CS134 Lecture 25: Inheritance and Board Class

Announcements & Logistics

- **HW 8** will be released today (due Mon 10 pm)
- **Lab 6** graded feedback returned
- **Lab 8** due tonight 10 pm (~Mon lab), Thurs 10 pm (~Tues lab)
- Lab 9 (two week) lab: strongly encourage you work in pairs
    - "Mini project" : different from standard labs in length/complexity
    - Fill out Lida's partner form by noon tomorrow
- TA applications due Friday:
    - https://csci.williams.edu/tatutor-application/
- Please give feedback on CS134 TAs by Friday:
    - https://forms.gle/nZSPcwbaP3WCWxqEA

Do You Have Any Questions?

Last Time

- Designed a Library class that stores a sorted shelf of Book objects
- Learnt how to:
    - call `sorted()` function in Python by specifying the `key` function
    - how to *pass a function* as an ***argument*** to another function
    - define/call functions with optional arguments
- Reviewed some useful (built-in) string and list methods:
    - `s (str): s.strip(), s.split(), s.join(), s.format()`
    - `l (list): l.append(), l.remove()`

Today’s Plan

- Continue discussing some of the important OOP principles
  - **Abstraction** - handle complexity by ignoring/hiding messy details
  - **Inheritance** - derive a class from another class that shares a set of attributes and methods
  - **Encapsulation** bundling data & methods that work together in a class
  - **Polymorphism** - using a single method or operator for different uses
- Focus on inheritance
- Start implementing a text-based board game

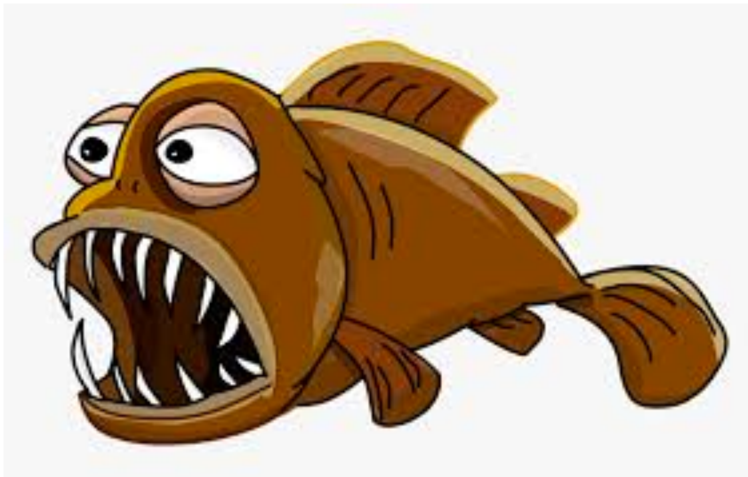
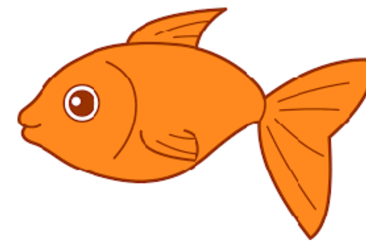
Inheritance

Introduction to Inheritance

- **Inheritance** is the capability of one class to derive or ***inherit*** the properties from another class
- Benefits of inheritance:
  - Often represents real-world relationships well
  - **Code reuse:** avoid writing the same code again and again
  - Allows us to add more features to a class without modifying it
- Inheritance is **transitive** in nature: if class B inherits from class A, then all the subclasses of B would also automatically inherit from class A
- When a class inherits from another class, all methods and attributes are accessible to subclass, ***except private attributes*** (indicated with **__**)

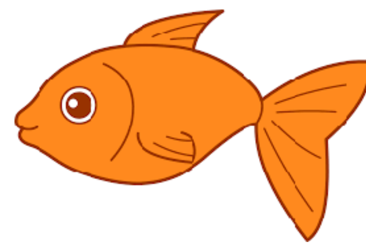
Inheritance Toy Example

- Suppose we have a base (or parent) class `Fish`
- `Fish` defines several methods that are common to all fish:
  - `eat()`, `swim()`
- `Fish` also defines several data attributes with default values:
  - `_length`, `_weight`, `_lifespan`

Inheritance Toy Example

- All fish have some features in common
  - But not all fish are the same!
- Each `Fish` instance will specify different values for attributes (`_length`, `_weight`, `_lifespan`)
- Some fish may still need extra functionality!



Inheritance Toy Example

- For example, Sharks might need an `attack()` method
- Pufferfish might need a `puff()` method
- We might even want to ***override*** an existing method with a different (more specialized) implementation
  - Inheritance allows for all of this!

Inheritance: Constructor

```
class Rectangle:

    def __init__(self, length, width):

        self._length = length
        self._width = width


class Square(Rectangle):

    def __init__(self, length):

        super().__init__(length, length)
```

Parent (super class)

Calls **constructor** of super class

Inheritance represents **"is a"** relationship.
A `Square` is a `Rectangle`.

Inheritance: Methods

```
class Rectangle:

    def __init__(self, length, width):
        self._length = length
        self._width = width

    def draw(self):
        print('draws a rectangle')

class Square(Rectangle):

    def __init__(self, length):
        super().__init__(length, length)

    def draw(self):
        print('draws a square')
```

```
sq = Square(12)

sq.draw()
```

calls `draw` of square

```
"draws a square"
```

Inheritance: Methods

```
class Rectangle:

    def __init__(self, length, width):
        self._length = length
        self._width = width

    def draw(self):
        print('draws a rectangle')

class Square(Rectangle):

    def __init__(self, length):
        super().__init__(length, length)

    def draw(self):
        print('draws a square')
```

```
sq = Square(12)

sq.draw()

"draws a square"
```

calls `draw` of square

draw method of `Square` overrides that of `Rectangle`

Inheritance: Methods

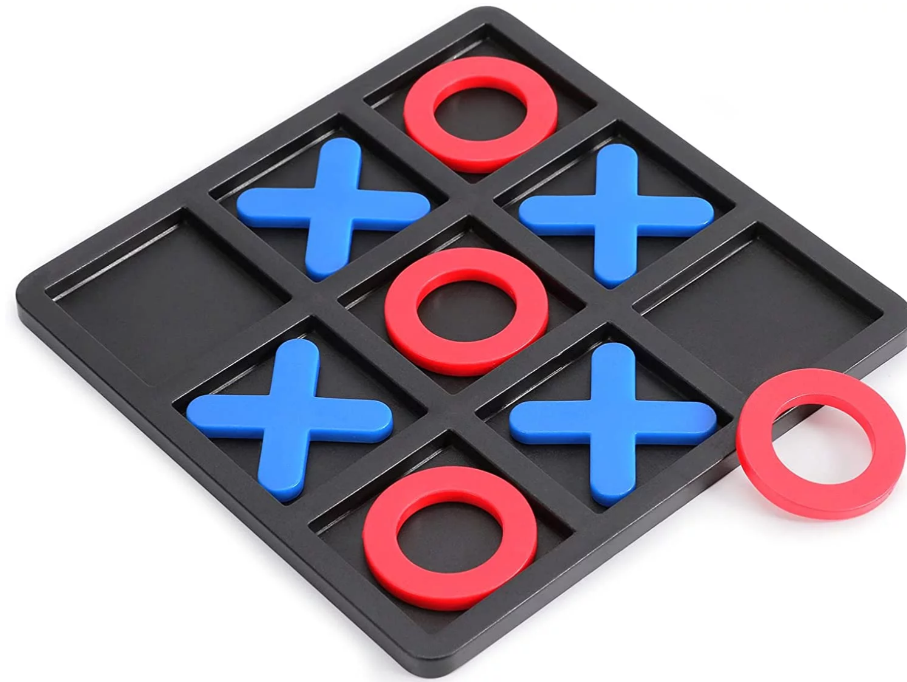
```
class Rectangle:

    def __init__(self, length, width):
        self._length = length
        self._width = width

    def draw(self):
        print('draws a rectangle')

class Square(Rectangle):

    def __init__(self, length):
        super().__init__(length, length)

    def draw(self):
        print('draws a square')
```

```
sq = Square(12)

sq.draw()

"draws a rectangle"
```

If `Square` has no `draw` method, it calls draw of `super` class

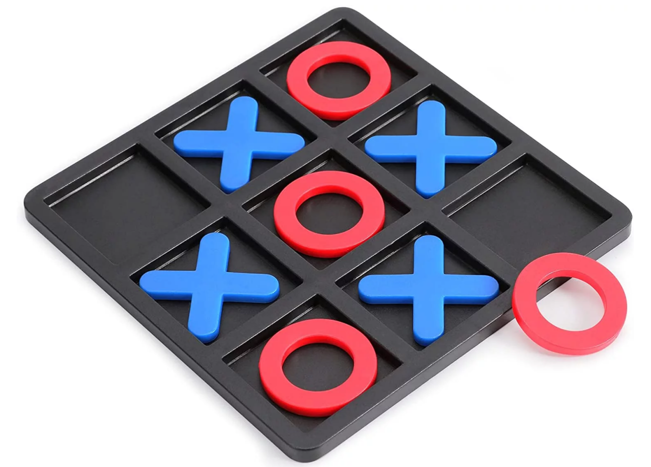
Inheritance and OOP: word-based board games

Simple Board Games



Common Features of Physical Game?

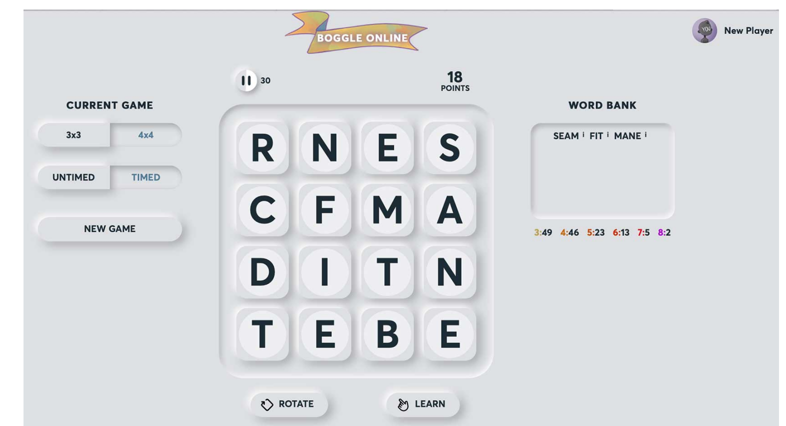
- Often 2 or many player
- Board at the bottom
    - Grid-based (rows and columns)
- Game pieces (tiles/cubes)
    - Go "on top" of the board
    - Have a letter (or many letters) on them
- Some uncertainty is part of the fun
    - Randomness in the configurations
- May or may not be timed

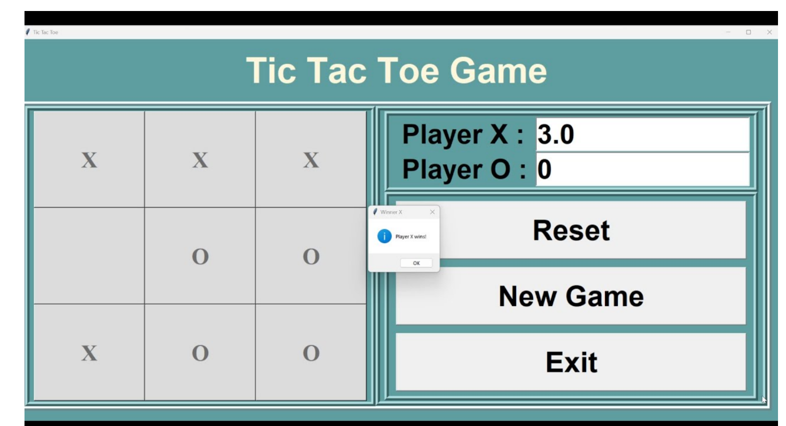


Computer Variants

Common Features of Computer Variants?

- Often 1 player (or play with computer)
- Game board: now a graphical screen
  - A grid area to place the pieces
  - Text areas on the sides to give game status
  - "Buttons" to reset/exit game
- Some uncertainty is part of the fun
  - Randomness in the configurations
- May or may not be timed

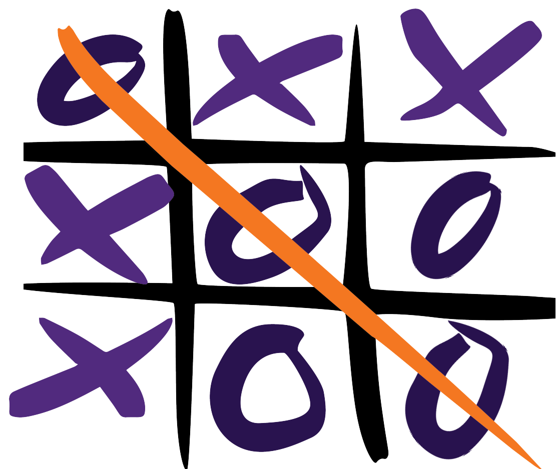


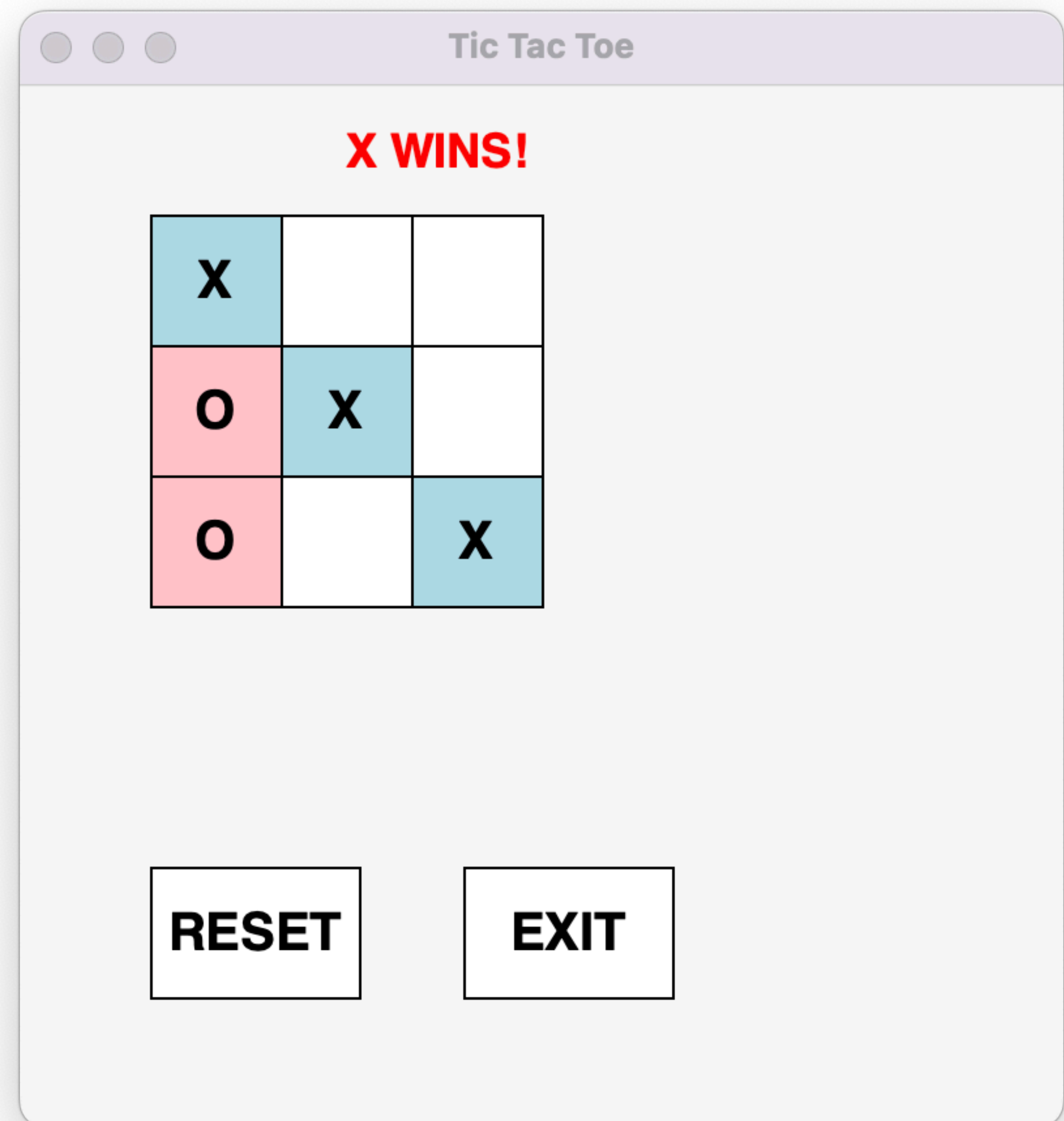


Example: Tic Tac Toe

- Suppose we want to implement Tic Tac Toe
- Teaser demo…

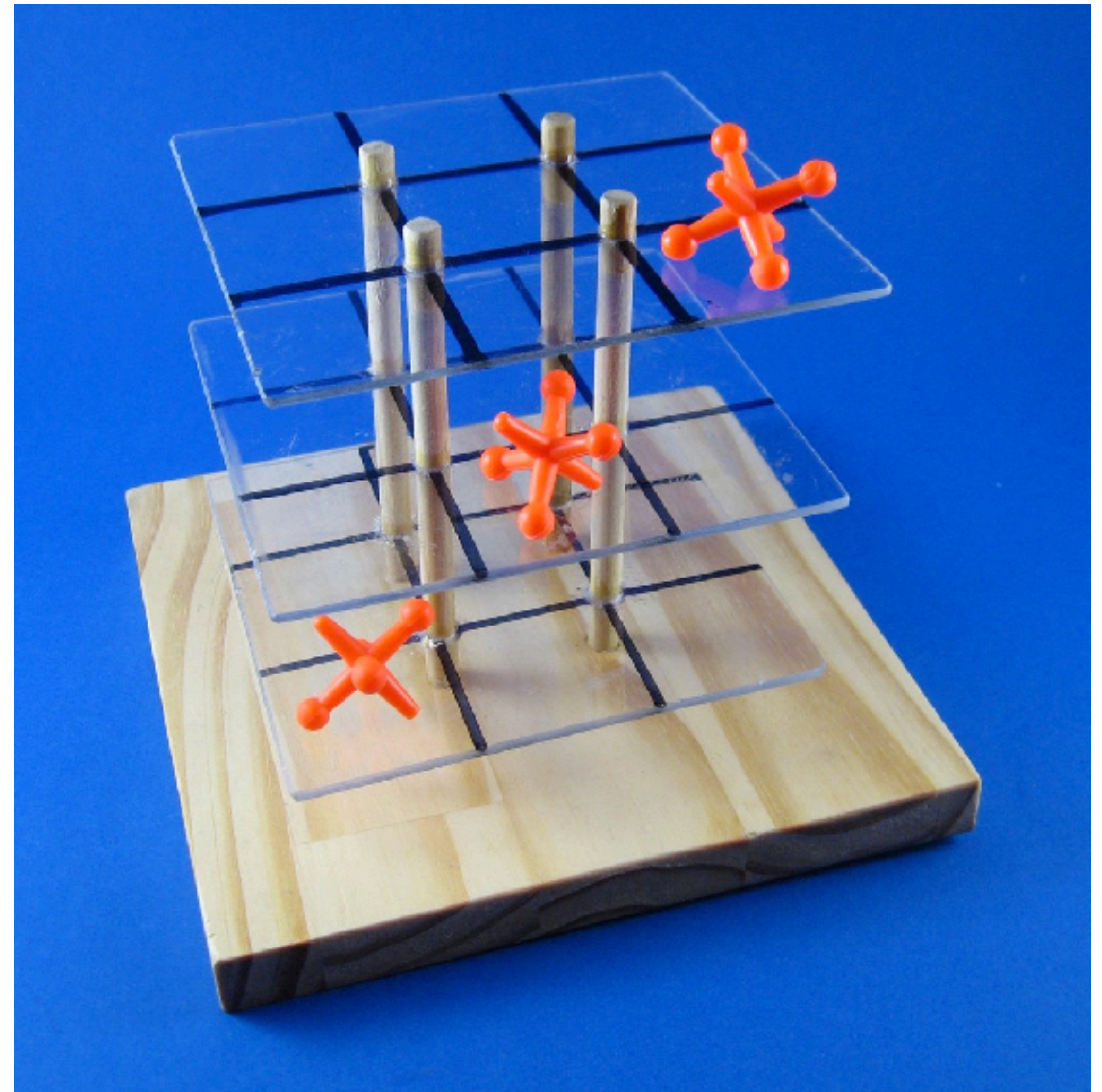
```
>>> python3 tttgame.py
```



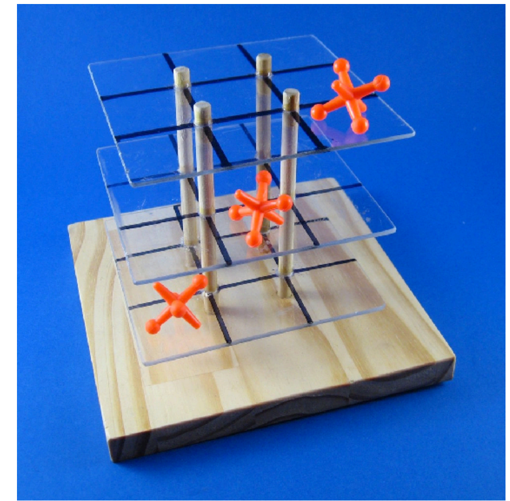


Decomposition

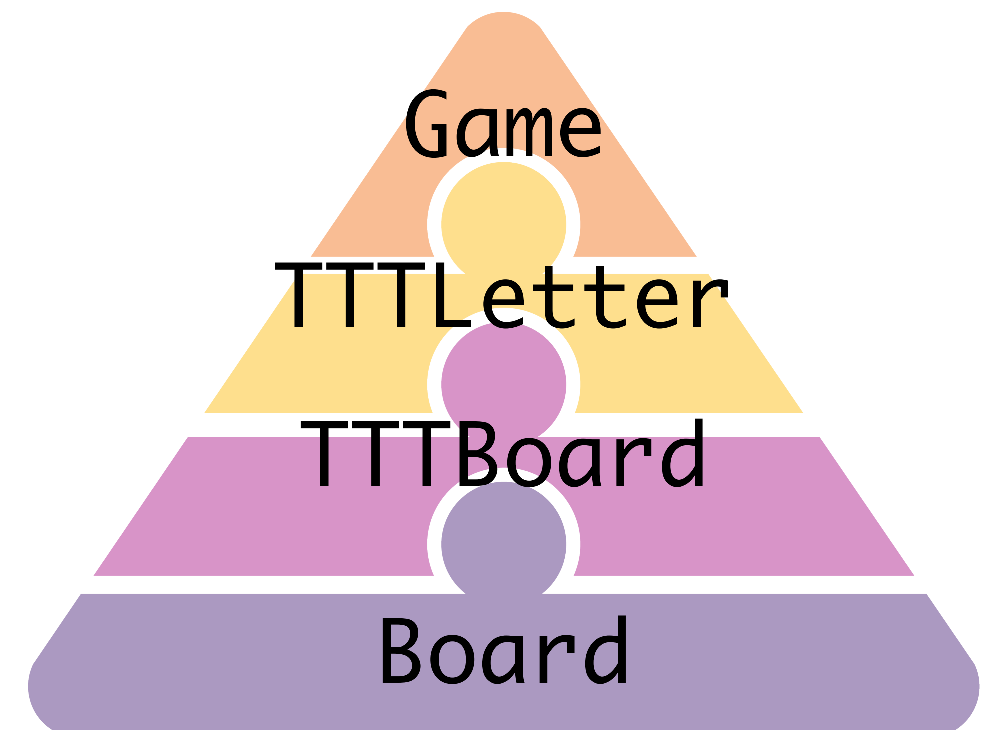
- Let's try to identify the "layers" of this game
- Through abstraction and encapsulation, each layer can ignore what's happening in the other layers
- What are the layers of Tic Tac Toe?



Decomposition

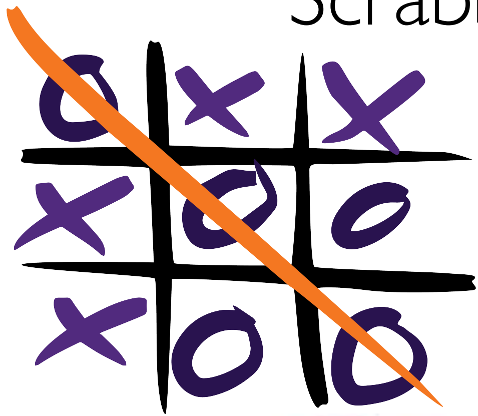


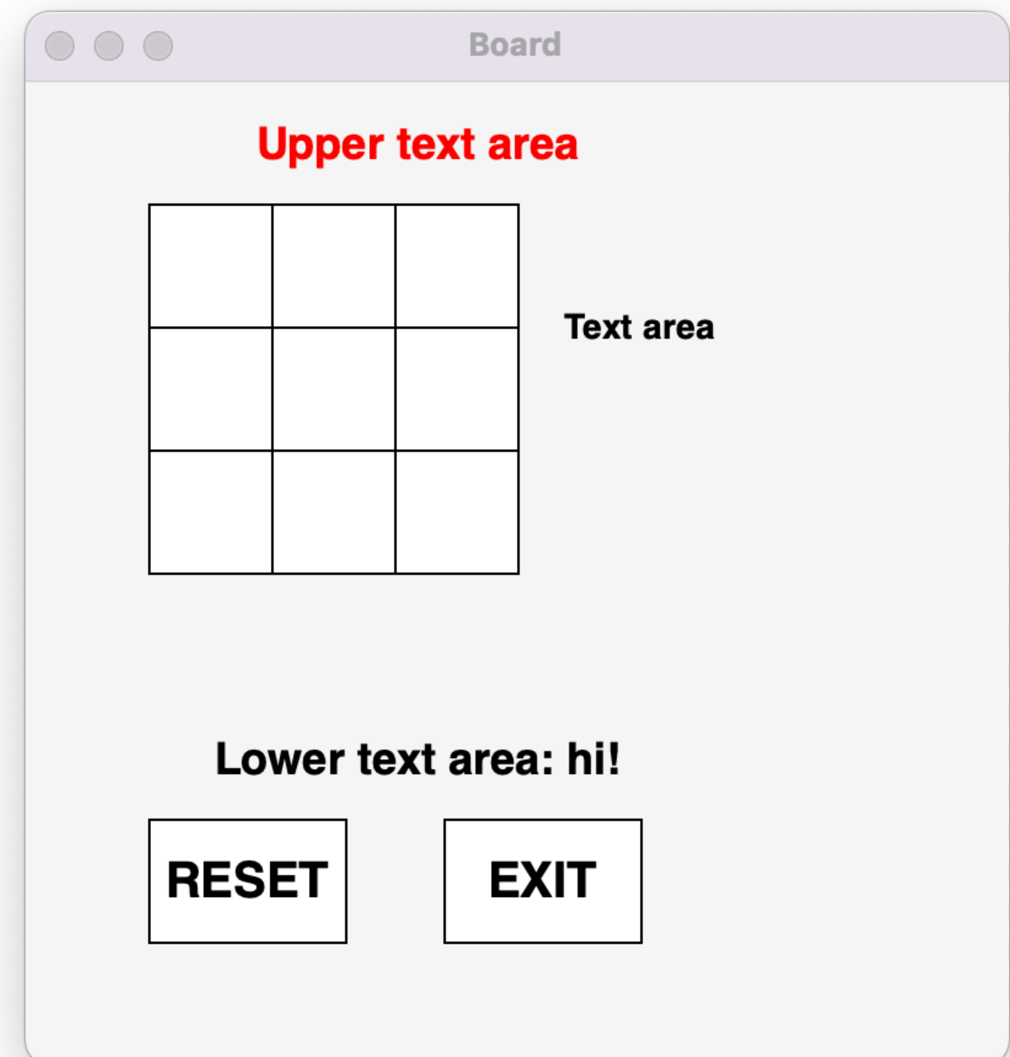
- Bottom layer: **Basic board** w/buttons, text areas, mouse click detection (not specific to Tic Tac Toe!)
- Lower middle layer: Extend the **basic board with Tic Tac Toe specific features** (3x3 grid, of TTTLetters, initial board state: all letters start blank)
- Upper middle layer: **Tic Tac Toe "cubes" or "letters"** (9 in total!); set text to X or O
- Top layer: **Game logic** (alternating turns, checking for valid moves, etc)



Board class

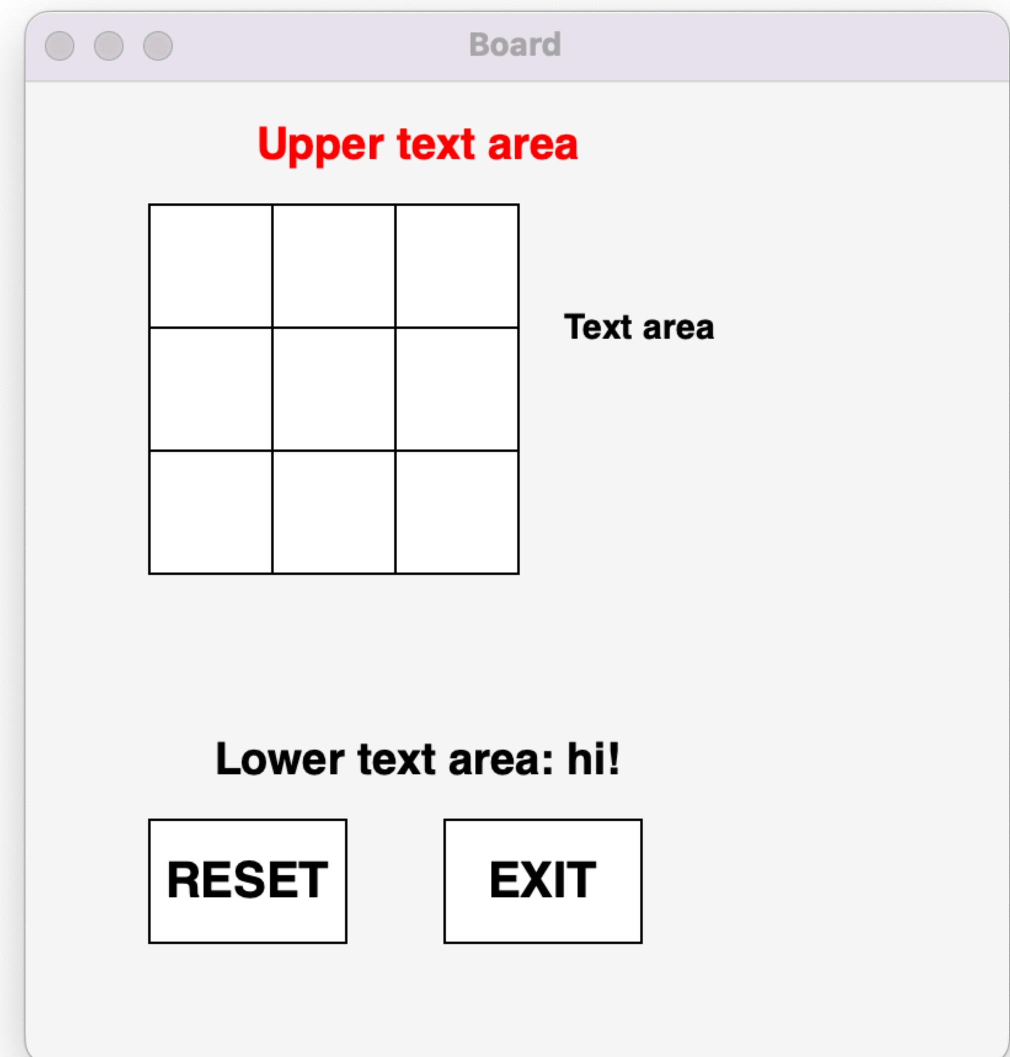
- Let's start at the bottom: Board class
- What are basic features of all game boards?
  - Think generally…many board-based games have the similar basic requirements
  - (For example, Boggle, TicTacToe, Scrabble, etc)



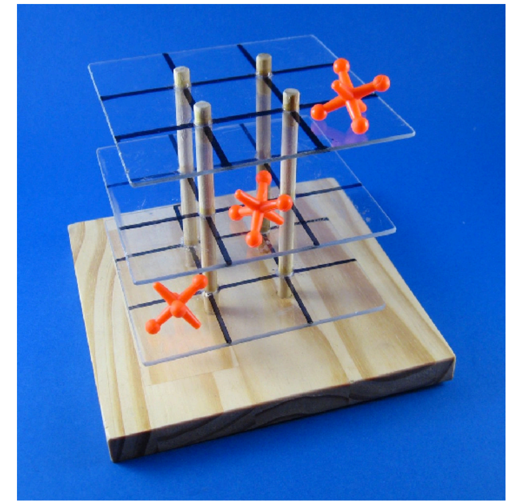


Board class

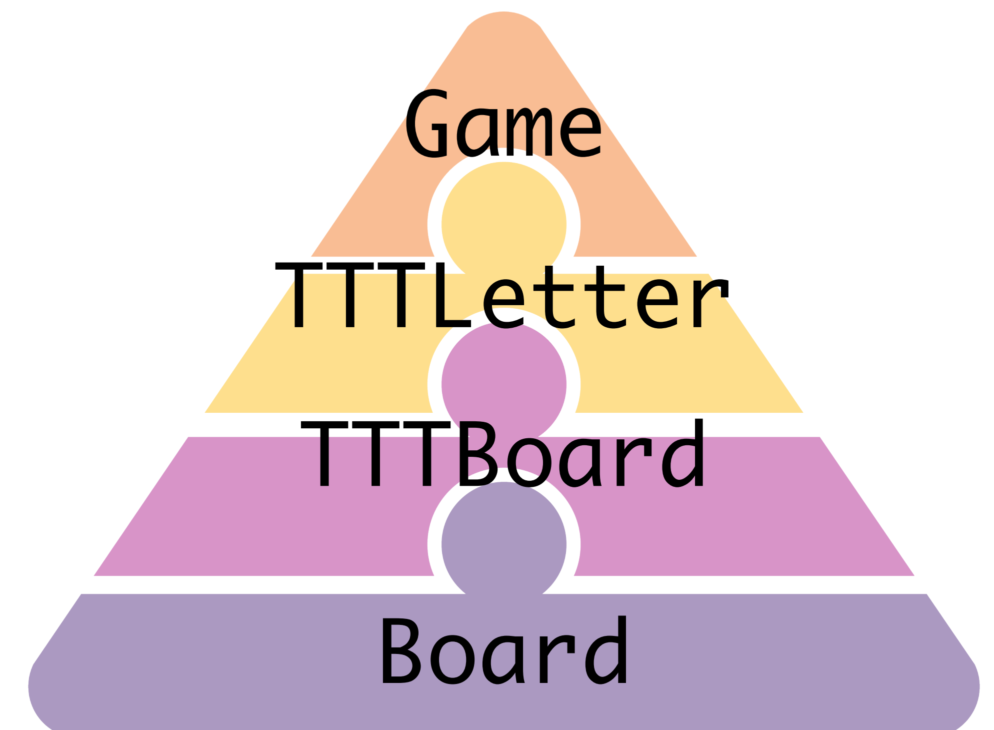
- Let's start at the bottom: Board class
- What are basic features of all game boards?
  - Text areas: above, below, right of grid
  - Grid of squares of set size: rows x cols
  - Reset and Exit buttons
  - React to mouse clicks (less obvious!)
- These are all **graphical** (GUI) components
  - Code for graphics is a little messy at times
  - Lot's of things to specify: color, size, location on screen, etc



Inheritance



- Board Class: (super class)
  - **Basic board** w/buttons, text areas, mouse click detection
- Tic Tac Toe (sub class)
  - Inherits from Board and extends it to TTT specific features and methods
  - Doesn't have to recreate a Board
- Looking ahead: Boggle (Lab 9)
  - Similar grid-based board game, also inherits from Board and extends it to Boggle features and methods



Graphics Module

Graphics Package for Board

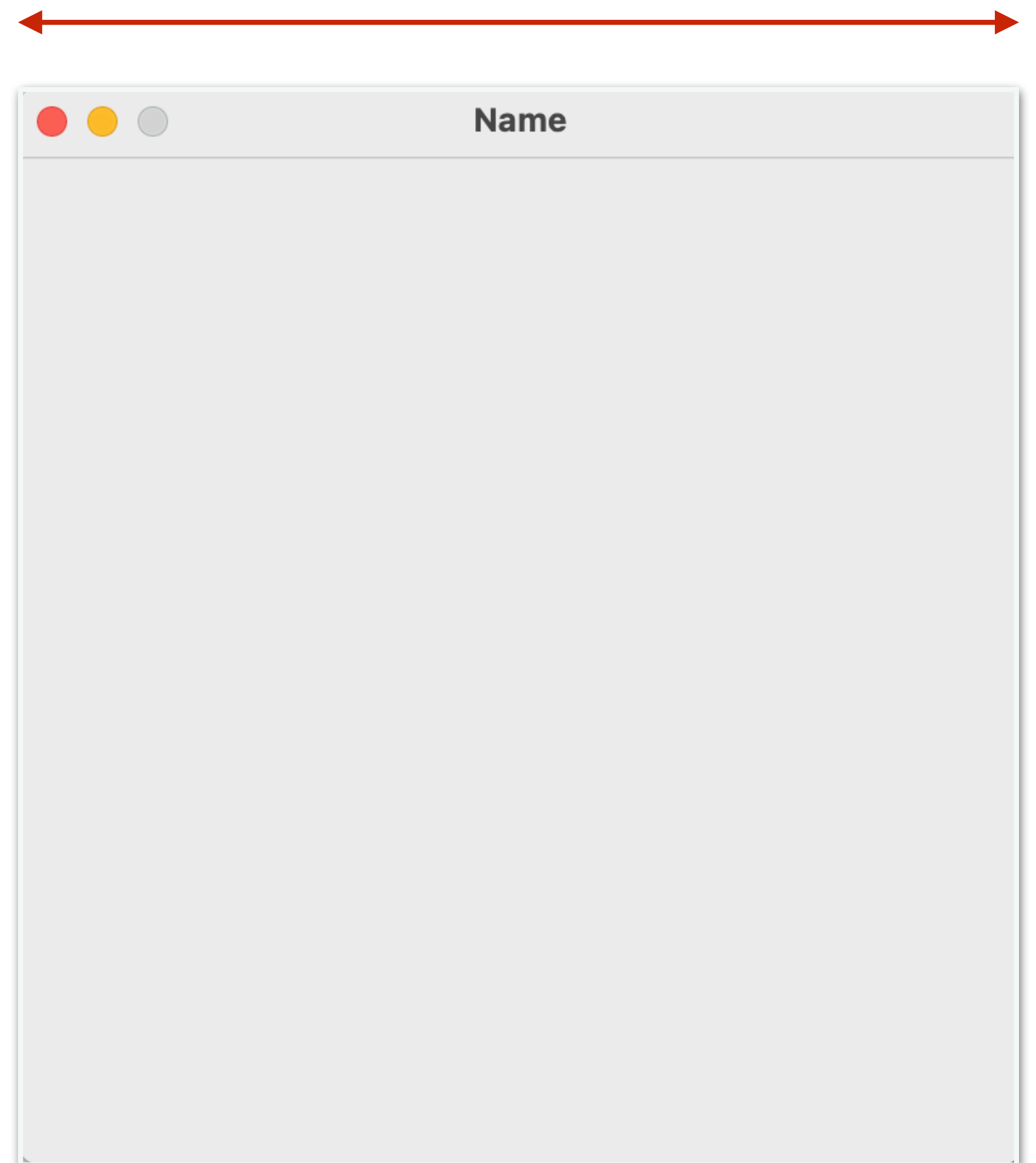
```
>>> from graphics import *
>>> # takes title and size of window
>>> win = GraphWin("Name", 400, 400)
```

We are going to use a simple graphics package to implement our game board

Create a window with title "Name" and size 400x400 (measured in pixels)

400 pixels

400 pixels



A **pixel** is one of the small dots or squares that make up an image on a computer screen.

Graphics Package for Board

```
>>> from graphics import *
>>> # takes title and size of window
>>> win = GraphWin("Name", 400, 400)
```

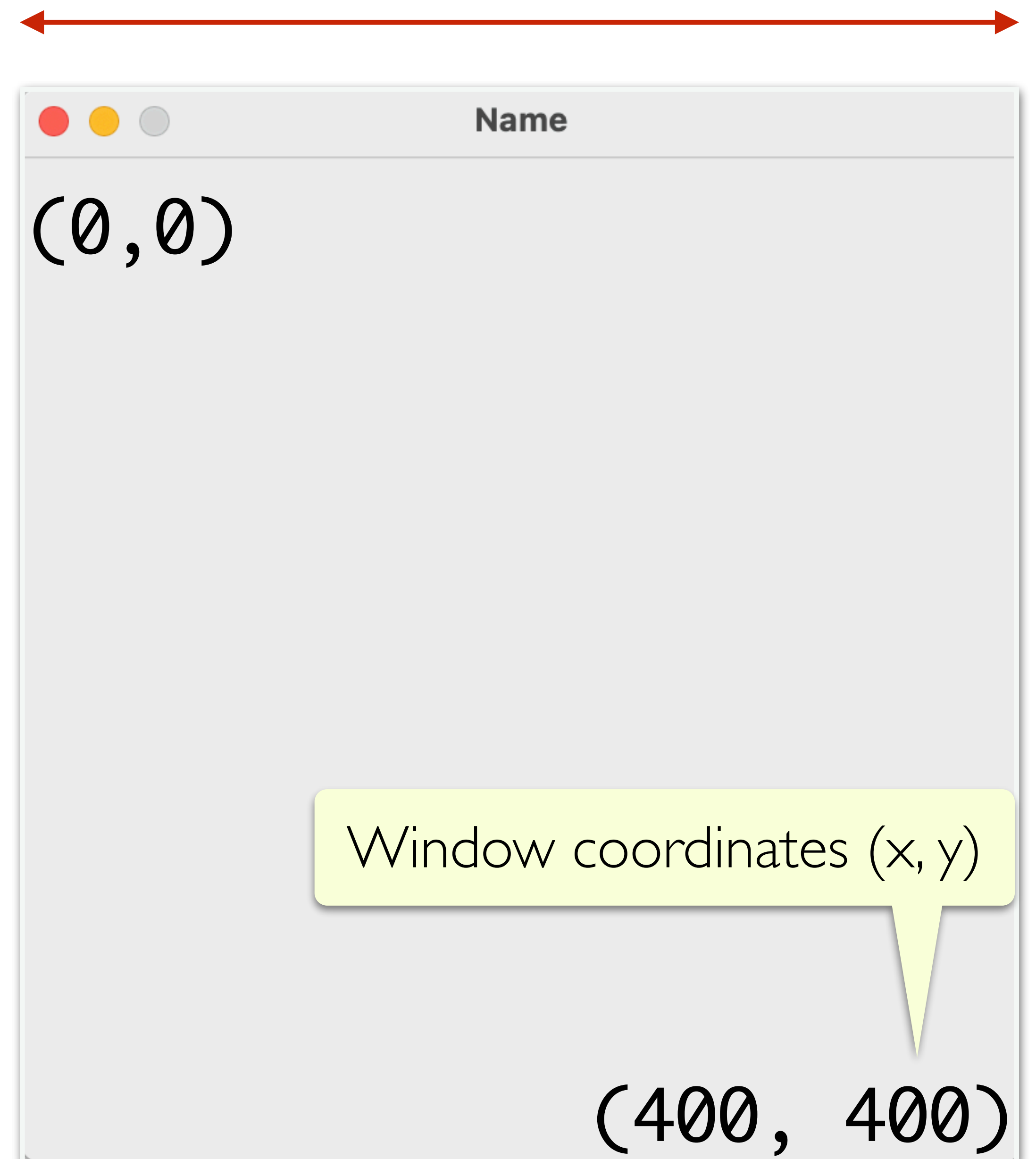
We are going to use a simple graphics package to implement our game board

Create a window with title "Name" and size 400x400 (measured in pixels)

400 pixels

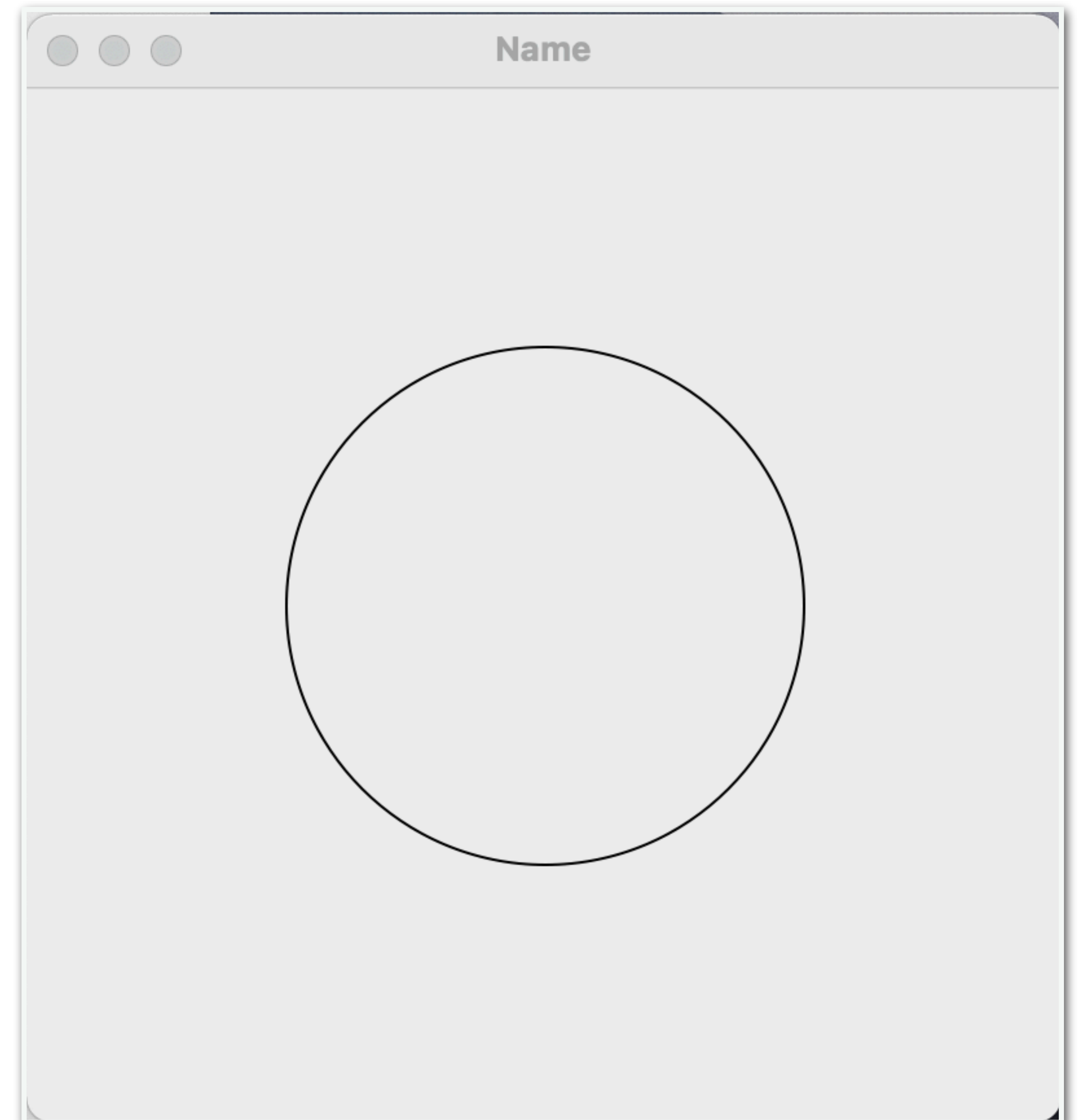
400 pixels

A **pixel** is one of the small dots or squares that make up an image on a computer screen.



Graphics Package for Board

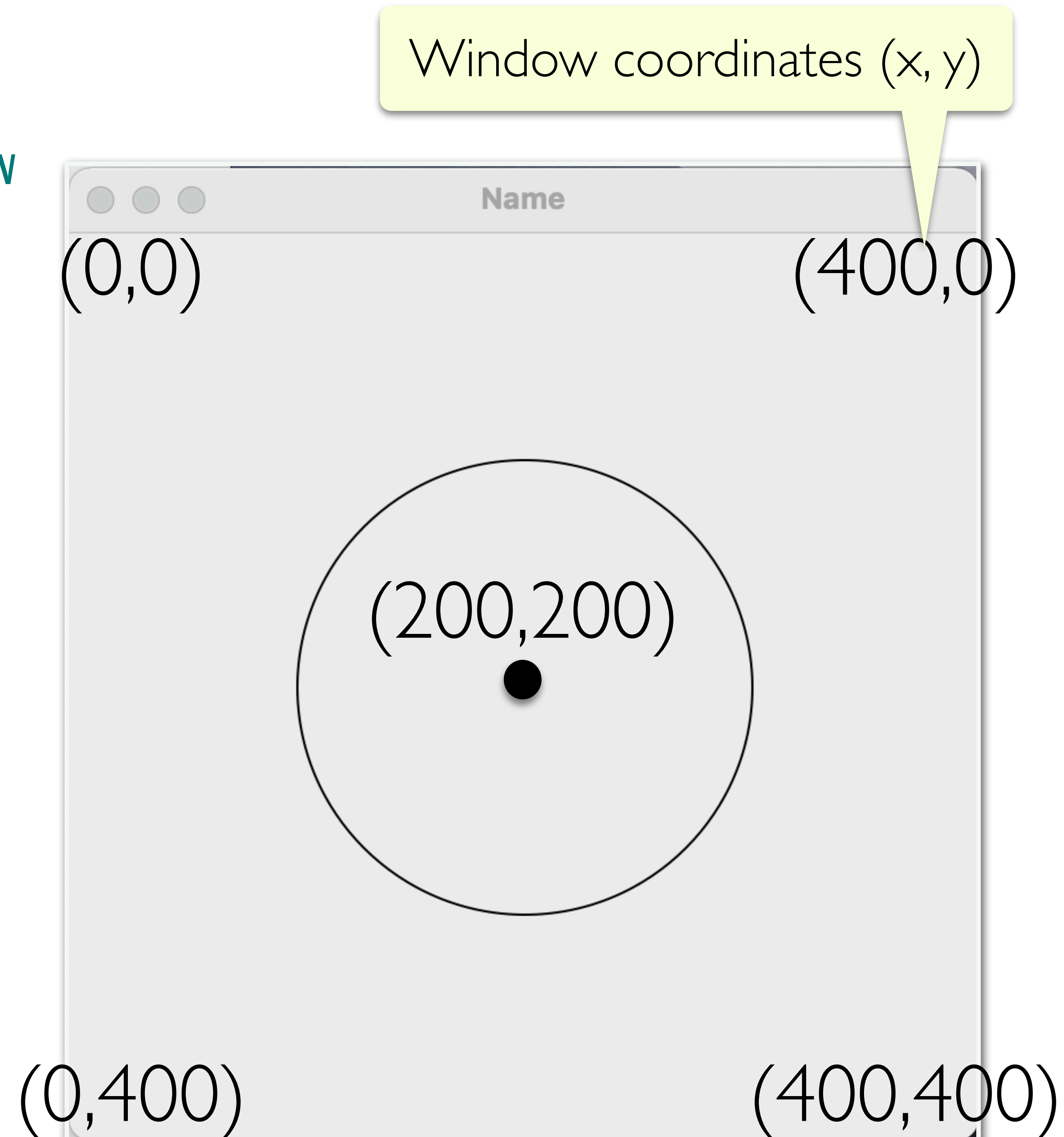
```
>>> # create point obj at x,y coordinate in window
>>> pt = Point(200, 200)
>>> # create circle w center at pt and radius 100
>>> c = Circle(pt, 100)
>>> # draw the circle on the window
>>> c.draw(win)
Circle(Point(200.0, 200.0), 100)
```



Graphics Package for Board

```
>>> # create point obj at x,y coordinate in window
>>> pt = Point(200, 200)
>>> # create circle w center at pt and radius 100
>>> c = Circle(pt, 100)
>>> # draw the circle on the window
>>> c.draw(win)
Circle(Point(200.0, 200.0), 100)
```

Window coordinates (x, y)



We can draw other shapes as well.

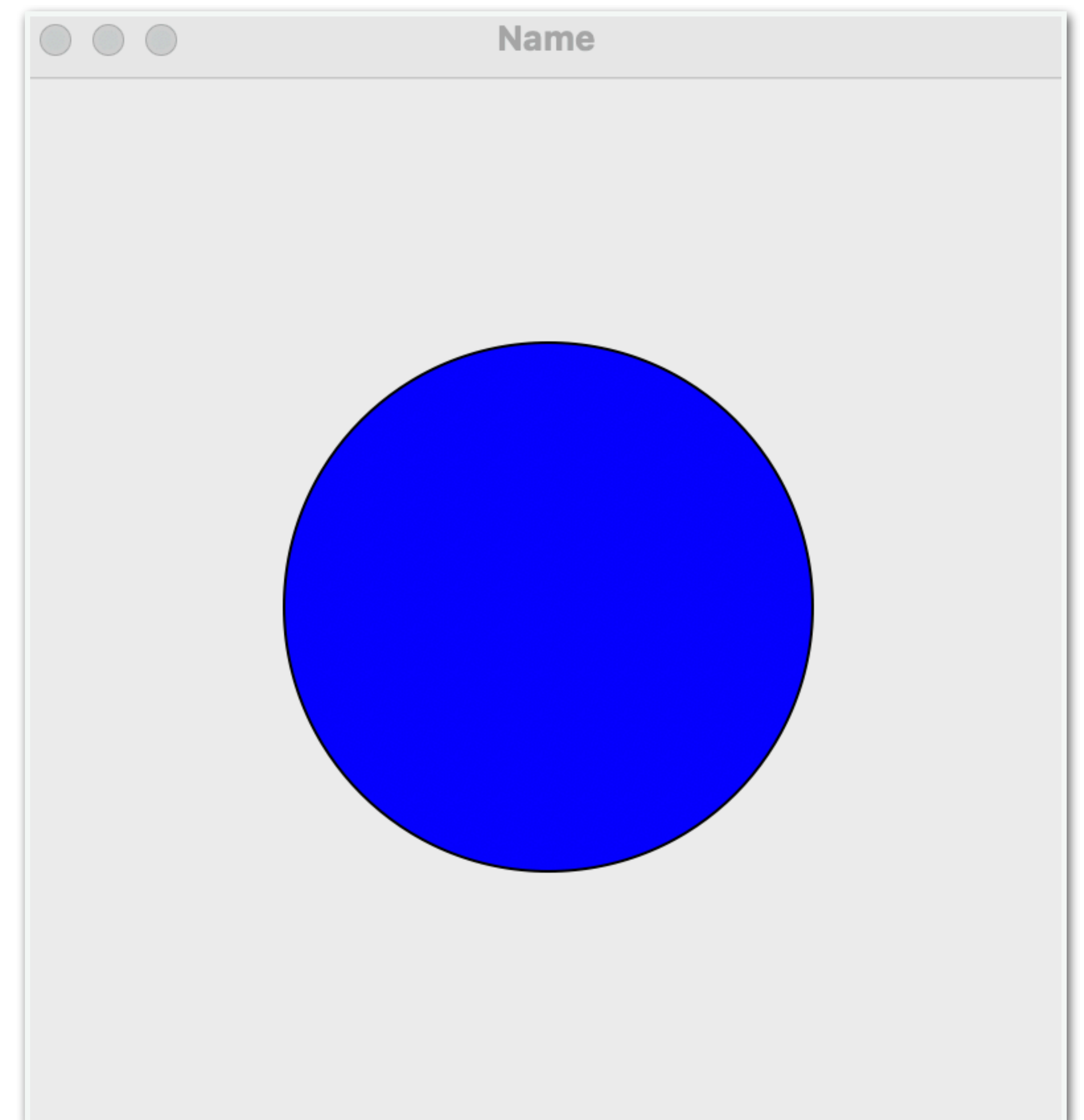
We'll want to draw Rectangles in our Board class.

Graphics Package for Board

```
>>> # set color to blue
>>> c.setFill("blue")
>>> # Pause to view result
>>> win.getMouse()
Point(76.0, 322.0)
>>> # close window when done
>>> win.close()
```

Detecting "***events***" like mouse clicks are an important part of a graphical program.

`win.getMouse()` is a ***blocking*** method call that "blocks" or ***waits*** until a click is detected.

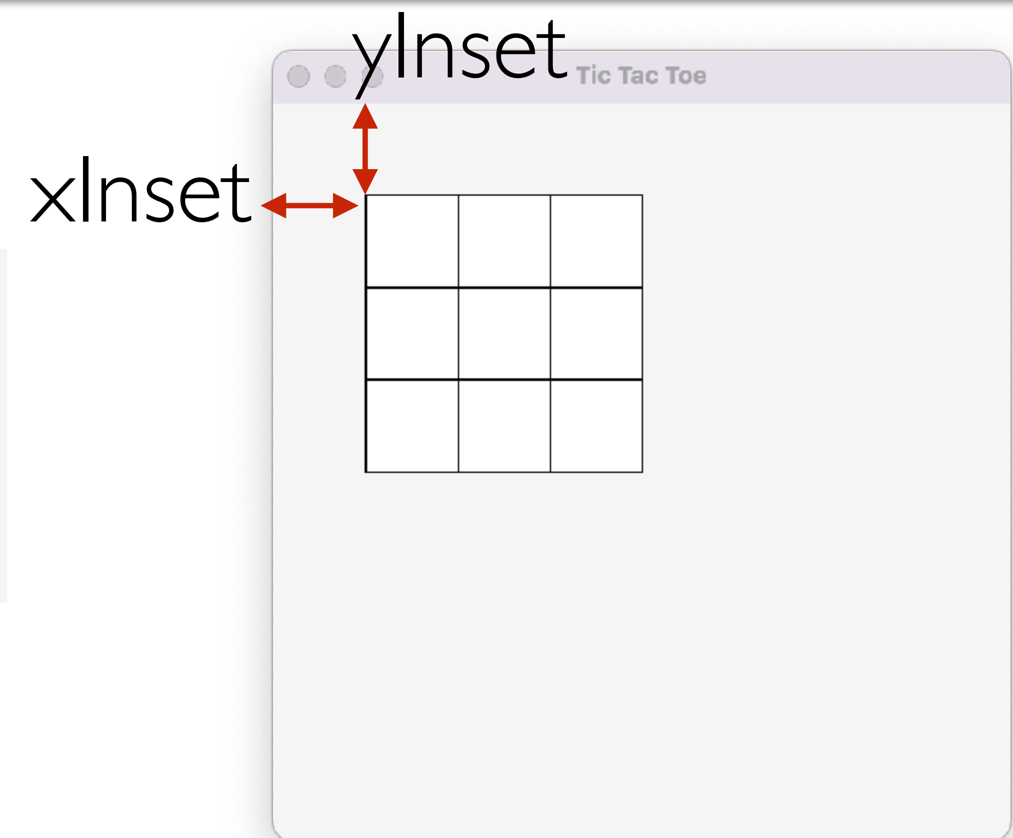


Board Class

Board class: Getting Started

- Attributes:

```
# _win: graphical window on which we will draw our board
# _xInset: avoids drawing in corner of window
# _yInset: avoids drawing in corner of window
# _rows: number of rows in grid of squares
# _cols: number of columns in grid of squares
# _size: edge size of each square
```

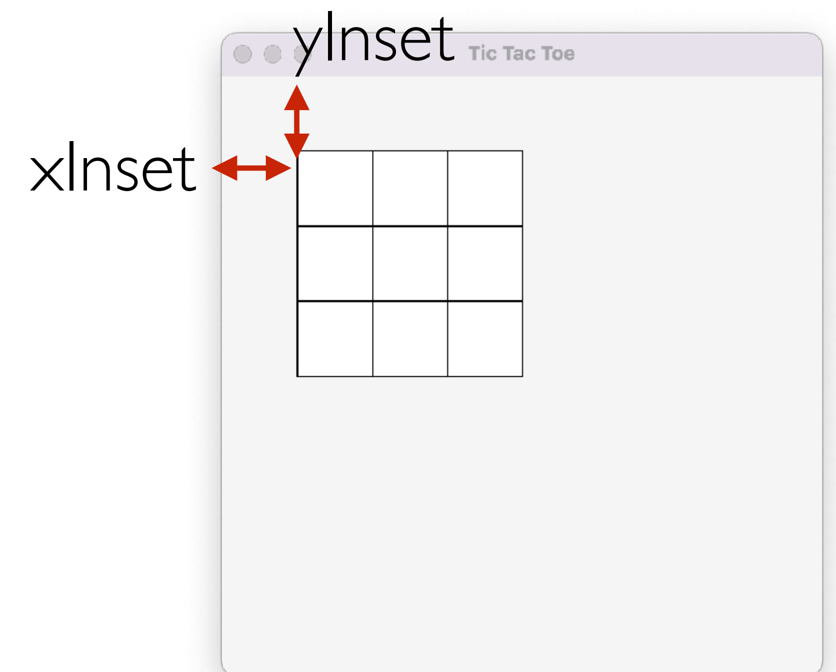


- (We will add a few more attributes later)
- We need to draw the **grid**, **text areas**, and **buttons**
- Might need some helper methods to organize our code
- Let's start by **drawing the grid** on our board

Board Class: __init__ and getters

```python
class Board:
    # _win: graphical window on which we will draw our board
    # _xinset: avoids drawing in corner of window
    # _yinset: avoids drawing in corner of window
    # _rows: number of rows in grid of squares
    # _cols: number of columns in grid of squares
    # _size: edge size of each square

    __slots__ = [ '_xinset', '_yinset', '_rows', '_cols', '_size', \
                  '_win', '_exit_button', '_reset_button', \
                  '_text_area', '_lower_word', '_upper_word']

    def __init__(self, win, xinset=50, yinset=50, rows=3, cols=3, size=50):
        # update class attributes
        self._xinset = xinset; self._yinset = yinset
        self._rows = rows; self._cols = cols
        self._size = size
        self._win = win
        self.draw_board()

    # getter methods for attributes
    def get_win(self):
        return self._win

    def get_xinset(self):
        return self._xinset

    def get_yinset(self):
        return self._yinset

    def get_rows(self):
        return self._rows

    def get_cols(self):
        return self._cols

    def get_size(self):
        return self._size

    def get_board(self):
        return self
```

Notice the default values



Board class: Drawing the grid

```
def _make_rect(self, point1, point2, fillcolor="white", text=""):
    """Creates a rectangle with text in the center"""
    rect = Rectangle(point1, point2, fillcolor)
    rect.draw(self._win)
    text = Text(rect.getCenter(), text)
    text.setTextColor("black")
    text.draw(self._win)
    return rect
```

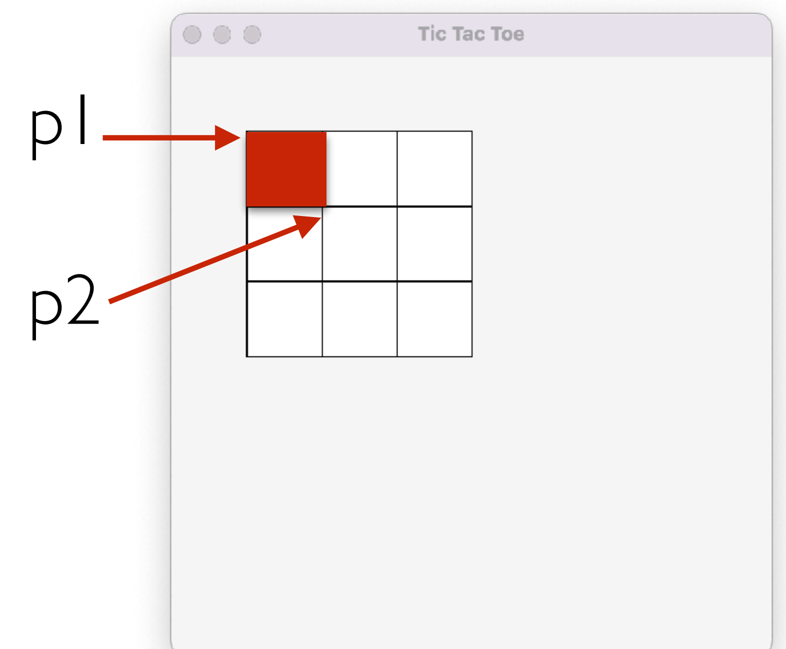
We always need a window (_win) on which to `draw`.

```
def __draw_grid(self):
    """Creates a row x col grid, filled with empty squares"""
    for x in range(self._cols):
        for y in range(self._rows):
            # create first point
            p1 = Point(self._xinset + self._size * x,
                       self._yinset + self._size * y)
            # create second point
            p2 = Point(self._xinset + self._size * (x + 1),
                       self._yinset + self._size * (y + 1))
            # create rectangle and add to graphical window
            self._make_rect(p1, p2)
```

Board class: Drawing the grid

```
def __draw_grid(self):
    """Creates a row x col grid, filled with empty squares"""
    for x in range(self._cols):
        for y in range(self._rows):
            # create first point
            p1 = Point(self._xinset + self._size * x,
                       self._yinset + self._size * y)
            # create second point
            p2 = Point(self._xinset + self._size * (x + 1),
                       self._yinset + self._size * (y + 1))
            # create rectangle and add to graphical window
            self._make_rect(p1, p2)
```

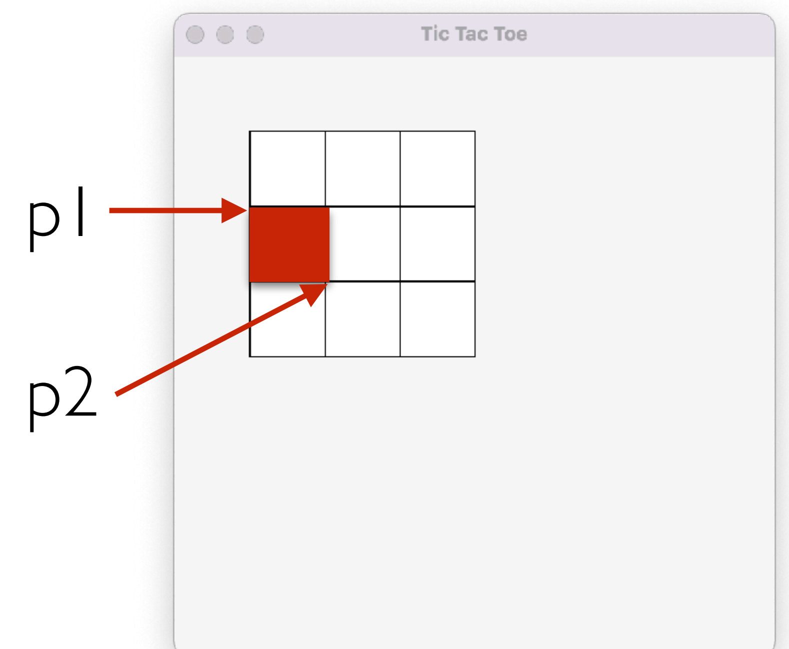
```
x=0, y=0:
 p1:
 xInset + (size * x) = xInset
 yInset + (size * y) = yInset
 p2:
 xInset + (size * (x+1)) = xInset + size
 yInset + (size * (y+1)) = yInset + size
```



Board class: Drawing the grid

```
def __draw_grid(self):
    """Creates a row x col grid, filled with empty squares"""
    for x in range(self._cols):
        for y in range(self._rows):
            # create first point
            p1 = Point(self._xinset + self._size * x,
                       self._yinset + self._size * y)
            # create second point
            p2 = Point(self._xinset + self._size * (x + 1),
                       self._yinset + self._size * (y + 1))
            # create rectangle and add to graphical window
            self._make_rect(p1, p2)
```

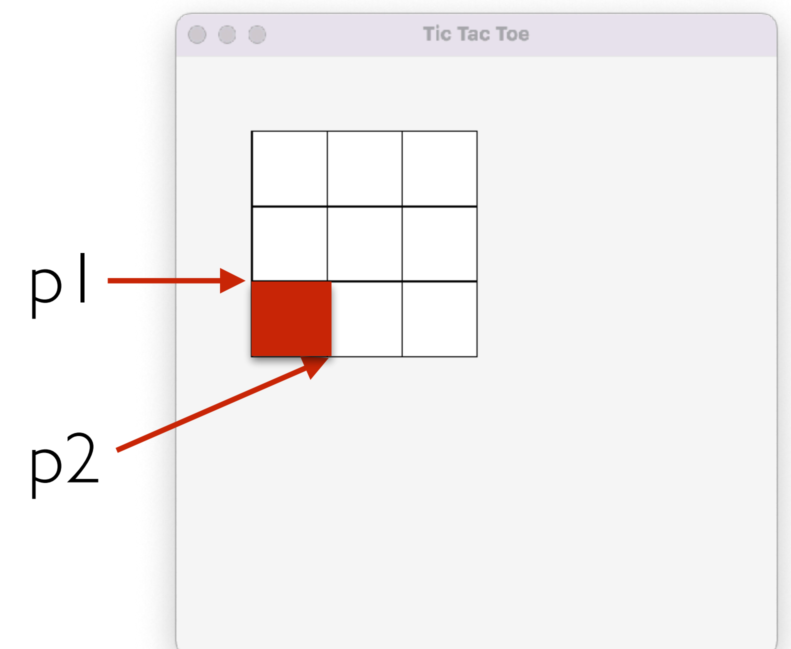
```
x=0, y=1:
  p1:
  xInset + (size * x) = xInset
  yInset + (size * y) = yInset + size
  p2:
  xInset + (size * (x+1)) = xInset + size
  yInset + (size * (y+1)) = yInset + 2 * size
```



Board class: Drawing the grid

```
def __draw_grid(self):
    """Creates a row x col grid, filled with empty squares"""
    for x in range(self._cols):
        for y in range(self._rows):
            # create first point
            p1 = Point(self._xinset + self._size * x,
                       self._yinset + self._size * y)
            # create second point
            p2 = Point(self._xinset + self._size * (x + 1),
                       self._yinset + self._size * (y + 1))
            # create rectangle and add to graphical window
            self._make_rect(p1, p2)
```

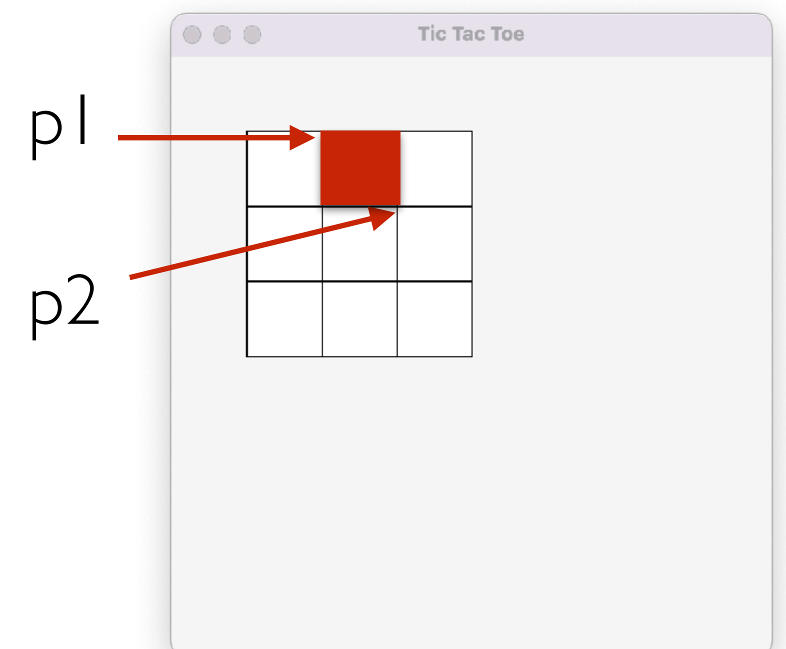
```
x=0, y=2:
 p1:
 xinset + (size * x) = xinset
 yinset + (size * y) = yinset + 2 * size
 p2:
 xinset + (size * (x+1)) = xinset + size
 yinset + (size * (y+1)) = yinset + 3 * size
```



Board class: Drawing the grid

```
def __draw_grid(self):
    """Creates a row x col grid, filled with empty squares"""
    for x in range(self._cols):
        for y in range(self._rows):
            # create first point
            p1 = Point(self._xinset + self._size * x,
                       self._yinset + self._size * y)
            # create second point
            p2 = Point(self._xinset + self._size * (x + 1),
                       self._yinset + self._size * (y + 1))
            # create rectangle and add to graphical window
            self._make_rect(p1, p2)
```

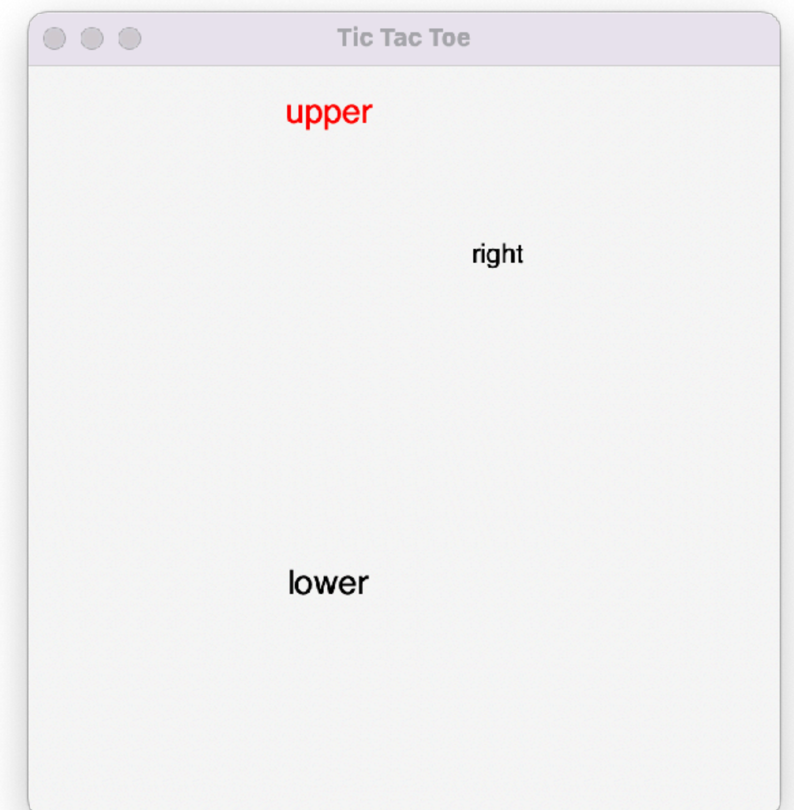
```
x=1, y=0:
 p1:
 xinset + (size * x) = xInset + size
 yinset + (size * y) = yInset
 p2:
 xinset + (size * (x+1)) = xInset + 2 * size
 yinset + (size * (y+1)) = yInset + size
```



And so on…

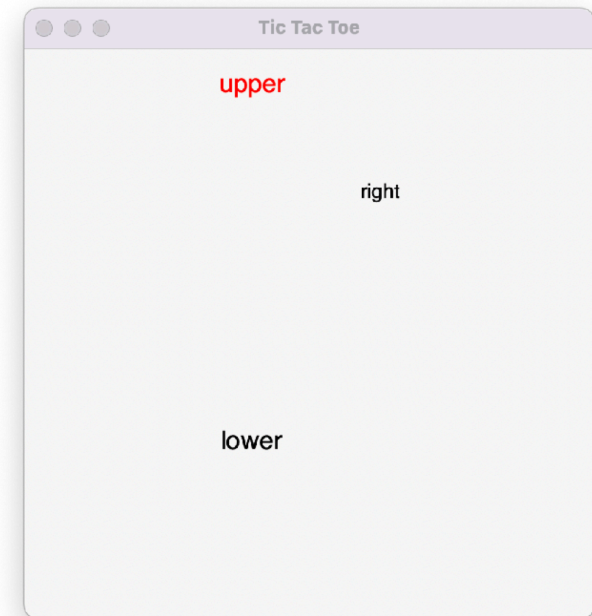
Board Class: Text Areas

- We need to draw the **grid**, **text areas**, and **buttons**
- Might need some helper methods to organize our code
- Now let's **draw the text areas** (we need 3!)
  - Text areas are just called `Text` objects in our graphics package
  - Can customize the font size, color, style, and size and call "`setText`" to add text



Board class: Drawing the Text Areas

- We'll add attributes for the text areas:
  `_text_area, _lower_word, _upper_word`

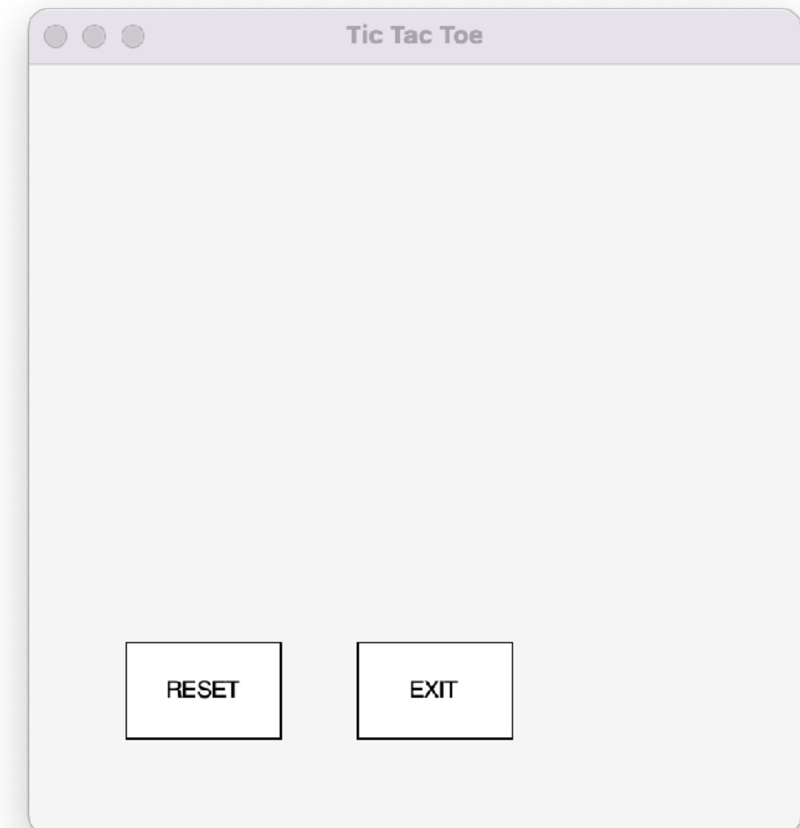



```
def __make_text_area(self, point, fontsize=18, color="black", text=""):
    """Creates a text area"""
    text_area = Text(point, text)
    text_area.setSize(fontsize)
    text_area.setTextColor(color)
    text_area.setStyle("normal")
    text_area.draw(self._win)
    return text_area

def __draw_text_areas(self):
    """Draw the text areas to the right/lower/upper side of main grid"""
    # draw main text area (right of grid)
    self._text_area = self.__make_text_area(Point(self._xinset * self._rows + self._size * 2,
                                                   self._yinset + 50), 14)
    #draw the text area below grid
    self._lower_word = self.__make_text_area(Point(160, 275))
    #draw the text area above grid
    self._upper_word = self.__make_text_area(Point(160, 25), color="red")
```

Board Class: Draw Buttons

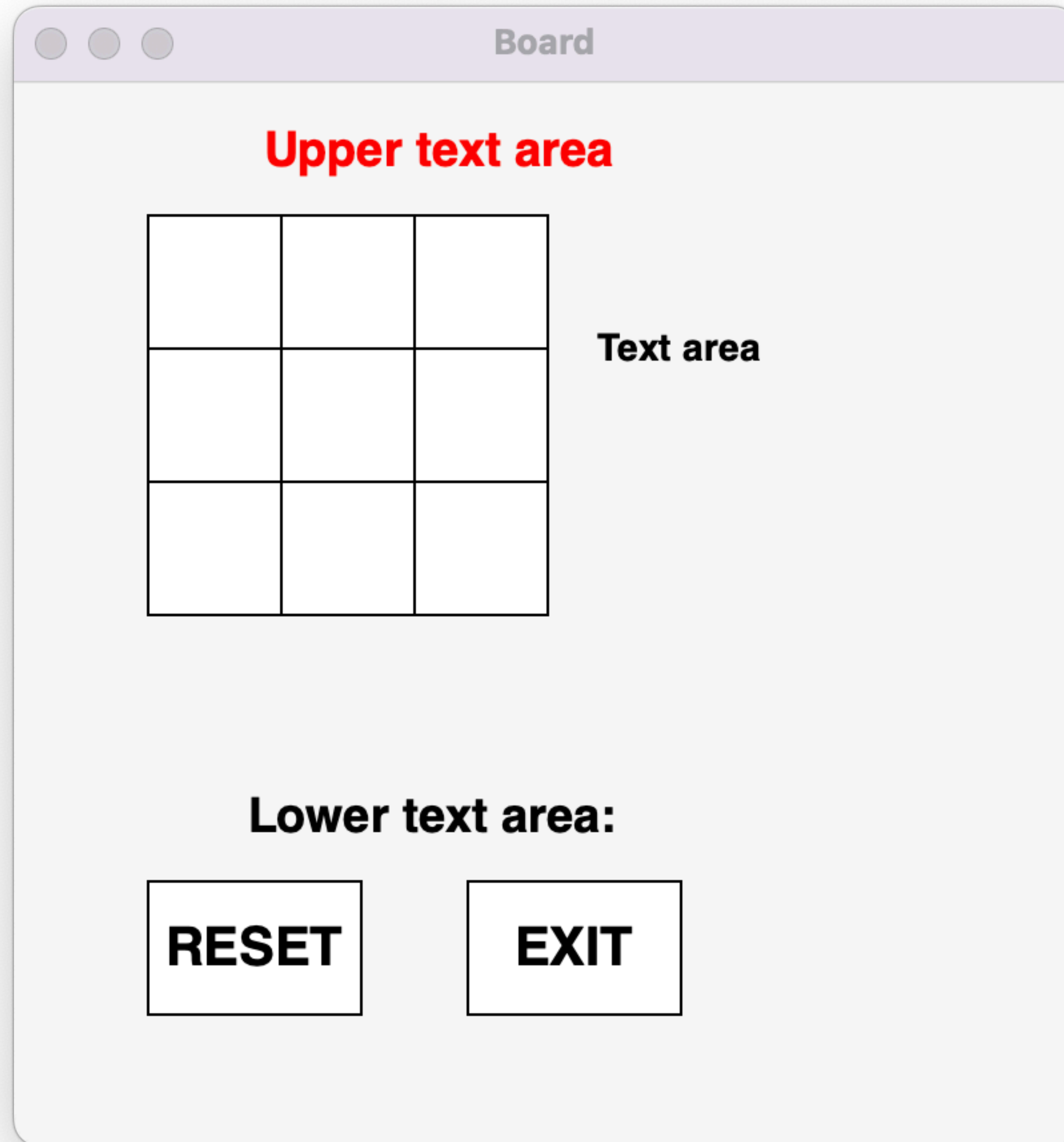
- We need to draw the **grid**, **text areas**, and **buttons**
- Might need some helper methods to organize our code
- Finally, let's **draw the buttons**!
  - Buttons are just more rectangles…

Board Class: Draw Buttons




```
def __draw_buttons(self):
    """Create reset and exit buttons"""
    p1 = Point(50, 300); p2 = Point(130, 350)
    self._reset_button = self._make_rect(p1, p2, text="RESET")
    p3 = Point(170, 300); p4 = Point(250, 350)
    self._exit_button = self._make_rect(p3, p4, text="EXIT")

def draw_board(self):
    """Create the board with the grid, text areas, and buttons"""
    self._win.setBackground("white smoke")
    self.__draw_grid()
    self.__draw_text_areas()
    self.__draw_buttons()
```

Putting it all together



Board Helper Methods

Helper Methods

- Now that we have a board with a grid, buttons, and text areas, it would be useful to define some methods for interacting with these objects
- Helpful methods?

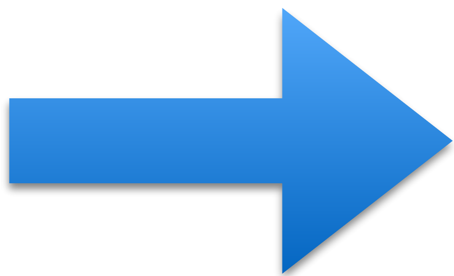
Helper Methods

- Now that we have a board with a grid, buttons, and text areas, it would be useful to define some methods for interacting with these objects
- Helpful methods?
  - Get grid coordinate of mouse click
  - Determine if click was in grid, reset, or exit buttons
  - Set text to one of 3 text areas
  - …
- Note that none of this is specific to Tic Tac Toe (yet)!
- Always good to start general and then get more specific

Helper Methods

>>> pydoc3 board

Public methods!



```
CLASSES
    builtins.object
        Board

    class Board(builtins.object)
     |  Board(win, xinset=50, yinset=50, rows=3, cols=3, size=50)
     |
     |  Methods defined here:
     |
     |  __init__(self, win, xinset=50, yinset=50, rows=3, cols=3, s
     |      Initialize self.  See help(type(self)) for accurate sig
     |
     |  draw_board(self)
     |      Create the board with the grid, text areas, and button
     |
     |  get_board(self)
     |
     |  get_cols(self)
     |
     |  get_position(self, point)
     |      Converts a window location (Point) to a grid position
     |      Note: Grid positions are always returned as col, row.
     |
     |  get_rows(self)
     |
     |  get_size(self)
     |
     |  get_string_from_lower_text(self)
     |      Get text from text area below grid.
     |
     |  get_string_from_text_area(self)
     |      Get text from text area to right of grid.
     |
     |  get_string_from_upper_text(self)
     |      Get text from text area above grid.
     |
```