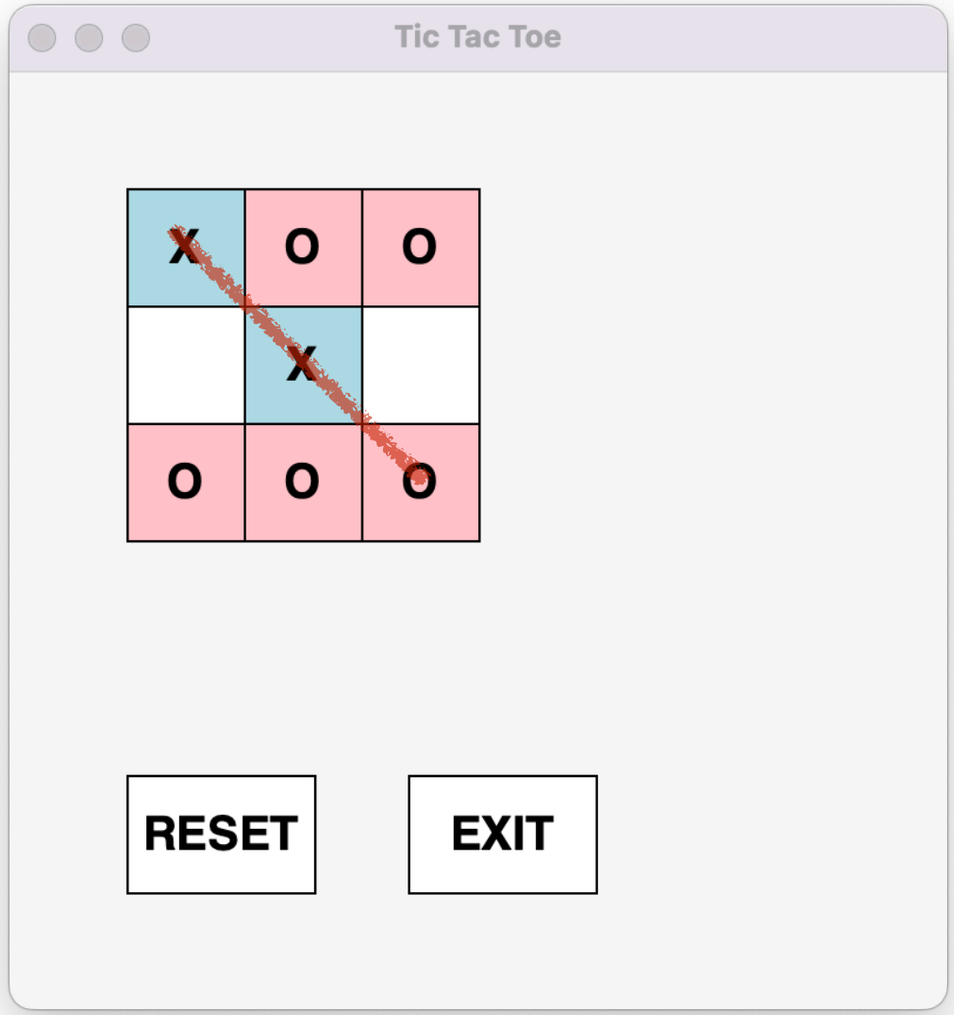
Check Diagonals




```
def _check_diagonals(self, letter):
        """Check diagonals for a win (3 in a row)."""
        # counts for primary and secondary diagonal
        count_primary = 0
        count_second = 0

        for col in range(self._cols):
            for row in range(self._rows):
                cube = self._cubes[col][row]

                # update count for primary diagonal
                if (_____________________________________________):
                    count_primary += 1

                # update count for secondary diagonal
                if (_____________________________________________):
                    count_second += 1

        # return true if either return in win
        return ___________________________________
```

Primary diagonal has row/col same

Check Diagonals

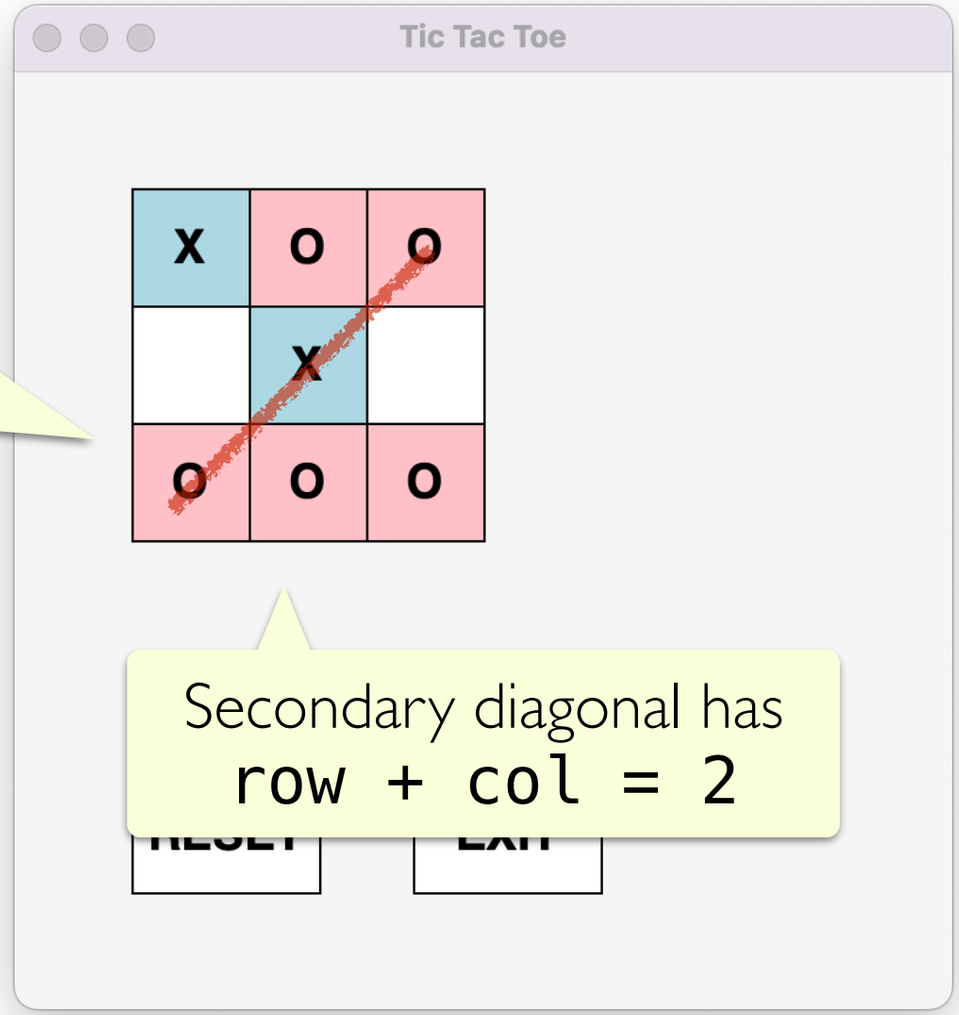



```
def _check_diagonals(self, letter):
        """Check diagonals for a win (3 in a row)."""
        # counts for primary and secondary diagonal
        count_primary = 0
        count_second = 0

        for col in range(self._cols):
            for row in range(self._rows):
                cube = self._cubes[col][row]

                # update count for primary diagonal
                if (row == col and cube.get_letter() == letter):
                    count_primary += 1

                # update count for secondary diagonal
                if (_____________________________________________________):
                    count_second += 1

        # return true if either return in win
        return ___________________________________
```

Primary diagonal has row/col same

Check Diagonals

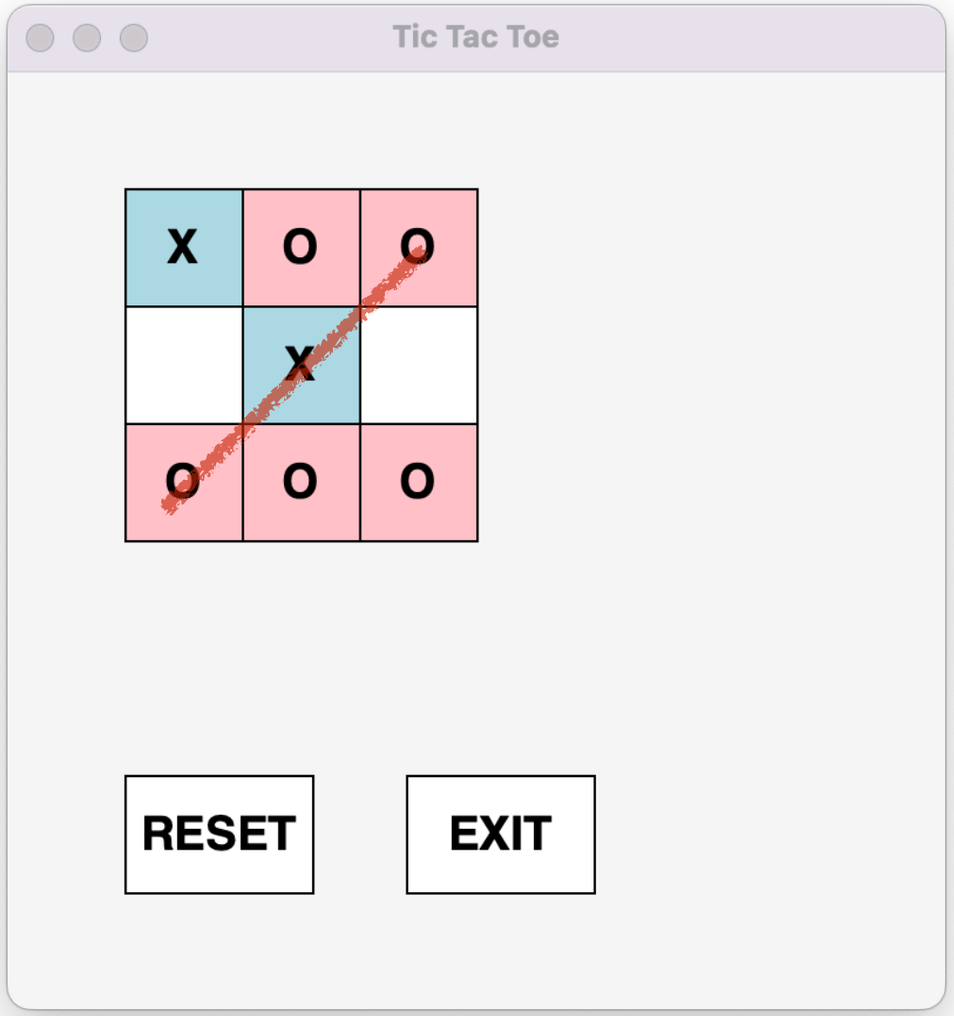
Secondary diagonal:
(0, 2), (1,1), (2, 0) for a 3x3 board




```
def _check_diagonals(self, letter):
        """Check diagonals for a win (3 in a row)."""
        # counts for primary and secondary diagonal
        count_primary = 0
        count_second = 0

        for col in range(self._cols):
            for row in range(self._rows):
                cube = self._cubes[col][row]

                # update count for primary diagonal
                if (row == col and cube.get_letter() == letter):
                    count_primary += 1

                # update count for secondary diagonal
                if (_____________________________________________):
                    count_second += 1

        # return true if either return in win
        return ____________________________________
```

What condition defines secondary diagonal?

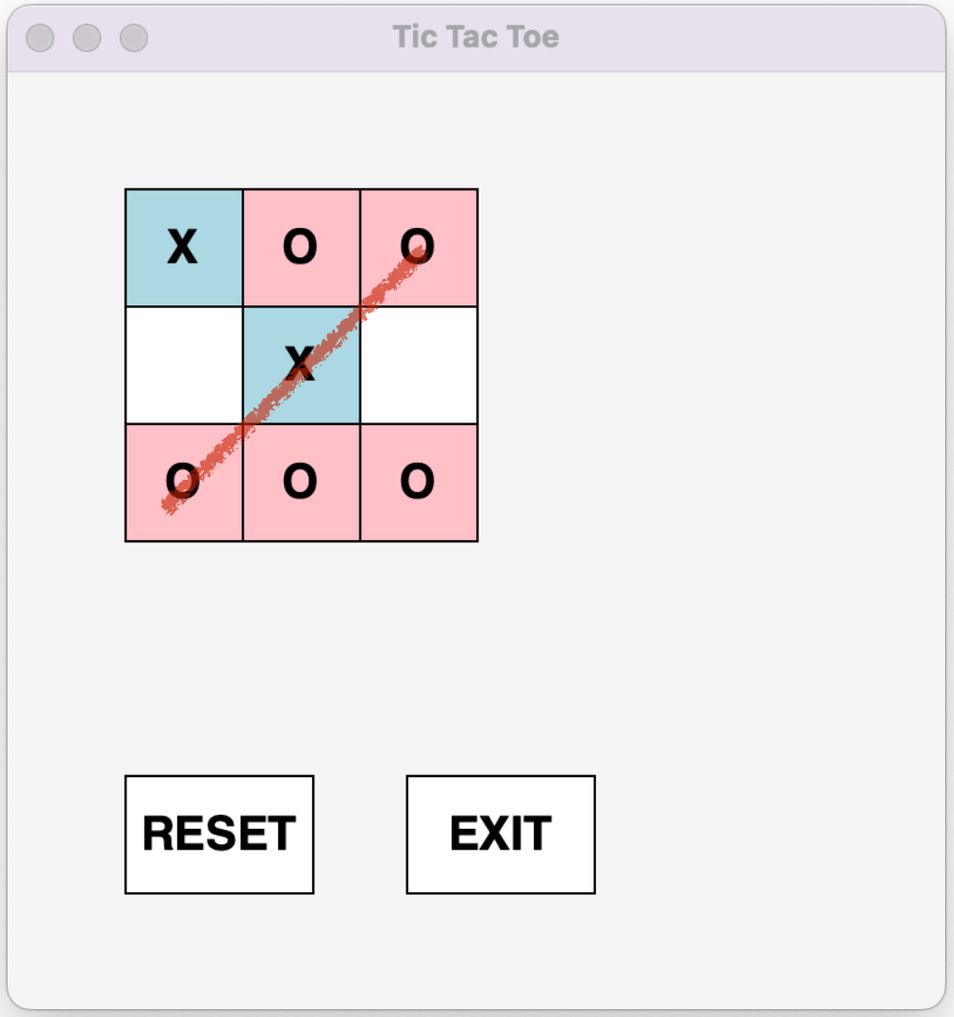
Check Diagonals




```
def _check_diagonals(self, letter):
        """Check diagonals for a win (3 in a row)."""
        # counts for primary and secondary diagonal
        count_primary = 0
        count_second = 0

        for col in range(self._cols):
            for row in range(self._rows):
                cube = self._cubes[col][row]

                # update count for primary diagonal
                if (row == col and cube.get_letter() == letter):
                    count_primary += 1

                # update count for secondary diagonal
                if (row + col == self._rows - 1 and cube.get_letter() == letter):
                    count_second += 1

        # return true if either return in win
        return ____________________________________
```

What condition defines secondary diagonal?

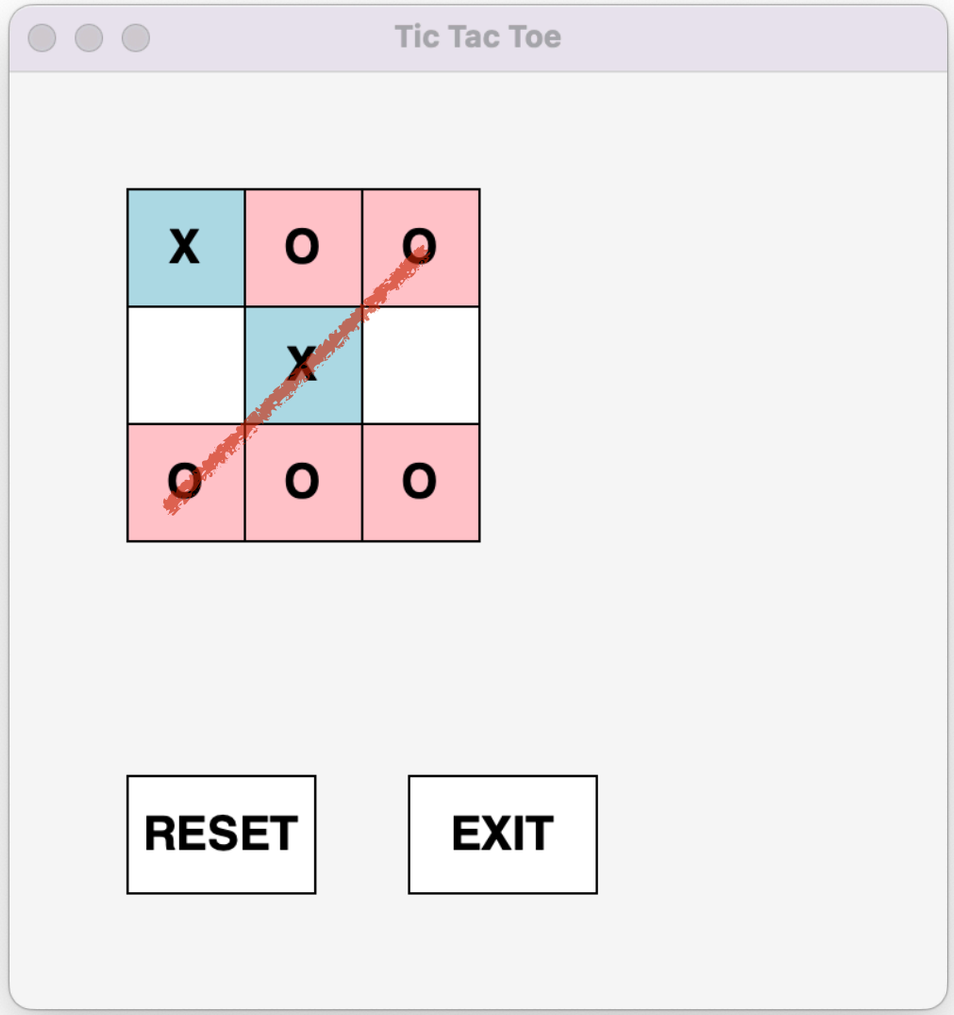
Check Diagonals




```
def _check_diagonals(self, letter):
        """Check diagonals for a win (3 in a row)."""
        # counts for primary and secondary diagonal
        count_primary = 0
        count_second = 0

        for col in range(self._cols):
            for row in range(self._rows):
                cube = self._cubes[col][row]

                # update count for primary diagonal
                if (row == col and cube.get_letter() == letter):
                    count_primary += 1

                # update count for secondary diagonal
                if (row + col == self._rows - 1 and cube.get_letter(  == letter):
                    count_second += 1

        # return true if either return in win
        return ____________________________________
```

Condition that is true on win, false otherwise?

Check Diagonals



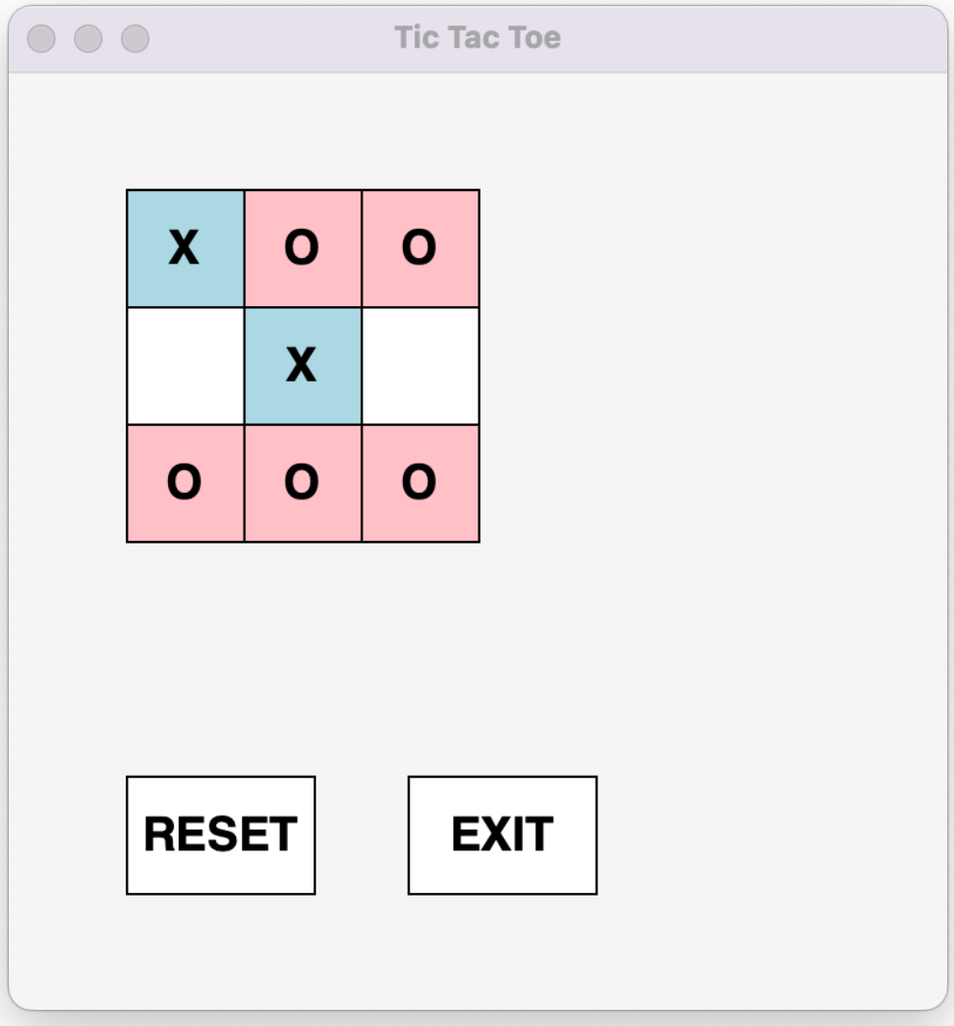

```
def _check_diagonals(self, letter):
        """Check diagonals for a win (3 in a row)."""
        # counts for primary and secondary diagonal
        count_primary = 0
        count_second = 0

        for col in range(self._cols):
            for row in range(self._rows):
                cube = self._cubes[col][row]

                # update count for primary diagonal
                if (row == col and cube.get_letter() == letter):
                    count_primary += 1

                # update count for secondary diagonal
                if (row + col == self._rows - 1 and cube.get_letter(  == letter):
                    count_second += 1

        # return true if either return in win
        return count_primary == self.get_rows() or count_second == self.get_rows()
```

Condition that is true on win, false otherwise?

Final Check for Win

- Putting it all together: the board is in a winning state if any of the three winning conditions are true
- We will make this method public as it will needed outside of this class

```
def check_for_win(self, letter):
        """Check board for a win."""
        row_win = self._check_rows(letter)
        col_win = self._check_cols(letter)
        diag_win = self._check_diagonals(letter)

        return row_win or col_win or diag_win
```



Leftovers: Next time

- We don't have a working Tic Tac Toe game yet
    - But we're getting close!
- What's left?
    - We have been using TTTCube, so we'll look at it briefly
    - We need to implement the **game logic**
- What do we need to do to put this all together?
    - Keep track of mouse clicks
    - Keep track of players ("X" and "O" must alternate)
    - Use TTTCube and TTTBoard to check for win/tie

The end!