Python Lab: From Tic-Tac-Toe to Minimax, then Connect Four

Part 0 — Learning goals

By the end, you should be able to:

1. Implement a complete **turn-based game loop** (human vs human, then human vs AI).
2. Represent a game state (board), validate moves, detect wins/draws.
3. Explain and implement the **Minimax algorithm**.
4. Adapt Minimax from **Tic-Tac-Toe** to a larger game (**Connect Four**) using:
    - a depth limit
    - a heuristic evaluation function
    - (optional) alpha–beta pruning

Part 1 — Build Tic-Tac-Toe (Morpion)

1. Specifications

- Board: **3×3**
- Players: `"X"` and `"O"`
- Input: the human selects a move (row/col or cell number 1–9)
- Output: print the board after each move
- End conditions:
    - win (three in a row)
    - draw (board full)

2. Required functions (design checklist)

You should implement at least these concepts (names up to you):

- `create_board()` → initial empty 3×3 board
- `display(board)` → show the board nicely
- `available_moves(board)` → list of legal moves

- `apply_move(board, move, player)` → returns updated board/state
- `winner(board)` → returns `"X"`, `"O"`, or `None`
- `is_draw(board)` → true if full and no winner
- `game_over(board)` → true if win or draw

Pseudocode: main game loop (human vs human)

```
board = create_board()
current = "X"

WHILE NOT game_over(board):
   display(board)
   move = ask_player_for_move(current)
   IF move NOT IN available_moves(board):
      print("Invalid move, try again")
      CONTINUE
   board = apply_move(board, move, current)
   current = other_player(current)

display(board)
IF winner(board) IS NOT None:
   print("Winner:", winner(board))
ELSE:
   print("Draw")
```

3. Deliverable (Part 1)

- A working Tic-Tac-Toe game (human vs human)
- Clean structure: small functions, no duplicated logic

Part 2 — What is Minimax?

1. Idea in one sentence

Minimax chooses the move that maximizes your outcome assuming the opponent plays perfectly to minimize it.

It's used for **two-player, turn-based, zero-sum** games (what I gain, you lose).

2. The game tree

From the current position, imagine all possible moves, then all opponent replies, etc. This forms a **tree**:

- nodes = game states
- edges = moves
- leaves = terminal states (win/lose/draw) or depth limit states

3. Scoring terminal states (example for Tic-Tac-Toe)

Define a function `score(state)`:

- if AI wins → `+1`
- if AI loses → `-1`
- draw → `0`

Then Minimax backs up scores:

- On AI's turn: choose the **maximum** child score
- On opponent's turn: choose the **minimum** child score

Part 3 — Implement Minimax for Tic-Tac-Toe

1. Required behavior

- Mode: human vs AI (AI plays perfectly)
- The AI must:
    - examine possible moves
    - select the move returned by Minimax as optimal

2. Pseudocode: Minimax (full search, because Tic-Tac-Toe is small)

```
FUNCTION minimax(state, player_to_move):
    IF game_over(state):
```

```
    RETURN score(state)   # +1, 0, -1 from AI perspective

  moves = available_moves(state)

  IF player_to_move == AI:
    best = -infinity
    FOR move IN moves:
      next_state = apply_move(state, move, AI)
      val = minimax(next_state, HUMAN)
      best = max(best, val)
    RETURN best
  ELSE:
    best = +infinity
    FOR move IN moves:
      next_state = apply_move(state, move, HUMAN)
      val = minimax(next_state, AI)
      best = min(best, val)
    RETURN best
```

3. Pseudocode: choosing the best move

```
FUNCTION best_move_for_ai(state):
  best_val = -infinity
  best_move = NONE

  FOR move IN available_moves(state):
    next_state = apply_move(state, move, AI)
    val = minimax(next_state, HUMAN)
    IF val > best_val:
      best_val = val
      best_move = move

  RETURN best_move
```

4. Deliverable (Part 3)

- Tic-Tac-Toe where the human can play against Minimax AI
- A short explanation (5–10 lines) in your report:
    - what Minimax assumes
    - why it’s optimal for Tic-Tac-Toe

Part 4 — Upgrade to Connect Four (Puissance 4)

Connect Four is bigger, so **full Minimax is too slow** without limits.

1. Game rules recap

- Board: **6 rows × 7 columns**
- A move: choose a **column**, the piece falls to the lowest empty row
- Win: **4 in a row** horizontally, vertically, or diagonally

2. Required state representation

- `board[r][c]` with values: empty / `"X"` / `"O"`
- `available_moves(board)` returns columns where the top cell is empty
- `apply_move(board, col, player)` drops a piece in that column

3. Win detection (must be correct)

Check all lines of length 4:

- horizontal windows
- vertical windows
- diagonal down-right
- diagonal up-right

Pseudocode: checking a 4-in-a-row window

```
FUNCTION four_in_a_row(board, player):
    FOR each cell (r, c):
        IF window of length 4 starting at (r, c) in a direction is inside board:
            IF all 4 cells belong to player:
                RETURN True
    RETURN False
```

4. Why we need “depth-limited Minimax”

The branching factor is up to 7 moves per turn, and the game can last many turns.
So we:

- stop searching after a depth `D`
- evaluate the board with a **heuristic** function

5. Evaluation function (heuristic)

Instead of only scoring terminal states, we score “how good” a position is.

Typical heuristic ideas (choose some)

- Reward having 2 or 3 in a row with open spaces to extend.
- Penalize the opponent’s threats.
- Prefer center column control (often stronger).
- Big positive/negative for immediate win/loss.

Pseudocode: evaluate using 4-cell “windows”

```
FUNCTION evaluate(board):
    IF AI has won: RETURN +infinity
    IF HUMAN has won: RETURN -infinity

    score = 0

    FOR each window of 4 cells on board:
        score += window_score(window)

    RETURN score

FUNCTION window_score(window):
    ai_count = number of AI pieces in window
    human_count = number of HUMAN pieces in window
    empty_count = number of empty cells in window

    IF ai_count == 4: RETURN +100000
    IF human_count == 4: RETURN -100000

    IF ai_count == 3 AND empty_count == 1: RETURN +100
    IF ai_count == 2 AND empty_count == 2: RETURN +10

    IF human_count == 3 AND empty_count == 1: RETURN -120   # stronger defense
    IF human_count == 2 AND empty_count == 2: RETURN -10

    RETURN 0
```

6. Depth-limited Minimax for Connect Four

```
FUNCTION minimax(state, depth, player_to_move):
    IF game_over(state) OR depth == 0:
        RETURN evaluate(state)

    moves = available_moves(state)

    IF player_to_move == AI:
        best = -infinity
        FOR move IN moves:
            next_state = apply_move(state, move, AI)
            val = minimax(next_state, depth - 1, HUMAN)
            best = max(best, val)
        RETURN best
    ELSE:
        best = +infinity
        FOR move IN moves:
            next_state = apply_move(state, move, HUMAN)
            val = minimax(next_state, depth - 1, AI)
            best = min(best, val)
        RETURN best
```

Choosing the AI move

Same as before: test each move, call minimax on resulting state, pick the maximum.

7. (Optional but recommended) Alpha–Beta pruning

Alpha–beta pruning speeds up Minimax by avoiding branches that can't change the final decision.

- `alpha`: best score the maximizer can guarantee so far
- `beta`: best score the minimizer can guarantee so far
- If `alpha >= beta`, stop exploring that branch (**prune**)

```
FUNCTION minimax_ab(state, depth, alpha, beta, player_to_move):
    IF game_over(state) OR depth == 0:
        RETURN evaluate(state)

    moves = available_moves(state)

    IF player_to_move == AI:
```

```
    value = -infinity
    FOR move IN moves:
        value = max(value, minimax_ab(next_state, depth-1, alpha, beta, HUMAN))
        alpha = max(alpha, value)
        IF alpha >= beta:
            BREAK
    RETURN value
ELSE:
    value = +infinity
    FOR move IN moves:
        value = min(value, minimax_ab(next_state, depth-1, alpha, beta, AI))
        beta = min(beta, value)
        IF alpha >= beta:
            BREAK
    RETURN value
```

Part 5 — What to submit

Deliverables

1. **Tic-Tac-Toe**
    - human vs human
    - human vs Minimax AI
2. **Connect Four**
    - human vs human (or directly human vs AI if you want)
    - human vs AI using depth-limited Minimax
    - include your evaluation heuristic
3. **Short report** (1–2 pages)
    - your state representation
    - your `evaluate()` design choices
    - what depth you used and why
    - (if done) alpha–beta pruning explanation

Suggested grading rubric

- Correctness (wins/draws/legal moves): 40%
- Code structure & readability: 20%
- Minimax correctness (Tic-Tac-Toe plays perfectly): 15%
- Connect Four AI quality (reasonable play): 15%

- Report clarity: 10%

Extensions (if you finish early)

- Add difficulty levels (depth 2 / 4 / 6).
- Add move ordering (try center columns first) to improve alpha–beta pruning.
- Add “transposition table” (memoization) for repeated states.
- Add a simple GUI (optional).