Lab 2 — Tic-Tac-Toe (n×n)

In this lab you will build an **n×n tic-tac-toe** game.

As you work through the exercises, make sure your solutions work for **any** board size n (not just 3×3), unless an exercise states otherwise.

Responsible Use of Large Language Models (LLMs)

In this lab, **you are allowed and encouraged to use LLMs responsibly** as learning tools. Think of them as **tutors, reference books, and debugging partners** — not as answer generators.

Appropriate uses

- Asking for **explanations** of Python concepts (lists, loops, functions, conditionals)
- Getting **hints** or alternative approaches when you are stuck
- Debugging errors *after* you try to reason about them yourself
- Asking an LLM to **explain your own code** back to you

Not appropriate

- Copy-pasting complete solutions without understanding them
- Submitting code you cannot explain
- Using an LLM instead of thinking through the problem first

You may be asked to explain your code or reflect briefly on how you used an LLM.

Commonly used LLMs (examples)

- **ChatGPT** — https://chat.openai.com (https://chat.openai.com)
  General-purpose reasoning, explanations, and debugging. Good for step-by-step thinking.
- **Claude** — https://claude.ai (https://claude.ai)
  Strong at reading longer code and giving structured explanations.
- **Gemini** — https://gemini.google.com (https://gemini.google.com)
  Useful for conceptual explanations and comparisons.
- **GitHub Copilot** — https://github.com/features/copilot (https://github.com/features/copilot)
  IDE-integrated suggestions. Treat suggestions as *ideas*, not answers.
- **Perplexity** — https://www.perplexity.ai (https://www.perplexity.ai)
  Search-oriented answers with sources; useful for "how does X work?" questions.

No single tool is required or preferred. What matters is **how** you use it.

Use of Large Language Models

We are explicitly going to use LLMs to help with this Lab. Choose an LLM that you will use today. Unless you are already paying for a service, please just use the free versions.

In exercise 1, we'll practice using an LLM. For subsequent exercises, the rule is that you first try to solve it yourself. If you can't do it off the top of you head, go through the lectures. Everything you need to know is there, including very useful examples. In some cases, solutions are simply minimal modifications of code from lecture. Test your solution and demonstrate that it works as explect. If a problem's solution is eluding you, practice solving problems in the same way as in class, make a plan and decompose it into smaller parts before coding. If it doesn't work correctly, iterate until it does or you are stuck.

You may use LLMs if you get stuck. If you do so, you will need to add cells to this notebook showing:

- Your original solution until you got stuck.
- The final prompt you used to solve the problem.
- The solution and an explanation of what was your mistake, lack of understanding, or

Exercise 1: Write a function that creates an **n×n matrix** (a list of lists) representing the state of a tic-tac-toe game.

Use the integers:

- `0` = empty
- `1 = "X"`
- `2 = "O"`

In [1]:
```
# Write your solution here
move_1 = "X"
move_2 = "O"
empty = 0
size=3
def create_board1(n):
    board=[]
    for i in range(size):
        board.append([empty]*size)
    return board
```

In [2]:
```
# Test your solution here
move_1 = "X"
move_2 = "O"
empty = 0
size=3
def create_board(n):
    board=[]
    for i in range(n):
        board.append([empty]*n)
    return board

create_board(size)
```

Out[2]:
```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

In [3]:
```
# (Optional) Ask an LLM for 3 different solutions here
# Then compare them to your own.
```

In [4]:
```python
# Solution 1
def create_board2(n):
    return [[0 for _ in range(n)] for _ in range(n)]
create_board(3)
```

Out[4]:
```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

In [5]:
```python
# Solution 2
def create_board3(n):
    board = []
    for i in range(n):
        row = [0] * n
        board.append(row)
    return board

create_board(3)
```

Out[5]:
```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

In [6]:
```python
# Solution 3
def create_board4(n):
    return list(map(lambda _: [0] * n, range(n)))

create_board(3)
```

Out[6]:
```
[[0, 0, 0], [0, 0, 0], [0, 0, 0]]
```

Question: Which solution most closely matches your solution? What are the main differences?

Answer: My solution is closest to Solution 2 because both use loops to build the board step by step, while the main differences are that my code uses a global variables like 'empty' and 'size', and creates shared rows, and unlike Solution 1 (list comprehension) and Solution 3 (map + lambda), my approach is less compact.

Exercise 2: Write a function that takes two integers `n` and `m` and **draws** an `n` by `m` game board.

For example, the following is a 3×3 board:

```
 --- --- --- 
|   |   |   | 
 --- --- ---  
|   |   |   | 
 --- --- ---  
|   |   |   | 
 --- --- --- 
```

In [7]:
```
# Write your solution here
def draw_board1(n, m):
    for i in range(n):
        print("---" * m)
        print("|   " * m + "|")
    print("---" * m)
```

In [8]:
```
# Test your solution here
def draw_board(n, m):
    for i in range(n):
        print(" ---" * m)
        print("|   " * m + "|")
    print(" ---" * m)

draw_board(3,3)
```

```
 --- --- ---
|   |   |   |
 --- --- ---
|   |   |   |
 --- --- ---
|   |   |   |
 --- --- ---
```

Exercise 3: Modify Exercise 2 so that it takes a matrix in the format from Exercise 1 and draws a tic-tac-toe board with `"X"` s and `"O"` s.

In [9]:
```
# Write your solution here
def draw_tic_tac_toe1(board):
    n = len(board)

    for i in range(n):
        print(" ---" * n)

        for j in range(n):
            if board[i][j] == 1:
                print("| X ", end="")
            elif board[i][j] == 2:
                print("| O ", end="")
            else:
                print("|   ", end="")
            print("|")

    print(" ---" * n)
```

In [10]:
```
# Test your solution here
def draw_tic_tac_toe(board):
    n = len(board)

    for i in range(n):
        print(" ---" * n)

        for j in range(n):
            if board[i][j] == 1:
                print("| X ", end="")
            elif board[i][j] == 2:
                print("| O ", end="")
            else:
                print("|   ", end="")
        print("|")

    print(" ---" * n)


board1 = [[1, 0, 2],[0, 1, 0],[2, 0, 1]]


draw_tic_tac_toe(board1)
```

```
 --- --- ---
| X |   | O |
 --- --- ---
|   | X |   |
 --- --- ---
| O |   | X |
 --- --- ---
```

Exercise 4: Write a function that takes an `n×n` matrix representing a tic-tac-toe game and returns one of the following values:

- `-1` if the game is **incomplete** (still empty spaces and no winner)
- `0` if the game is a **draw**
- `1` if **player 1** (`"X"`) has won
- `2` if **player 2** (`"O"`) has won

Here are some example inputs you can use to test your code:

In [11]:
```python
# Write your solution here
def check_game1(board):
    n = len(board)

    # rows
    for i in range(n):
        if board[i][0] != 0:
            win = True
            for j in range(n):
                if board[i][j] != board[i][0]:
                    win = False
            if win:
                return board[i][0]

    # columns
    for j in range(n):
        if board[0][j] != 0:
            win = True
            for i in range(n):
                if board[i][j] != board[0][j]:
                    win = False
            if win:
                return board[0][j]

    # 1st diagonal
    if board[0][0] != 0:
        win = True
        for i in range(n):
            if board[i][i] != board[0][0]:
                win = False
        if win:
            return board[0][0]

    # 2nd diagonal
    if board[0][n-1] != 0:
        win = True
        for i in range(n):
            if board[i][n-1-i] != board[0][n-1]:
                win = False
        if win:
            return board[0][n-1]

    return 0
```

In [12]:
```
# Test your solution here
def check_game(board):
    n = len(board)

    # rows
    for i in range(n):
        if board[i][0] != 0:
            win = True
            for j in range(n):
                if board[i][j] != board[i][0]:
                    win = False
            if win:
                return board[i][0]

    # columns
    for j in range(n):
        if board[0][j] != 0:
            win = True
            for i in range(n):
                if board[i][j] != board[0][j]:
                    win = False
            if win:
                return board[0][j]

    # 1st diagonal
    if board[0][0] != 0:
        win = True
        for i in range(n):
            if board[i][i] != board[0][0]:
                win = False
        if win:
            return board[0][0]

    # 2nd diagonal
    if board[0][n-1] != 0:
        win = True
        for i in range(n):
            if board[i][n-1-i] != board[0][n-1]:
                win = False
        if win:
            return board[0][n-1]

    # empty spaces
    for i in range(n):
        for j in range(n):
            if board[i][j] == 0:
                return -1

    return 0
```

In [13]:
```
winner_is_2 = [[2, 2, 0],
    [2, 1, 0],
    [2, 1, 1]]

winner_is_1 = [[1, 2, 0],
    [2, 1, 0],
    [2, 1, 1]]

winner_is_also_1 = [[0, 1, 0],
    [2, 1, 0],
    [2, 1, 1]]

no_winner = [[1, 2, 0],
    [2, 1, 0],
    [2, 1, 2]]

also_no_winner = [[1, 2, 0],
    [2, 1, 0],
    [2, 1, 0]]
```

In [14]:
```
check_game(winner_is_2)
```

Out[14]:
```
2
```

In [15]:
```
check_game(winner_is_1)
```

Out[15]:
```
1
```

In [16]:
```
check_game(winner_is_also_1)
```

Out[16]:
```
1
```

In [17]:
```
check_game(no_winner)
```

Out[17]:
```
-1
```

In [18]:
```
check_game(also_no_winner)
```

Out[18]:
```
-1
```

Exercise 5: Write a function that takes a game board, a player number, and `(row, col)` coordinates and places the correct mark (`"X"` or `"O"`) in that location.

Requirements:

- Only allow placing a mark in a previously empty location.
- Return `True` if the move was successful, and `False` otherwise.

In [19]:
```
# Write your solution here
def make_move1(board, player, location):
    if board[location[0]][location[1]] == 0:
        board[location[0]][location[1]]  = player
        return True
    else:
        return False
```

In [20]:
```
# Test your solution here
def make_move(board, player, location):
    if board[location[0]][location[1]] == 0:
        board[location[0]][location[1]]  = player
        return True
    else:
        return False



board1 = [[1, 0, 2],[0, 1, 0],[2, 0, 1]]

print(make_move(board1, 1, (1, 1)))
print(make_move(board1, 2, (0, 1)))
```

```
False
True
```

In [ ]:

Exercise 6: Modify Exercise 3 to show **row and column labels** so that players can specify locations like `"A2"` or `"C1"`.

In [21]:
```
# Write your solution here

def draw_tic_tac_toe_labeled1(board):
    size = len(board)
    row_names = []
    column_names=[]
    for i in range(size):
        row_names.append(chr(65 + i))
        column_names.append(i)
    moves = {0: " ", 1: "X", 2: "O"}
    print(" ",end=" ")
    for j in range(size):
        print(column_names[j],end=" ")
    print()

    for i in range(size):
        print(row_names[i],end=" ")
        for j in range(size):
            print(moves[board[i][j]],end=" ")
        print()
```

In [22]:
```
# Test your solution here
def draw_tic_tac_toe_labeled(board):
    size = len(board)
    row_names = []
    column_names=[]
    for i in range(size):
        row_names.append(chr(65 + i))
        column_names.append(i+1)

    moves = {0: " ", 1: "X", 2: "O"}
    print(" ",end=" ")
    for j in range(size):
        print(column_names[j],end=" ")
    print()

    for i in range(size):
        print(row_names[i],end=" ")
        for j in range(size):
            print(moves[board[i][j]],end=" ")
        print()

board1 = [[1, 0, 2],[0, 1, 0],[2, 0, 1]]

draw_tic_tac_toe_labeled(board1)
```

```
  1 2 3
A X   O
B   X
C O   X
```

Exercise 7: Write a function that takes a board, a player number, and a location string (as in Exercise 6), then uses your function from Exercise 5 to update the board.

In [23]:
```
# Write your solution here
def make_move_by_label1(board, player, location):
    row = ord(location[0].upper()) - 65
    col = int(location[1]) - 1
    if board[row][col] == 0:
        board[row][col] = player
        return True
    return False
```

In [24]:
```
# Test your solution here
def make_move_by_label(board, player, location):
    row = ord(location[0].upper()) - 65
    col = int(location[1]) - 1
    if board[row][col] == 0:
        board[row][col] = player
        return True
    return False

board1 = [[1, 0, 2],[0, 1, 0],[2, 0, 1]]

make_move_by_label(board1,2,'A1')
```

Out[24]:
```
False
```

In [25]:
```
make_move_by_label(board1,1,'A2')
```

Out[25]:
```
True
```

In [26]:
```
print(board1)
```

```
[[1, 1, 2], [0, 1, 0], [2, 0, 1]]
```

Exercise 8: Write a function that is called with a board and player number, takes input from the player using Python's `input()` , and modifies the board using your function from Exercise 7.

Keep asking for input until the player enters a valid location that results in a valid move.

In [27]:
```
# Write your solution here
def player_move1(board, player):
    valid_move = True
    while valid_move:
        location = input("Enter your move: ")
        valid_move = make_move_by_label(board, player, location)
        if valid_move!=True:
            print("Invalid move, try again.")
```

In [28]:
```
# Test your solution here
def player_move(board, player):
    valid_move = True
    while valid_move:
        location = input("Enter your move: ")
        valid_move = make_move_by_label(board, player, location)

        if valid_move!=True:
            print("Invalid move, try again.")
        else:
            break
```

In [29]:
```
player_move(board1, 1)
```

```
Enter your move: A1
Invalid move, try again.
```

In [30]:
```
player_move(board1, 2)
```

```
Enter your move: C2
```

In [31]:
```
print(board1)
```

```
[[1, 1, 2], [0, 1, 0], [2, 2, 1]]
```

Exercise 9: Use all of the previous exercises to implement a full tic-tac-toe game:

- draw the board,
- repeatedly ask two players for a location,
- apply valid moves,
- check the game status until a player wins or the game is a draw.

In [32]:
```python
# Write your solution here
def play_game1():
    size=3
    my_board=create_board(size)
    draw_tic_tac_toe_labeled(my_board)

    valid_move = True
    results=-1
    while results==-1:
        player= int(input('Are you player 1 or 2? :'))
        location = input('Enter your move:')
        valid_move = make_move_by_label(my_board, player, location)

        if valid_move!=True:
            print("Invalid move, try again.")
        else:
            draw_tic_tac_toe_labeled(my_board)
            results = check_game(my_board)
```

In [35]:
```python
# Test your solution here
def play_game(size):
    my_board=create_board(size)
    draw_tic_tac_toe_labeled(my_board)

    valid_move = True
    results=-1
    player=1
    while results==-1:
        location = input('Enter your move:')
        valid_move = make_move_by_label(my_board, player, location)

        if valid_move!=True:
            print('Invalid move, try again.')
        else:
            draw_tic_tac_toe_labeled(my_board)
            results = check_game(my_board)
            if player==1:
                  player=2
            else:
                  player=1
    if results == 0:
        print('We have a draw!')
    elif results ==1:
        print('Player 1 "X" has won!')
    elif results ==2:
        print('Player 2 "O" has won!')
```

In [36]:
```
play_game(3)
```

```
  1 2 3
A
B
C
Enter your move:A1
  1 2 3
A X
B
C
Enter your move:A1
Invalid move, try again.
Enter your move:B1
  1 2 3
A X
B O
C
Enter your move:A2
  1 2 3
A X X
B O
C
Enter your move:B3
  1 2 3
A X X
B O   O
C
Enter your move:A3
  1 2 3
A X X X
B O   O
C
Player 1 "X" has won!
```

Exercise 10: Test that your game works for **5×5** tic-tac-toe.

In [37]:
```
# Test your solution here
play_game(5)
```

```
  1 2 3 4 5
A
B
C
D
E
Enter your move:A1
  1 2 3 4 5
A X
B
C
D
E
Enter your move:B3
  1 2 3 4 5
A X
B     O
C
D
E
Enter your move:B2
  1 2 3 4 5
A X
B   X O
C
D
E
Enter your move:E1
  1 2 3 4 5
A X
B   X O
C
D
E O
Enter your move:C3
  1 2 3 4 5
A X
B   X O
C     X
D
E O
Enter your move:D5
  1 2 3 4 5
A X
B   X O
C     X
D         O
E O
Enter your move:D4
  1 2 3 4 5
A X
B   X O
C     X
D       X O
E O
Enter your move:C2
  1 2 3 4 5
A X
B   X O
C   O X
D       X O
```

```
E O
Enter your move:E5
  1 2 3 4 5
A X
B   X O
C   O X
D       X O
E O       X
Player 1 "X" has won!
```

Exercise 11: Develop a version of the game where one player is the computer.

Note: you do **not** need an extensive search for the best move. For example, you can have the computer:

- block obvious losses
- otherwise try to create a winning row/column/diagonal

In [ ]:
```
# Write your solution here
```

In [ ]:
```
# Test your solution here
```

Exercise 12: Develop a version of the game where one player is the computer. This time, write a computer player using exhaustive search with a max depth parameter, similar to lecture.

In [ ]:
```
# Write your solution here
```

In [ ]:
```
# Test your solution here
```

Exercise 13: Make the 2 computer players play each-other for 10 games on a 3x3, then 4x4, then 5x5 grid. Set the max depth so that the games only take seconds. Measure the "smarter" player's win rate for each grid.

In [ ]:
```
# Write your solution here
```

In [ ]:
```
# Test your solution here
```

Lab Summary

In this lab you practiced:

- Representing a game board using nested lists
- Writing small, focused functions
- Using conditionals and loops to analyze program state
- Thinking carefully about assumptions and edge cases
- Using LLMs **responsibly** as learning tools rather than answer generators

The goal is not just to make the program work, but to understand *why* it works. That understanding is what allows you to use tools — including AI — effectively.