Using ChatGPT to make a game AI

Tero Halkoaho

Loihde Factor

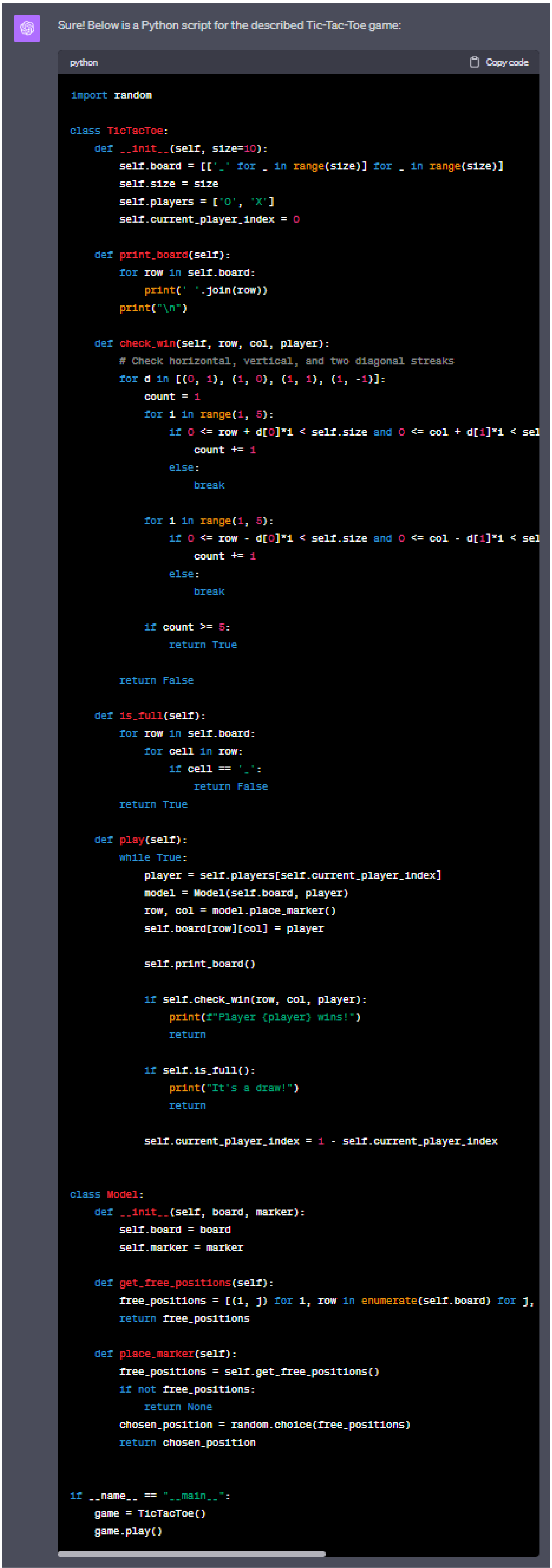
I came up with an interesting idea: using ChatGPT to analyze a game and develop an AI through an iterative process that can challenge a human. I would ask ChatGPT to create a simple game, in this case, the classic "Ristinolla" (i.e., TicTacToe). Then, I would utilize ChatGPT's ability to analyze gameplay to develop better gaming algorithms. I would repeat this until it beat me.

Could this work? I bet it could.

To start the process, I crafted the following prompt:

> **create me a python script that does the following:**
>
> **- creates a game of tic-tac-toe, where there are ten rows and columns instead of three. The players take turns to add markers (marked by either a O or an X) to a square without one, depending on which player they are. The game checks if any player has five or more similar markers in a row horizontally, vertically, or diagonally and the player who achieves to get five of their markers in a row first wins.**
>
> **- contains a method that prints out each board setting with markers signified by Os and Xs in appropriate places and _ -character where there is none. This method is called after each turn.**
>
> **- there is an model object that is responsible for placing the marker (either X or O depending on which side it is). The model has a method that gets called every time it is its turn, and the turn will end the moment it places the marker on the board. The other model will then take the turn and will place its marker until either one of the players wins, or there are no longer empty space to put the marker into. If there are no empty space to put a marker into, the game ends in a draw.**
>
> **- the models will place their corresponding markers randomly into any free space. This will be changed later to have better functionality.**

The response was quite a functional piece of Python:



Sure! Below is a Python script for the described Tic-Tac-Toe game:

```python
import random

class TicTacToe:
    def __init__(self, size=10):
        self.board = [['_' for _ in range(size)] for _ in range(size)]
        self.size = size
        self.players = ['O', 'X']
        self.current_player_index = 0

    def print_board(self):
        for row in self.board:
            print(' '.join(row))
        print("\n")

    def check_win(self, row, col, player):
        # Check horizontal, vertical, and two diagonal streaks
        for d in [(0, 1), (1, 0), (1, 1), (1, -1)]:
            count = 1
            for i in range(1, 5):
                if 0 <= row + d[0]*i < self.size and 0 <= col + d[1]*i < sel
                    count += 1
                else:
                    break

            for i in range(1, 5):
                if 0 <= row - d[0]*i < self.size and 0 <= col - d[1]*i < sel
                    count += 1
                else:
                    break

            if count >= 5:
                return True

        return False

    def is_full(self):
        for row in self.board:
            for cell in row:
                if cell == '_':
                    return False
        return True

    def play(self):
        while True:
            player = self.players[self.current_player_index]
            model = Model(self.board, player)
            row, col = model.place_marker()
            self.board[row][col] = player

            self.print_board()

            if self.check_win(row, col, player):
                print(f"Player {player} wins!")
                return

            if self.is_full():
                print("It's a draw!")
                return

            self.current_player_index = 1 - self.current_player_index


class Model:
    def __init__(self, board, marker):
        self.board = board
        self.marker = marker

    def get_free_positions(self):
        free_positions = [(i, j) for i, row in enumerate(self.board) for j,
        return free_positions

    def place_marker(self):
        free_positions = self.get_free_positions()
        if not free_positions:
            return None
        chosen_position = random.choice(free_positions)
        return chosen_position


if __name__ == "__main__":
    game = TicTacToe()
    game.play()
```

After ensuring it worked and that both players made random moves on the board, I refined it further.

> **modify this so that a human can be the second player, and the board is clickable with a mouse so that the player can add markers:**

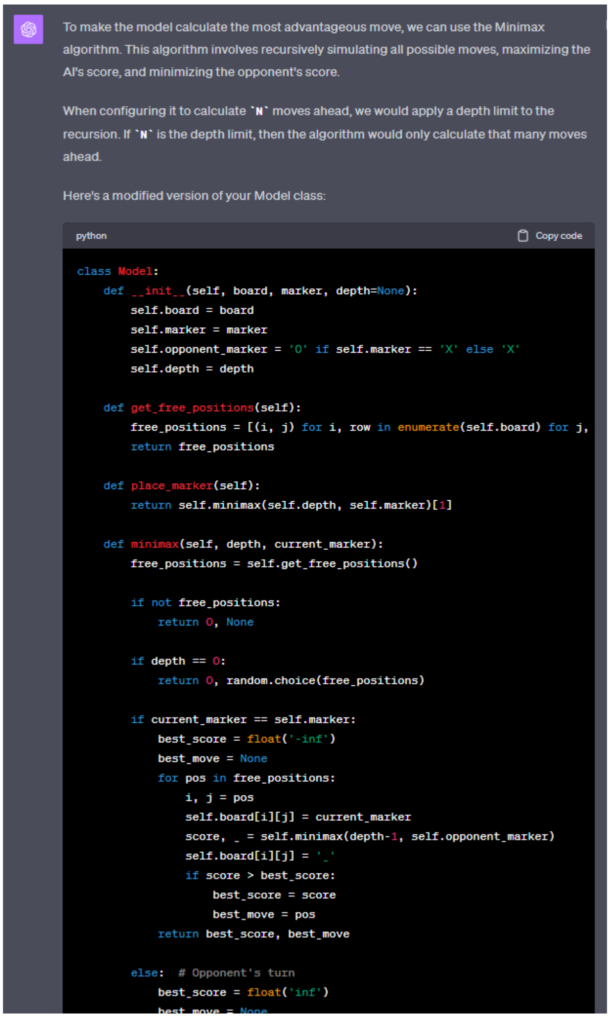
For allowing a human to play and making the board clickable, we would have to introduce a GUI. One of the popular and beginner-friendly GUI libraries for Python is `tkinter`. Here's a modification of your code to include these features using `tkinter`.

Note: This code is quite long and will change a lot of your original structure to adapt to the GUI environment. You need to make sure `tkinter` is available in your Python environment.

```python
import tkinter as tk
from tkinter import messagebox
import random

class TicTacToe:
    def __init__(self, master, size=10):
        self.master = master
        self.size = size
        self.board = [['_' for _ in range(size)] for _ in range(size)]
        self.players = ['O', 'X']
        self.current_player_index = 0
        self.buttons = [[None for _ in range(size)] for _ in range(size)]
        self.init_ui()

    def init_ui(self):
        for i in range(self.size):
            for j in range(self.size):
                self.buttons[i][j] = tk.Button(self.master, text='_', width=
                                           command=lambda i=i, j=j: self
                self.buttons[i][j].grid(row=i, column=j)

    def cell_clicked(self, row, col):
        if self.board[row][col] == '_':
            player = self.players[self.current_player_index]
            self.board[row][col] = player
            self.buttons[row][col]['text'] = player
            if self.check_win(row, col, player):
                messagebox.showinfo("Game Over", f"Player {player} wins!")
                self.master.quit()
            elif self.is_full():
                messagebox.showinfo("Game Over", "It's a draw!")
                self.master.quit()
            else:
                self.current_player_index = 1 - self.current_player_index
                if self.current_player_index == 0:  # AI's turn
                    self.ai_play()

    def ai_play(self):
        model = Model(self.board, self.players[0])
        row, col = model.place_marker()
        self.cell_clicked(row, col)

    # ... [rest of your functions from TicTacToe unchanged] ...

class Model:
    # ... [rest of your functions from Model unchanged] ...
```

Then was the time for the actual logic to emerge:

> **modify the model so that it calculates the most advantageous move. Make it configurable so that it calculates N moves ahead.**



To make the model calculate the most advantageous move, we can use the Minimax algorithm. This algorithm involves recursively simulating all possible moves, maximizing the AI's score, and minimizing the opponent's score.

When configuring it to calculate `N` moves ahead, we would apply a depth limit to the recursion. If `N` is the depth limit, then the algorithm would only calculate that many moves ahead.

Here's a modified version of your Model class:

```python
class Model:
    def __init__(self, board, marker, depth=None):
        self.board = board
        self.marker = marker
        self.opponent_marker = 'O' if self.marker == 'X' else 'X'
        self.depth = depth

    def get_free_positions(self):
        free_positions = [(i, j) for i, row in enumerate(self.board) for j,
        return free_positions

    def place_marker(self):
        return self.minimax(self.depth, self.marker)[1]

    def minimax(self, depth, current_marker):
        free_positions = self.get_free_positions()

        if not free_positions:
            return 0, None

        if depth == 0:
            return 0, random.choice(free_positions)

        if current_marker == self.marker:
            best_score = float('-inf')
            best_move = None
            for pos in free_positions:
                i, j = pos
                self.board[i][j] = current_marker
                score, _ = self.minimax(depth-1, self.opponent_marker)
                self.board[i][j] = '_'
                if score > best_score:
                    best_score = score
                    best_move = pos
            return best_score, best_move

        else:  # Opponent's turn
            best_score = float('inf')
            best_move = None
```

I also made a few additional requests to make the gameplay more varied:

> **make this so that if there are multiple equally good places to put the marker, the model chooses one in random**

To make the game play more like a human would play:

> **make it so that if it does not find any good positions, it will always place the marker next to its other marker**

As well as something to speed up the AI:

> **modify it so that it does not consider boxes that have no markers attached next to them**

After a few more basic tweaks like those, I got the AI to work well enough to be playable. Using the 'print_board' method, it displayed the current state of the board after each move:

```
_ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ _ _ _ _
_ _ _ _ _ _ O _ _ _
_ _ _ X _ O _ _ _ _
_ _ _ _ O X _ _ _ _
_ _ _ _ _ X _ _ _ _
```

```
_ _ _ _ _ X _ _ _ _
_ _ _ _ _ _ _ _ _ _
_ _ _ _ _ O _ _ _ _
```

Then started the interesting stuff.

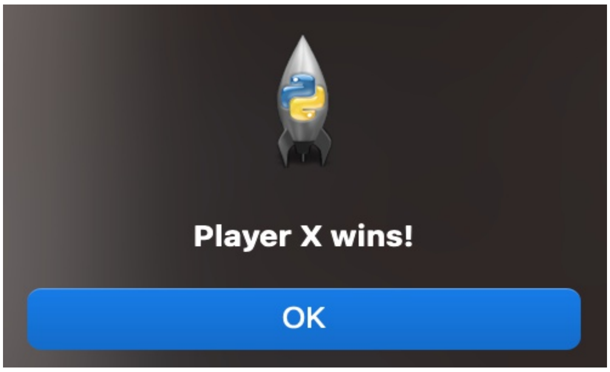
After defeating the AI, I copied a few of the previously printed board layouts from the game log, along with the AI model code, into ChatGPT. I then asked it to modify the code so the AI wouldn't be defeated as easily. I implemented the modifications ChatGPT suggested into my code and played the game again. This process was repeated several times.

_	_	_	_	_	_	_	_	_	_
_	_	_	_	_	x	_	_	_	_
_	_	_	_	_	_	x	_	_	_
_	_	_	_	x	o	_	_	x	x
_	_	_	_	o	x	_	o	_	_
_	_	_	_	o	o	o	o	_	_
_	_	_	_	x	o	_	_	_	_
_	_	_	_	o	_	x	_	_	_
_	_	_	x	_	_	_	_	_	_
_	_	_	_	_	_	_	_	_	_

The game

Every time the game lost, I would send the game log and the AI Model code to ChatGPT and simply ask it to improve the AI. With each round of improvements, I could see how the AI made better moves and became harder to beat.

After fewer than twenty iterations, this appeared:


Player X wins!

It won.

In under two hours, I managed to create a simple game of Ristinolla with an AI that's often smart enough to beat a human – without writing a single line of code myself and with only the most rudimentary consideration given to what the AI needs to do to achieve victory.

I have no doubt that each subsequent iteration will improve the AI even further, but by this point, I had already proven that my concept works. The game of Tic-tac-toe is simple, but I see no reason why this exact same approach wouldn't work in other, more complex scenarios. This process could even be automated so that after every defeat, the game log and AI code are sent to the OpenAI API, and the improved code is implemented.

Perhaps you, dear reader, might be inspired to achieve this?

You can observe the coding style produced by ChatGPT in the final code I've provided below. Use it as you see fit.

Remember: *not a single line of this code was written or edited by a human.*

```
import random

import tkinter as tk
from tkinter import messagebox
import random
import time

class TicTacToe:
    def __init__(self, master, size=10):
        self.master = master
        self.size = size
        self.board = [['_' for _ in range(size)] for _ in range(size)]
        self.players = ['O', 'X']
        self.current_player_index = 0
        self.buttons = [[None for _ in range(size)] for _ in range(size)]
        self.init_ui()

    def init_ui(self):
        for i in range(self.size):
            for j in range(self.size):
                self.buttons[i][j] = tk.Button(self.master, text='_', width=5,
height=2,
                                               command=lambda i=i, j=j:
self.cell_clicked(i, j))
                self.buttons[i][j].grid(row=i, column=j)

    def cell_clicked(self, row, col):
        if self.board[row][col] == '_':
            player = self.players[self.current_player_index]
            self.place_marker(row, col, player)
            if self.check_win(row, col, player):
                messagebox.showinfo("Game Over", f"Player {player} wins!")
                self.master.quit()
            elif self.is_full():
                messagebox.showinfo("Game Over", "It's a draw!")
                self.master.quit()
            else:
                self.current_player_index = 1 - self.current_player_index
                if self.current_player_index == 1:  # AI's turn
                    self.ai_play()

    def place_marker(self, row, col, player):
        self.board[row][col] = player
        self.buttons[row][col]['text'] = player

    def print_board(self):
        for row in self.board:
            print(' '.join(row))
        print("\n")

    def ai_play(self):
        model = Model(self.board, self.players[1], depth=5)  # Explicitly set
depth
```

```
        free_positions_next_to_X = model.get_free_positions()
        if not free_positions_next_to_X:  # No crosses yet, or no adjacent free
positions.
            free_positions = [(i, j) for i, row in enumerate(self.board) for j,
cell in enumerate(row) if cell == '_']
            row, col = random.choice(free_positions)
        else:
            #_, (row, col) = model.place_marker()  # Use minimax to get the best
move
            result = model.place_marker()
            print(result)
            row, col = result

        if row is not None and col is not None:
            self.place_marker(row, col, self.players[1])
            if self.check_win(row, col, self.players[1]):  # Note: Here you
should check for AI player's win (self.players[1])
                messagebox.showinfo("Game Over", f"Player {self.players[1]}
wins!")
                self.master.quit()
            elif self.is_full():
                messagebox.showinfo("Game Over", "It's a draw!")
                self.master.quit()
            else:
                self.current_player_index = 1 - self.current_player_index
        else:
            # No free positions, should not happen as the is_full check precedes
it
            pass

        self.print_board()


    def check_win(self, row, col, player):
        # Check horizontal, vertical, and two diagonal streaks
        for d in [(0, 1), (1, 0), (1, 1), (1, -1)]:
            count = 1
            for i in range(1, 5):
                if 0 <= row + d[0]*i < self.size and 0 <= col + d[1]*i <
self.size and self.board[row + d[0]*i][col + d[1]*i] == player:
                    count += 1
                else:
                    break

            for i in range(1, 5):
                if 0 <= row - d[0]*i < self.size and 0 <= col - d[1]*i <
self.size and self.board[row - d[0]*i][col - d[1]*i] == player:
                    count += 1
                else:
                    break

            if count >= 5:
                return True

        return False

    def is_full(self):
        for row in self.board:
            for cell in row:
                if cell == '_':
                    return False
        return True
```

```
    def play(self):
        while True:
            player = self.players[self.current_player_index]
            model = Model(self.board, player)
            row, col = model.place_marker()
            self.board[row][col] = player

            self.print_board()

            if self.check_win(row, col, player):
                print(f"Player {player} wins!")
                return

            if self.is_full():
                print("It's a draw!")
                return

            self.current_player_index = 1 - self.current_player_index

class Model:
    def __init__(self, board, marker, depth=5):
        self.board = board
        self.marker = marker
        self.opponent_marker = 'O' if self.marker == 'X' else 'X'
        self.depth = depth

    def can_make_five_in_a_row(self, pos):
        """Check if a position can potentially lead to five in a row."""
        row, col = pos
        size = len(self.board)
        directions = [(0, 1), (1, 0), (1, 1), (1, -1)]

        for d in directions:
            for start in range(-4, 1):
                count = 0
                empty_count = 0
                for i in range(start, start + 5):
                    x, y = row + d[0]*i, col + d[1]*i
                    if 0 <= x < size and 0 <= y < size:
                        if self.board[x][y] == '_':
                            empty_count += 1
                        elif self.board[x][y] == self.marker:
                            count += 1
                    else:
                        break
                if empty_count + count == 5:
                    return True
        return False

    def get_free_positions(self):
        """Get positions that are next to an 'X' or 'O' and can potentially make
five in a row."""
        free_positions = []
        for i in range(len(self.board)):
            for j in range(len(self.board[i])):
                if self.board[i][j] == '_':
                    for x, y in [(i-1, j), (i+1, j), (i, j-1), (i, j+1),
                                (i-1, j-1), (i+1, j+1), (i-1, j+1), (i+1, j-1)]:
                        if (0 <= x < len(self.board) and
                            0 <= y < len(self.board[i]) and
                            (self.board[x][y] == 'X' or self.board[x][y] ==
'O')):
                            if self.can_make_five_in_a_row((i, j)):
                                free_positions.append((i, j))
```

```
                                break
        return free_positions

    def place_marker(self, time_limit=1):
        start_time = time.time()
        depth = 1
        best_move = None
        while time.time() - start_time < time_limit:
            _, move = self.alpha_beta_minimax(depth, float('-inf'),
float('inf'), self.marker)
            if move:
                best_move = move
            depth += 1
        return best_move

    def alpha_beta_minimax(self, depth, alpha, beta, current_marker):
        if depth == 0 or self.check_win(self.marker) or
self.check_win(self.opponent_marker):
            return self.evaluate_board(), None

        free_positions = self.get_free_positions()
        free_positions.sort(key=lambda pos: -self.evaluate_position(pos,
current_marker))

        if current_marker == self.marker:
            max_score = float('-inf')
            best_move = None
            for pos in free_positions:
                i, j = pos
                self.board[i][j] = current_marker
                score, _ = self.alpha_beta_minimax(depth-1, alpha, beta,
self.opponent_marker)
                self.board[i][j] = '_'

                if score > max_score:
                    max_score = score
                    best_move = pos
                alpha = max(alpha, score)
                if beta <= alpha:
                    break
            return max_score, best_move
        else:
            min_score = float('inf')
            best_move = None
            for pos in free_positions:
                i, j = pos
                self.board[i][j] = current_marker
                score, _ = self.alpha_beta_minimax(depth-1, alpha, beta,
self.marker)
                self.board[i][j] = '_'

                if score < min_score:
                    min_score = score
                    best_move = pos
                beta = min(beta, score)
                if beta <= alpha:
                    break
            return min_score, best_move

    def evaluate_board(self):
        if self.check_win(self.marker):
            return 10**7
        elif self.check_win(self.opponent_marker):
            return -10**8
```

```
        else:
            ai_sequences = self.get_sequences(self.marker)
            opponent_sequences = self.get_sequences(self.opponent_marker)
            ai_adjacent = sum(self.adjacent_friendly_count(pos, self.marker) for
pos in self.get_free_positions())
            center_bonus = sum(self.center_distance_heuristic(pos) for pos in
self.get_free_positions())
        opponent_almost_winning =
self.count_sequences_of_length(self.opponent_marker, 4) * -10**6
        return (ai_sequences - opponent_sequences) + ai_adjacent + center_bonus
+ opponent_almost_winning

    def count_sequences_of_length(self, marker, length):
        count = 0
        size = len(self.board)
        for row in range(size):
            for col in range(size):
                if self.board[row][col] == marker:
                    for d in [(0, 1), (1, 0), (1, 1), (1, -1)]:
                        seq_count = 1
                        empty_count = 0
                        for i in range(1, 5):
                            if 0 <= row + d[0]*i < size and 0 <= col + d[1]*i <
size:
                                if self.board[row + d[0]*i][col + d[1]*i] ==
marker:
                                    seq_count += 1
                                elif self.board[row + d[0]*i][col + d[1]*i] ==
'_':

                                    empty_count += 1
                                else:
                                    break
                        if seq_count == length and empty_count == (5 - length):
                            count += 1
        return count

    def get_sequences(self, marker):
        count = 0
        size = len(self.board)
        for row in range(size):
            for col in range(size):
                if self.board[row][col] == marker:
                    for d in [(0, 1), (1, 0), (1, 1), (1, -1)]:
                        seq_count = 1
                        for i in range(1, 5):
                            if 0 <= row + d[0]*i < size and 0 <= col + d[1]*i <
size and self.board[row + d[0]*i][col + d[1]*i] == marker:
                                seq_count += 1
                            else:
                                break
                        if seq_count == 2:
                            count += 1
                        elif seq_count == 3:
                            count += 10
                        elif seq_count == 4:
                            count += 100
        return count

    def adjacent_friendly_count(self, pos, marker):
        size = len(self.board)
        row, col = pos
        count = 0
        for d in [(-1, 0), (1, 0), (0, -1), (0, 1), (-1, -1), (-1, 1), (1, -1),
```

```
(1, 1)]:
            if 0 <= row + d[0] < size and 0 <= col + d[1] < size and
self.board[row + d[0]][col + d[1]] == marker:
                count += 1
        return count

    def check_win(self, marker):
        size = len(self.board)
        for row in range(size):
            for col in range(size):
                if self.board[row][col] == marker:
                    for d in [(0, 1), (1, 0), (1, 1), (1, -1)]:
                        win = True
                        for i in range(5):
                            if not (0 <= row + d[0]*i < size and 0 <= col +
d[1]*i < size and self.board[row + d[0]*i][col + d[1]*i] == marker):
                                win = False
                                break
                        if win:
                            return True
        return False

    def evaluate_position(self, pos, marker):
        """Evaluate the value of a specific position on the board."""
        score = 0
        i, j = pos
        for d in [(-1, 0), (1, 0), (0, -1), (0, 1)]:
            sequence = [self.board[i + d[0]*k][j + d[1]*k] for k in range(-2, 3)
if 0 <= i + d[0]*k < len(self.board) and 0 <= j + d[1]*k < len(self.board[0])]
            score += self.evaluate_sequence(sequence, marker)
        return score

    def evaluate_sequence(self, sequence, marker):
        """Evaluate a sequence of five board positions."""
        score = 0
        count_marker = sequence.count(marker)
        count_empty = sequence.count('_')
        if count_marker == 4 and count_empty == 1:
            score += 10000
        elif count_marker == 3 and count_empty == 2:
            score += 1000
        elif count_marker == 2 and count_empty == 3:
            score += 100
        elif marker == self.opponent_marker and count_marker == 4 and
count_empty == 1:
            score -= 50000  # Very high penalty for opponent's potential win
        return score

    def center_distance_heuristic(self, pos):
        center = (len(self.board) // 2, len(self.board) // 2)
        distance = abs(center[0] - pos[0]) + abs(center[1] - pos[1])
        return -distance  # Closer to center is better




if __name__ == "__main__":
    root = tk.Tk()
    root.title('Tic Tac Toe')
    game = TicTacToe(root)
    root.mainloop()
```