8. **Juego de Tic-Tac-Toe (Gato):** Implementación para dos jugadores o contra la IA.

¡Claro que sí! Aquí tienes una implementación completa en Python para el juego **Tic-Tac-Toe (Gato)**, que permite jugar contra la IA utilizando el algoritmo **Minimax** para una experiencia desafiante.

?¬ﾀﾍ? Código del Juego Tic-Tac-Toe (Gato)

El código está estructurado en una clase TicTacToe para manejar la lógica del tablero y el algoritmo minimax para la IA.

Python

```
import math

class TicTacToe:
    """Clase para el juego Tic-Tac-Toe (Gato) con IA usando Minimax."""

    def __init__(self):
        # El tablero es una lista de 9 elementos, 0-8. ' ' = vacío.
        self.board = [' ' for _ in range(9)]
        self.human_player = 'O' # El jugador humano será 'O'
        self.ai_player = 'X'    # La IA será 'X'

    def print_board(self):
        """Imprime el estado actual del tablero en formato 3x3."""
        print("-------------")
        for i in range(0, 9, 3):
            # Formato: | X | O | X |
            print(f"| {self.board[i]} | {self.board[i+1]} | {self.board[i+2]} |")
            print("-------------")

    def print_board_numerado(self):
        """Imprime el tablero con números de posición para la entrada del usuario."""
        print("\n--- POSICIONES ---")
        print("-------------")
        # Números de 0 a 8
        for i in range(0, 9, 3):
            print(f"| {i} | {i+1} | {i+2} |")
            print("-------------")
        print("\n")

    def make_move(self, position, player):
        """Realiza un movimiento en la posición dada por el jugador."""
        if self.board[position] == ' ':
            self.board[position] = player
            return True
        return False

    def is_winner(self, board, player):
        """Comprueba si el jugador dado ha ganado en el tablero actual."""
        # Combinaciones ganadoras (filas, columnas, diagonales)
        win_conditions = [
            # Filas
            [0, 1, 2], [3, 4, 5], [6, 7, 8],
            # Columnas
```

```
            [0, 3, 6], [1, 4, 7], [2, 5, 8],
            # Diagonales
            [0, 4, 8], [2, 4, 6]
        ]

        for condition in win_conditions:
            if all(board[i] == player for i in condition):
                return True
        return False

    def is_board_full(self, board):
        """Comprueba si el tablero está lleno (empate)."""
        return ' ' not in board

    def get_available_moves(self, board):
        """Devuelve una lista de las posiciones vacías (movimientos disponibles)."""
        return [i for i, spot in enumerate(board) if spot == ' ']

    def minimax(self, board, depth, is_maximizing):
        """
        Algoritmo Minimax para determinar el mejor movimiento.

        :param depth: Profundidad de la búsqueda (cuántos movimientos por delante se mira).
        :param is_maximizing: True si es el turno del jugador MAX (IA), False si es MIN (Humano).
        :returns: La mejor puntuación para el estado actual.
        """
        # --- Condiciones Terminales ---
        if self.is_winner(board, self.ai_player):
            return 10 - depth  # La IA prefiere ganar rápido
        if self.is_winner(board, self.human_player):
            return depth - 10  # La IA prefiere perder tarde
        if self.is_board_full(board):
            return 0  # Empate

        # --- Lógica de la Búsqueda ---
        if is_maximizing:
            best_score = -math.inf
            player = self.ai_player

            for move in self.get_available_moves(board):
                board[move] = player
                # Llamada recursiva: ahora el humano (MIN) minimiza
                score = self.minimax(board, depth + 1, False)
                board[move] = ' ' # Deshacer el movimiento
                best_score = max(best_score, score)
            return best_score
        else: # is_minimizing
            best_score = math.inf
            player = self.human_player

            for move in self.get_available_moves(board):
                board[move] = player
                # Llamada recursiva: ahora la IA (MAX) maximiza
                score = self.minimax(board, depth + 1, True)
```

```
                board[move] = ' ' # Deshacer el movimiento
                best_score = min(best_score, score)
        return best_score

def find_best_move(self):
    """Usa Minimax para encontrar el movimiento óptimo para la IA."""
    best_score = -math.inf
    best_move = -1

    for move in self.get_available_moves(self.board):
        # 1. Probar el movimiento
        self.board[move] = self.ai_player

        # 2. Evaluar el movimiento (el humano minimiza)
        # El "False" indica que el siguiente turno es del jugador MIN (humano).
        score = self.minimax(self.board, 0, False)

        # 3. Deshacer el movimiento
        self.board[move] = ' '

        # 4. Actualizar el mejor movimiento
        if score > best_score:
            best_score = score
            best_move = move

    return best_move

def play_game(self):
    """Ciclo principal del juego."""
    print("? ¡Bienvenido al Tic-Tac-Toe contra la IA con Minimax!")
    self.print_board_numerado() # Muestra los números para elegir

    current_player = self.human_player # El humano siempre empieza

    while True:
        self.print_board()

        if self.is_winner(self.board, self.human_player):
            print("? ¡Felicidades! ¡Has ganado!")
            break
        if self.is_winner(self.board, self.ai_player):
            print("? ¡La IA ha ganado!")
            break
        if self.is_board_full(self.board):
            print("? ¡Es un empate!")
            break

        if current_player == self.human_player:
            while True:
                try:
                    move = int(input(f"Tu turno ({current_player}). Elige una posición (0-8): "))
                    if 0 <= move <= 8 and self.make_move(move, current_player):
                        break
                    else:
                        print("? Movimiento inválido. Elige un número entre 0 y 8 que esté vacío.")
```

```
                except ValueError:
                    print("? Entrada inválida. Por favor, introduce un número.")
            current_player = self.ai_player

        else: # Turno de la IA
            print("... Turno de la IA (X) ...")
            ai_move = self.find_best_move()
            self.make_move(ai_move, current_player)
            print(f"La IA se mueve a la posición: {ai_move}")
            current_player = self.human_player

if __name__ == '__main__':
    game = TicTacToe()
    game.play_game()
```

? Cómo usar el código

1. **Copia el Código:** Copia todo el bloque de código anterior en un archivo llamado, por ejemplo, gato_ia.py.
2. **Ejecuta:** Abre tu terminal o línea de comandos, navega hasta donde guardaste el archivo y ejecútalo con:
   Bash
   ```
   python gato_ia.py
   ```
3. **Juega:** El juego te mostrará el tablero numerado para que sepas qué número introducir para cada casilla. La IA (que juega con 'X') usará el algoritmo **Minimax** para jugar de manera casi perfecta, ¡haciéndolo muy difícil de ganar!

? Explicación de la Lógica (Minimax)

El algoritmo **Minimax** es el corazón de la IA y es fundamental para juegos de suma cero (donde lo que gana un jugador lo pierde el otro), como el Tic-Tac-Toe.

- **Maximizar y Minimizar:** La IA siempre intenta **Maximizar** su propia puntuación, mientras asume que el jugador humano siempre intentará **Minimizar** la puntuación de la IA.
- **Puntuación:**
- Ganar para la IA (X) = **+10**
- Perder para la IA (O gana) = **-10**
- Empate = **0**
- **Profundidad (depth):** Se utiliza para valorar las victorias y derrotas. La IA prefiere ganar en menos movimientos (gana rápido = $+10 - 0$) y, si va a perder, prefiere que sea en más movimientos (pierde lento = $-10 + \text{profundidad}$).
- **Función find_best_move:** Simula todos los movimientos posibles en el turno actual, usa minimax para evaluar qué tan bueno es cada movimiento al final del juego, y elige el que produce la mayor puntuación.
  Este video explica en más detalle cómo funciona el algoritmo Minimax para crear una IA de Tic-Tac-Toe.