In this lab, you will do a warm-up execrsice, then implement a way to store a memory in a Hopfield Network using Hebbian learning, a historically important algorithm that just needs a for-loop, and then write a program that lets you play Tic-Tac-Toe against the computer.
See `https://en.wikipedia.org/wiki/Tic-tac-toe` for details on the game. In this lab, you should try to write functions that are as concise as possible. In many cases, this means using loops even if a more tedious solution that doesn't use loops is possible.

Bit first, three (mandatory) warm-up questions

Problem 1.

Consider the following list:

```
L = [["CIV", 92],
     ["180", 98],
     ["103", 99],
     ["194", 95]]
```

Write code that prints the element `99` from this list.

Problem 2.

Now, write a function `get_nums(L))` that takes in a list in a format similar to `L`, and returns a list like `[92, 98, 99, 95]`. Hint: look are the code for the matrix-vector product from lecture, and understand how the result `res` was built up.

Problem 3.

Write a function `lookup(L, num)` that takes in a list like `L` and an argument like `99` and returns the corresponding value (like `"103"`). Return the first value if there are multiple matches. Return `None` if there are no matches.

Problem 4.

Note: this is not harder than any other exercise! Just reason through things line-by-line.

In this problem, you will be working with the Hopfield Network from last week.

First, we will modify the code to compute the energy for specific weights and x's:

```
def E(x0, x1, x2, w01, w02, w12):
    term1 = x0 * x1 * w01
    term2 = x0 * x2 * w02
    term3 = x1 * x2 * w12

    return -(term1 + term2 + term3)


def print_all_energies(w01, w02, w12):
    for x0 in [-1, 1]:
```

```
        for x1 in [-1, 1]:
            for x2 in [-1, 1]:
                print("x: (", x0, x1, x2, ")", "E:", E(x0, x1, x2, w01, w02, w12))
```

This could be used with the weights from last week as follows:

```
if __name__ == '__main__':
    w01 = 2
    w02 = -1
    w12 = 1
    print_all_energies(w01, w02, w12)
```

The output is as follows:

```
    x: ( -1 -1 -1 ) E: -2
    x: ( -1 -1 1 ) E: -2
    x: ( -1 1 -1 ) E: 4
    x: ( -1 1 1 ) E: 0
    x: ( 1 -1 -1 ) E: 0
    x: ( 1 -1 1 ) E: 4
    x: ( 1 1 -1 ) E: -2
    x: ( 1 1 1 ) E: -2
```

Recall that the minima of the energy function are "memories". This means that the network "remembers" the values $(-1,-1,-1),(-1,-1,1),(1,1,-1),(1,1,1)$.

We would now like to "store" a new memory in the network. We will do it by adjusting the weights w a little bit.

The strategy for storing a new memory (x_0, x_1, x_2) is as follows:

1. If $x_0 * x_1 > 0$, increasing w_{01} will decrease the energy of the network at (x_0, x_1, x_2), so we will increase w_{01} by 0.1. Otherwise, we will decrease w_{01} by 0.1.
2. If $x_0 * x_2 > 0$, increasing w_{02} will decrease the energy of the network (x_0, x_1, x_2), so we will increase w_{02} by 0.1. Otherwise, we will decrease w_{02} by 0.1.
3. If $x_1 * x_2 > 0$, increasing w_{12} will decrease the energy of the network at (x_0, x_1, x_2), so we will increase w_{12} by 0.1. Otherwise, we will decrease w_{12} by 0.1.

Part (a)

Explain why the claim "If $x_0 * x_1 > 0$, increasing w_{01} will decrease the energy of the network at (x_0, x_1, x_2)" makes sense.

Part (b)

Write three if-statements that execute the strategy above, and verify that the effect of that is to decrease the energy of the network at $(-1, 1, 1)$ when $x = (-1, 1, 1)$.

Note: all we are saying is "write three if-statements and use `print_all_energies()` to verify that the energy at $(-1, 1, 1)$ is decreased.

Part (c)

Write a for-loop that repeadly adjust the weights as above, and uses `print_all_energies()`

Suggestion: print `"=============="` in between iterations to make the output more readable.

Part (d)

Use the for-loop to verify that after about 4 iteration, the new memory is stored.

Part (e)

Put the for-loop in a function. The function should take in the weights and the memory to store, and should return the weights (as a list with three elements) after the memory is stored.

Part (f)

Use the function to store the memory $(1, -1, 1)$ in the network. Note that the function returns a list. There are several ways to extract the answers from a list. Use tools from the course (i.e., 0-th element of `L` is `L[0]`) to update `w01, w02, w12`.

Problem 5.

Your first task is to enable two users to play against each other. Download `ttt.py`, and understand the functions for creating an empty board and to print the board and the legend. Run `ttt.py` and observe how `print_board_and_legend(board)` prints the list of lists `board`, which represents the board.

The goal is to be able to produce a game that goes, for example, as follows:

```
   |   |        1 | 2 | 3
---+---+---    ---+---+---
   |   |        4 | 5 | 6
---+---+---    ---+---+---
   |   |        7 | 8 | 9
Enter your move: 5


   |   |        1 | 2 | 3
---+---+---    ---+---+---
   | X |        4 | 5 | 6
---+---+---    ---+---+---
   |   |        7 | 8 | 9
Enter your move: 1


 O |   |        1 | 2 | 3
---+---+---    ---+---+---
   | X |        4 | 5 | 6
---+---+---    ---+---+---
   |   |        7 | 8 | 9
Enter your move: 3
```

Part (a)

Write a function that takes in an integer `square_num` between `1` and `9`, and returns a list `coord` such that `board[coord[0]][coord[1]] = "X"` would put an `"X"` in square `square_num` (an integer from 1 to 9). Hint: the row number is `((square_num - 1)  // 3)`.

Part (b)

Write a function `put_in_board(board, mark, square_num)` that modifies the contents of `board` such that the string `mark` (`"X"` or `"O"`) is put in the coordinates in `board` that correspond to `square_num`.

Part (c)

Write a loop that asks for the user to alternately enter coordinates for `"X"`s and `"O"`s such that two users can play against each other as shown in the example above.

Here is a piece of code that counts how many times the user failed to input `"hi"` when repeatedly trying.

```
count = 0
input_str = ""
while input_str != "hi":
  input_str = input("Please say hi: ")
  count += 1

print(f"The user did not say hi {count} times")
```

Problem 6.

The goal now is to write a simple function that would have the computer play against the user.

Part (a)

Write a function with the signature `get_free_squares(board)` which creates and returns a new list which contains a list of the coordinates of the free squares in the board. For example, if the board is represented as follows

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

, the function should return `[[0, 1], [1, 0], [1, 2], [2, 1], [2, 2]]`.

Part (b)

Now write a function `make_random_move(board, mark)` that finds a random free square in `board`, and puts the string `mark` in the free square. Hint: you can print a random number between `0` and `n-1` as follows:

```
import random
print(int(n * random.random()))
```

Part (c)

Now use `make_random_move()` in order to have the computer play against the user.

Problem 7.

Now, the goal is to automatically figure out if the game is over. The game is over if there is a line of 3 `"X"`s or a line of 3 `"O"`s.

Part (a)

Write a function with the signature `is_row_all_marks(board, row_i, mark)` which returns `True` iff the row with index `row_i` in `board` contains 3 marks equal to `mark`.

Part (b)

Write a function with the signature `is_col_all_marks(board, col_i, mark)` which returns `True` iff the column with index `row_i` in `board` contains 3 marks equal to `mark`.

Part (c)

Using the functions above, and also checking the diagonals, write a function with the signature `is_win(board, mark)` that returns `True` iff the mark `mark` won on the board `board` (i.e., there is a line of 3 `mark`s somewhere in `board`).

Part (d)

Incorporate `is_win()` into the program such that the game stops when either the user or the computer win, and prints the result of the game.

Problem 8.

Your job now is to improve the the function that makes the computer's move.

Part (a)

Write a function which tries to put the computer's mark in every free square on the board, and checks whether `is_win()` returns `True` for the new board, returns if it does, and removes the mark and tries to place it in another square otherwise. If there is no square such that putting a mark in it leads to an immediate win, the function should put `mark` in a random free square.

Part (b)

Improve the algorithm that plays for the computer as much as you can.