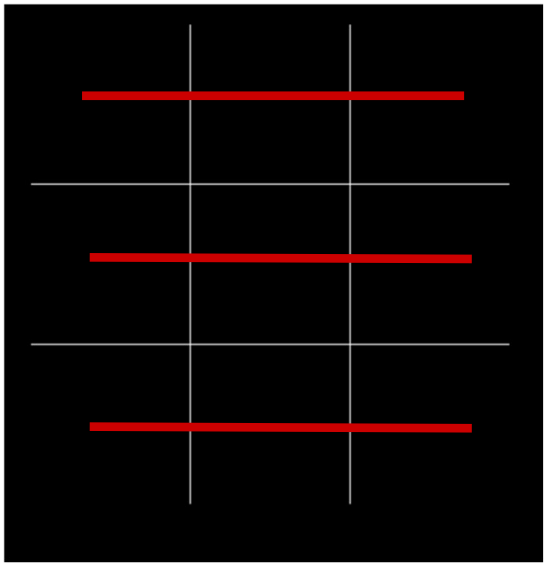
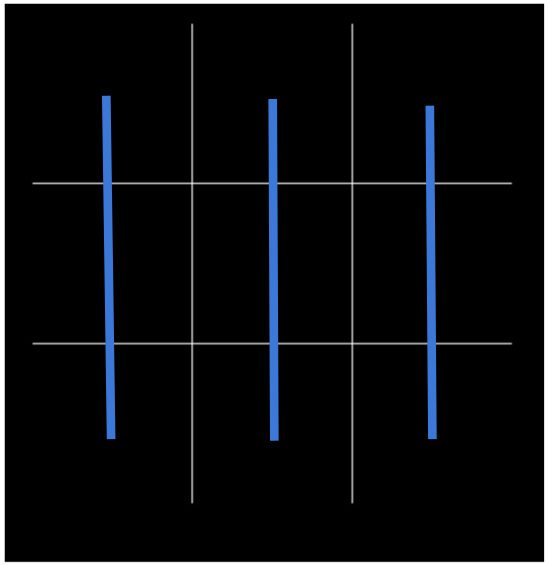
Intro to Coding

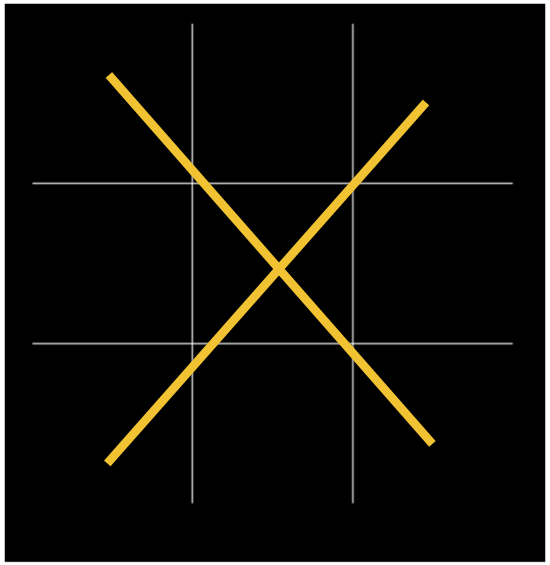
Class 10

Checking for a Win

There are 8 different ways to win in Tic Tac Toe: 3 ways for rows, 3 for columns, and 2 for diagonals



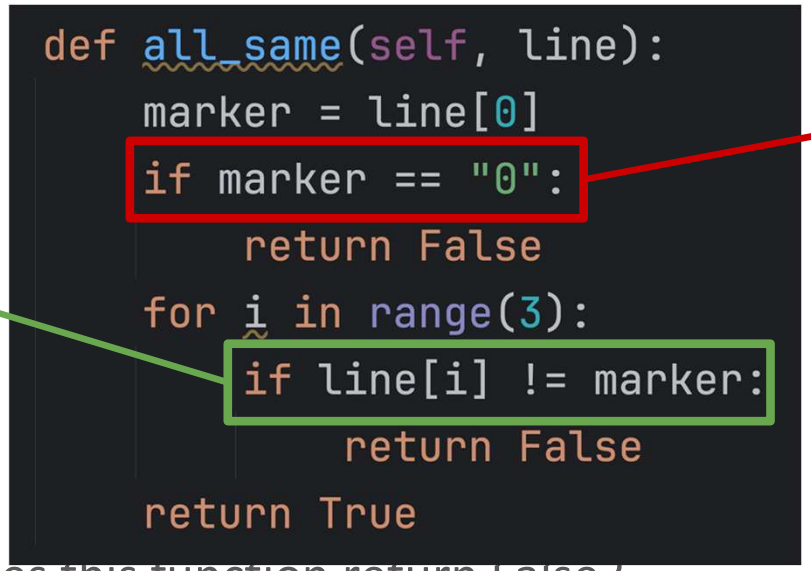




It would be too tedious to individually check for all 8 cases; we can make this more efficient with loops.

Checking if a List Contains Three of the Same Marker

This function compares the first element of the list with all the other elements inside it.



If the **current** element in the list is **not equal** to the **first** element, then the **entire** list is guaranteed to not contain the same marker

Recall: a value of "0" indicates that a square is empty

Under what conditions does this function return False?

Checking for a Win

Now, we can build our check_win() function. We can build a list to represent each row, column, and diagonal, and check to see if any of them contain three of the same marker.

```
def check_win(self):
    # checking rows
    for i in self.board_values:
        if self.all_same(i):
            self.game_over = True
            return i[0]
```

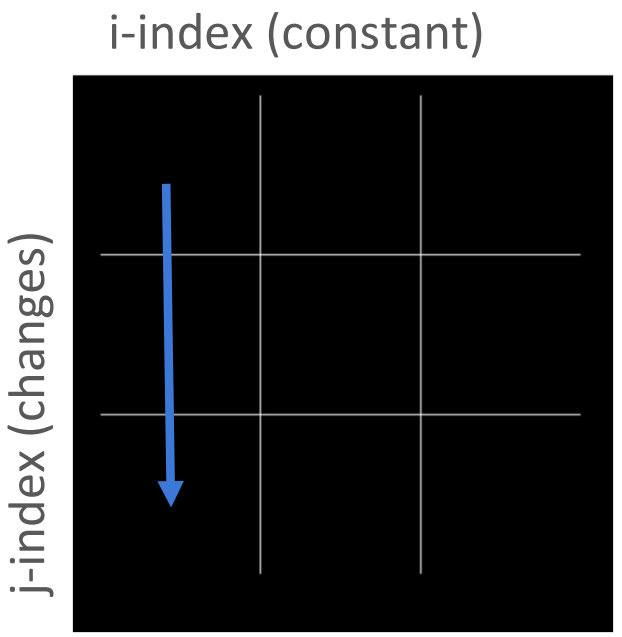
We can use an iterator-based loop to obtain the list form of each row, and check if they contain the same marker.

Set game_over as True to prevent any further changes to the board. Then, return the string of the winning marker

Checking for a Win

For columns, the list isn't "given" to us as with rows, so we need to build it ourselves. Note that we check to see if they contain three of the same marker after we have finished building the list for one column.

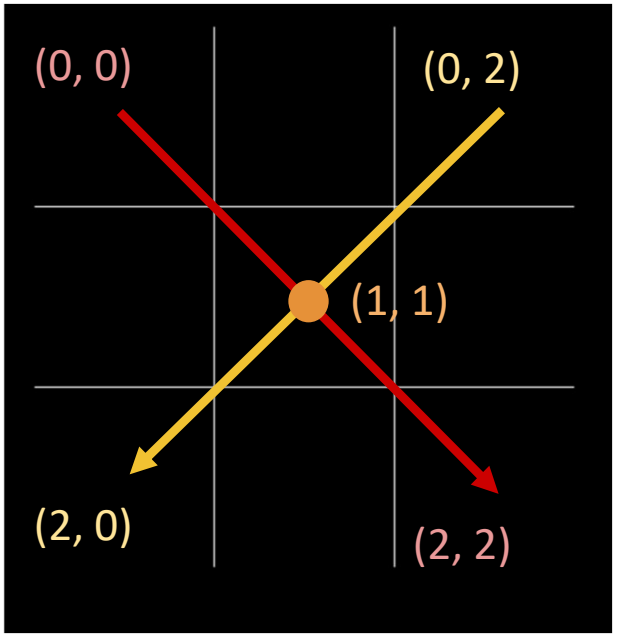
```
# checking columns
for i in range(3):
    col = []
    for j in range(3):
        col.append(self.board_values[j][i])
    if self.all_same(col):
        self.game_over = True
        return col[0]
```



Checking for a Win

Lastly, we build two lists for each diagonal. We can build both lists simultaneously, and then check to see if either list has 3 of the same marker.

```
# checking diagonals
diag1 = []
diag2 = []
for i in range(3):
    diag1.append(self.board_values[i][i])
    diag2.append(self.board_values[i][2 - i])
if self.all_same(diag1):
    self.game_over = True
    return diag1[0]
elif self.all_same(diag2):
    self.game_over = True
    return diag2[0]
```



get_col() and get_diag()

We can make our code even cleaner by creating two functions that return the desired column or diagonals in list form.

```
def get_col(self, c):
    col = []
    for i in range(3):
        col.append(self.board_values[i][c])
    return col

def get_diag(self):
    diag1 = []
    diag2 = []
    for i in range(3):
        diag1.append(self.board_values[i][i])
        diag2.append(self.board_values[i][2 - i])
    return diag1, diag2
```

get_col() reuses earlier logic with small adjustments

The code for get_diag() is the same as before

Simplifying the check_win() Function

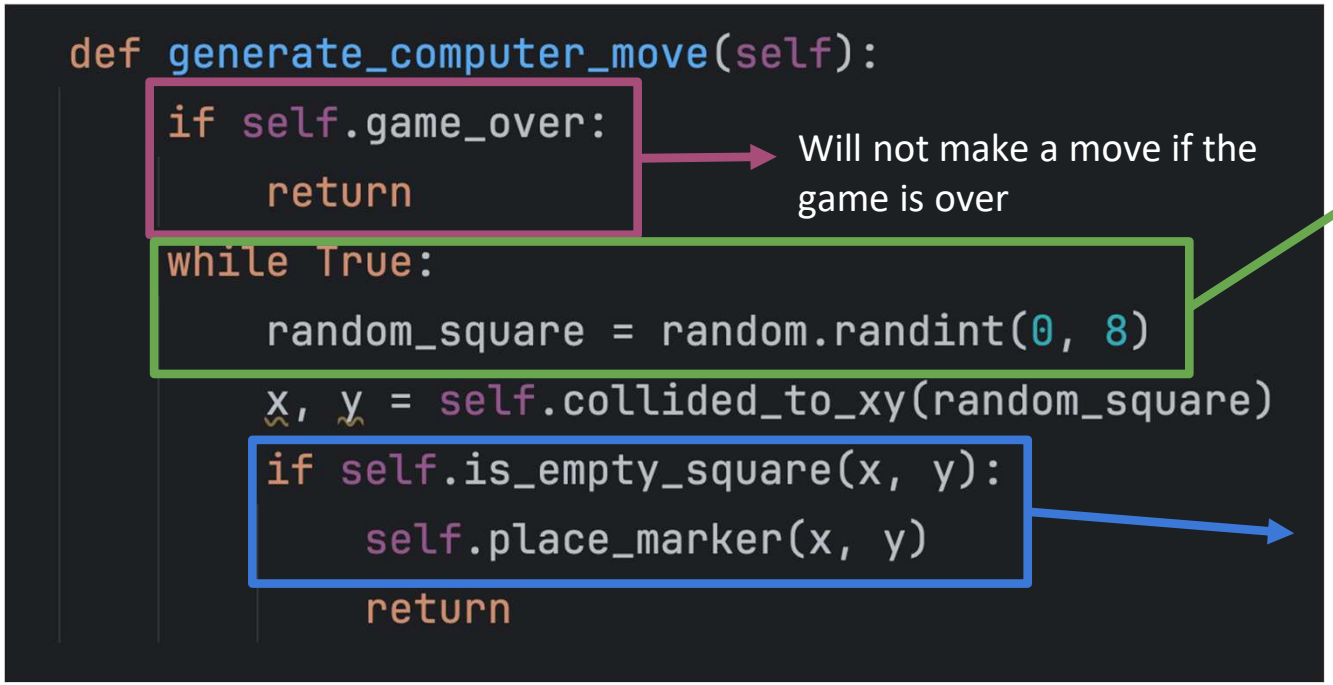
Using our newly made functions, how can we simplify the code in the check_win() function?

```
# checking columns
for i in range(3):
    col = self.get_col(i)
    if self.all_same(col):
        self.game_over = True
        return col[0]

# checking diagonals
diag1, diag2 = self.get_diag()
if self.all_same(diag1):
    self.game_over = True
    return diag1[0]
elif self.all_same(diag2):
    self.game_over = True
    return diag2[0]
```

Use the get_col() and get_diag() functions

Generating Computer Moves

Now, it's time to implement a computer opponent to play against. For now, it will select a random square to place its marker.



Updating place_marker() Function

We need to add two lines of code to our place_marker() function. This ensures that the computer only makes a move on its turn. self.generate_computer_moves indicates whether or not the computer should make a move on the current turn.

```
def place_marker(self, x, y):
    self.board_values[x][y] = self.current_move
    self.switch_move()
    if self.alternate_computer_move_value:
        self.generate_computer_moves = not self.generate_computer_moves
```

If singleplayer is selected

Flips the value of self.generate_computer_moves

Rendering Winner Font

Let's return to our main loop. These lines of code should be placed below the "Current Mode" part of text rendering.

Why is an if statement needed here?

```
# checking for a winner
winner = BOARD.check_win()
text_winner = small_font.render("Winner: ", True, (255, 255, 255))
if winner is not None:
    text_winner = small_font.render(f"Winner: {winner}", True, (255, 255, 255))
text_winner_rect = text_winner.get_rect()
text_winner_rect.center = (100, 135)
screen.blit(text_winner, text_winner_rect)
```

Rendering Winner Font

Let's return to our main loop. These lines of code should be placed below the "Current Mode" part of text rendering.

An if statement ensures that the winning marker is rendered on the screen.

```
# checking for a winner
winner = BOARD.check_win()
text_winner = small_font.render("Winner: ", True, (255, 255, 255))
if winner is not None:
    text_winner = small_font.render(f"Winner: {winner}", True, (255, 255, 255))
text_winner_rect = text_winner.get_rect()
text_winner_rect.center = (100, 135)
screen.blit(text_winner, text_winner_rect)
```

Calling .generate_computer_move()

Lastly, we call the .generate_computer_move() function after we have checked for a win, but before updating the screen.

Why do we call the function after we check for a win first? What if we called the function before checking?

```
if BOARD.generate_computer_moves:
    BOARD.generate_computer_move()

# drawing markers onto the screen
for i in range(9):
    BOARD.draw_value(screen, i)

pygame.display.flip()
```

Ensures the computer only moves on its turn

Calling .generate_computer_move()

Lastly, we call the .generate_computer_move() function after we have checked for a win, but before updating the screen.

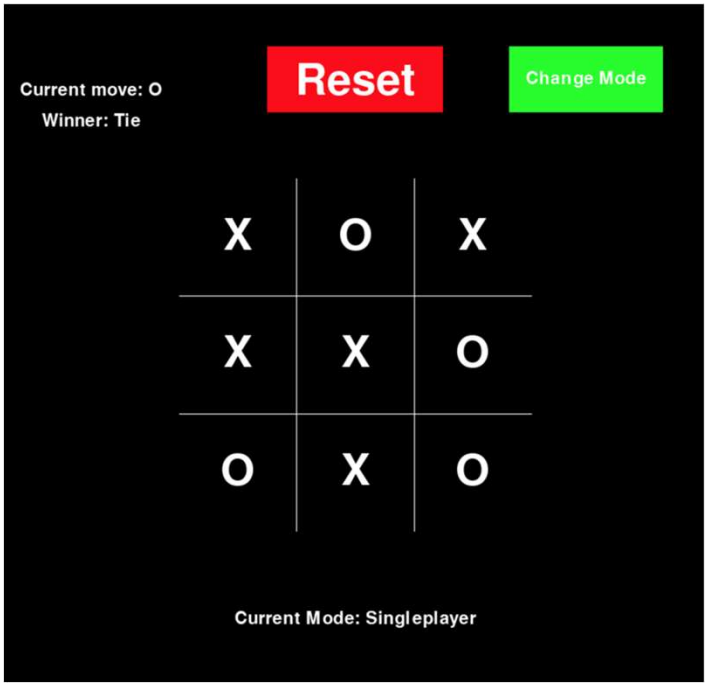
We call the function after we check for a win to prevent the computer from moving if the player has already won.

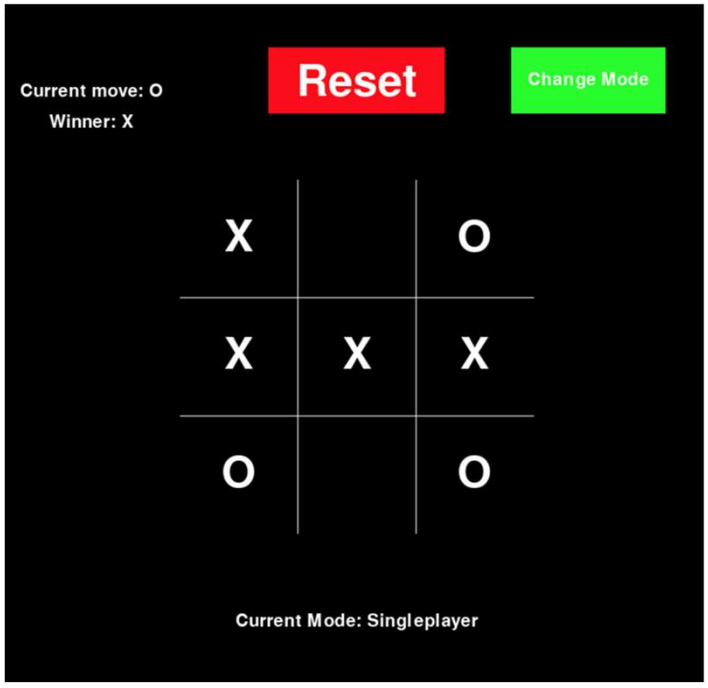
```
if BOARD.generate_computer_moves:
    BOARD.generate_computer_move()

# drawing markers onto the screen
for i in range(9):
    BOARD.draw_value(screen, i)

pygame.display.flip()
```

Ensures the computer only moves on its turn

Try it out!

Try running your code now. You should be able to play against the computer if you click the “Change Mode” button.





Improving Move Generation

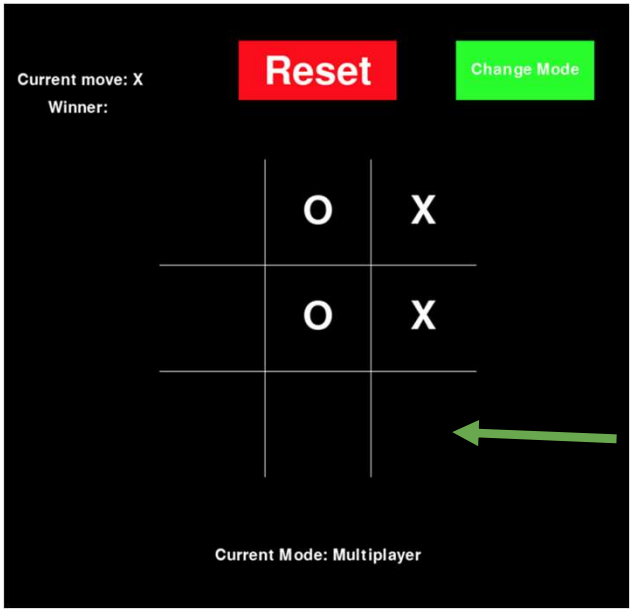
So far, the computer is pretty easy to beat, as it simply chooses a random square to place a move.

First, let's add an additional condition in our main loop so that the mode can only change when it's X's turn. This ensures that the computer will always play as O.

```
elif mode_button_rect.collidepoint(x, y) and BOARD.current_move == "X":
    BOARD.alternate_computer_move_value = not BOARD.alternate_computer_move_value
```

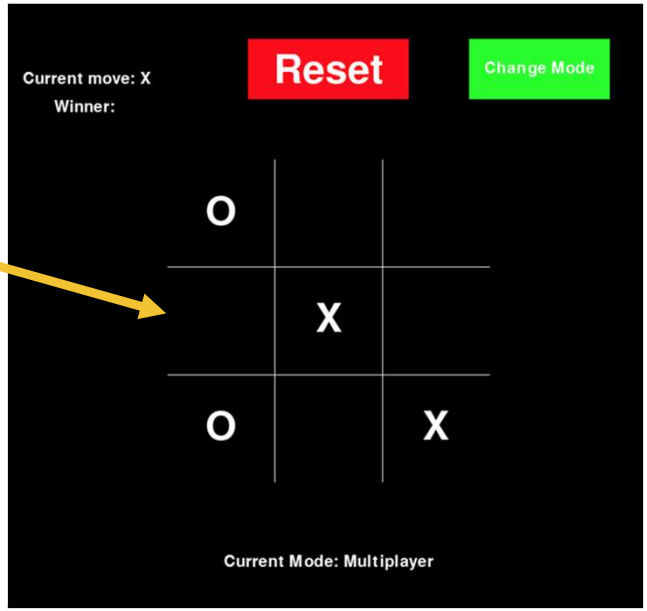
Connecting Two Pieces

In Tic-Tac-Toe, the most important situation to watch for is when a row, column, or diagonal already contains two of the same marker with one empty space left. For example:



X should move here to prevent O from winning

X should move here to win the game



Connecting Two Pieces

Now, let's implement this idea into our code.

Your Turn!

Using the same logic we used for rows, try implementing the code for the board's columns and diagonals.

Hint: What functions can you call to simplify your code?

Solution

What patterns are you noticing here?

Ensure that the markers are placed in the correct coordinates

If nothing was found, then return False

```
for i in range(3):
    col = self.get_col(i)
    if col.count(marker) == 2 and col.count("0") == 1:
        j = col.index("0")
        self.place_marker(j, i)
        return True

diag1, diag2 = self.get_diag()
if diag1.count(marker) == 2 and diag1.count("0") == 1:
    i = diag1.index("0")
    self.place_marker(i, i)
    return True
elif diag2.count(marker) == 2 and diag2.count("0") == 1:
    i = diag2.index("0")
    self.place_marker(i, 2 - i)
    return True
return False
```

Improving Move Generation

Let's update the generate_computer_move() function.

Does the order in which we check for O and X matter? Why or why not?

Which square has coordinates (1, 1)? Why might we check for this square individually?

```
def generate_computer_move(self):
    if self.game_over:
        return
    elif self.check_for_two("O"):
        return
    elif self.check_for_two("X"):
        return
    elif self.is_empty_square(1, 1):
        self.place_marker(1, 1)
        return
    while True:
        random_square = random.randint(0, 8)
        x, y = self.collided_to_xy(random_square)
        if self.is_empty_square(x, y):
            self.place_marker(x, y)
            return
```

check_for_two() will return a boolean value, which lets our code know if a move has already been made or not

The computer only chooses randomly if there are no better moves to make

Improving Move Generation

Let's update the generate_computer_move() function.

The order matters. We check to see if we can **win** before checking to see if we can **block**.

```
def generate_computer_move(self):
    if self.game_over:
        return
    elif self.check_for_two("O"):
        return
    elif self.check_for_two("X"):
        return
    elif self.is_empty_square(1, 1):
        self.place_marker(1, 1)
        return
    while True:
        random_square = random.randint(0, 8)
        x, y = self.collided_to_xy(random_square)
        if self.is_empty_square(x, y):
            self.place_marker(x, y)
            return
```

check_for_two() will return a boolean value, which lets our code know if a move has already been made or not

The center square has coordinates (1, 1). It's the best move to make, so we want to move there if it hasn't been occupied already.

The computer only chooses randomly if there are no better moves to make

Try it out!

We have successfully built a functional Tic Tac Toe game that supports both singleplayer and multiplayer. We have also improved our computer model to make smarter moves.