Assignment 4 – Tic-tac-toe, Matrices

Goal: Represent a tic-tac-toe grid with a matrix.

Background

This assignment is about Tic-tac-toe. A turn-based game where two players place markers on a rectangular board with 9 cells. One player places *X*s , and the other places *O*s. The first player with three markers in a row, colum, or diagonal, wins. Below is an example of a board where the *X* player won.

X	*O*	
X	*X*	*O*
O		*X*

Specification of the tic-tac-toe board

In this assignment, we will implement some functions to represent a tic-tac-toe board. We want to be able to create a new board, check if a cell is free, place a marker, get a marker, check if the board is full, and check if a player has won.

Before we start, we will make a *specification*: an English description of how the board should work, and what we want be able to do with it.

We want to be able to:

- create a new board. The board should have a size of 3×3.
- get what a specific cell on the board contains. The result should be either "X", "O", or "-" for an empty cell.
- check if a cell is empty or not. The result should be true or false.
- try to place a marker at a specific position. However, we cannot put any marker anywhere. If there is already a marker where we want to put ours, the operation should fail and return false. If the operation succeeds, it should return true.
- check if a board is full (we cannot place more markers). The operation should return true or false
- check if a player has won, should also return true or false.

In Tic-tac-toe, there is the additional rule that each player should play in turn, but we assume this is outside of the work of the board.

Making a board

How do we make a board, that is, how do we translate the concept of a two dimensional board in Python? One approach is to think that a board is composed of rows and columns. So we could represent it as a list of rows, each row contains several cells. If a row is a list of cells, and a board a list of rows, we have a list of lists, also called a matrix.

Similarly, for the markers, we will represent them with the characters "X" and "O". When a cell is free, we represent it with a "-", and if there is a marker in it, then we use the marker, "X" or "O". For example, we can represent the example listed earlier on the page with the matrix:

```
[
    ['X', 'O', '-'],
    ['X', 'X', 'O'],
    ['O', '-', 'X']
]
```

To make things more convenient, we want to visualize a board in a format that doesn't have all these `[]` and `,`. To do that, we will implement the function `print_board()`, which takes a board as an input, and prints with a formatting like the one below:

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

Tasks

1. Create a new file *board.py* in the directory *lab04*.

2. Create the function `new_board`, which takes no parameter, and create a 3×3 matrix, with only empty cells.

3. Implement the function `get` which takes three arguments, the board, the row number and the column number, and returns the value at that position in the provided board.

4. Implement the function `is_empty`, takes a row and column number as a parameter, and returns `True` if the cell is empty (contains `'-'`), `False` otherwise. Assume that we index cells from zero.

5. Implement the function `print_board`, which prints the board. Remember from assignment 1 that the function `print` can be configured. Usually, when we use the function without any parameters, it adds a new line when it's done printing. You can change that by passing the parameter `end`, so that it adds what you want when it's done printing. The default value of the parameter is a new line character. Here is an example on how you can use this:

   ```
   print('A', end=' ')  # Writes ' ' after 'A', but doesn't move to a new line
   print('B')           # Writes 'B', and goes to the next line :
   # Prints: A B
   ```

6. Implement the function `place`, which takes a board, a marker, a row and a column, and places the marker at the right position on board. Notice that if you change the board that you passed as an argument, it will change the original board.

   For example, the following code:

   ```
   b = new_board()
   place(b, "X", 0, 0)
   print_board(b)
   ```

   Prints:

   ```
   X - -
   - - -
   - - -
   ```

Which indicates that the function *modified* the board `b`. This does not happen if we pass an integer or a string, to the function. When the argument is a list, or a dictionary, it is *passed by reference*, which means that the function can modify the object and the changes will be visible from outside.

Note: When we pass an argument to a function, it can be handled in two different ways: Passed by *value* or by *reference*. When it is passed by value (like integers are), the argument is copied before it is passed, so if you change the value of the parameter inside the function, the change will not be visible from outside the function. When it is passed by reference (as lists are), the modifications that you make on the argument will change it, and the changes will be visible from the outside of the function. This is what is happening here.

7. Implement the function `is_full`, which returns true if the board is full (no empty cells), and false otherwise. Test your functions in the REPL.

   If your conditions in if-statements become too big, you can wrap them in parentheses and split them across several times.

```
if (...  first part of conditional ...
       ...  second part of conditional ...):
    ...  statements that execute if the condition is true ...
```

8. Finally, we will implement the function `is_winner`, which is the most complicated. The function takes a board as input, and a marker, and returns true if the player with that marker has won. To check that, we need to check if the player has positioned the markers on three cell in a row, in a column, or in a diagonal.

   This task is a bit more complicated, so take it step by step. To implement it, remember to test your code. For example, you can implement checking that the player has a full row only, and then test it with several boards, to see if it returns true when it should. Then move to adding checks for columns, repeat the process, and lastly, check diagonals.

9. When the functions are implemented, we can test them together with already implemented code, which will create a graphical interface to play tic-tac-toe. The code that generates the interface will have no problem calling your code if you used the names we provided for the functions. This is usually what a specification is for: an agreement between programmers writing different parts of software, so that they fit together nicely.

   The finished interface is in the file *window_tk.py*, which imports and creates an object of type `TicTacToeBoard`. This can be see at the end of the file *window_tk.py*. Run this file and test that it works. A graphical interface should appear, allowing you to play Tic-Tac-Toe by clicking on squares. The interface uses the `tkinter` library, which is normally included in every installation of Python.

10. When you run the file *window_tk.py*, the code in the file *board.py* also executes. So the functions that you wrote are read as well. Sometimes, when we create a module, we want to have *test code*, which we only want to execute while we're designing the module, to check that everything works. When we launch the full program, we don't want test code to run, though.

    To do that, we can add an if-statement at the end of *board.py*.

```
if __name__ == '__main__':
    ...  Test code for the functions, for example ...
    b = new_board()
```

```
place(b, 'X', 0, 0)
print_board(b)
```

The `__name__` variable is a variable that Python sets automatically. When the module is loaded from another module, it will be equal to the name of that module (in our example `board`, because the file is called *board.py*).

However, if you run the file `board.py` in full, then it is not imported from somewhere else, and the value of `__name__` is `__main__`. In which case, what we put in the if-statement is executed. Usually, it's testing code.

Add some tests to the end of the *board.py* file.

11. The file *window_tk.py* contains what is called a *class*. Classes are a bit outside of the scope of this course, but put briefly, a class is a way to define our own *objects*. Objects are "boxes" that can contain variables and functions. Variables that are part of a class are called *attributes*, while functions which are part of a class are called *methods*.

    We have seen methods before: When we append to a list, we call a method, which will change the list.

    In our implementation of the board, we passed a parameter called `board` to all the functions. Because methods usually modify the variables which are also part of the class, they have a parameter that plays a similar role, called `self`.

    The class in this file is called `TicTacToeApp`. It has methods to initialize the graphical components (`init_components`), catch when the user clicks on a box (`clicked`), or the "Restart!" button (`restart_clicked`) and update the interface (`update`).

    The class has the following attributes: The board (`_board`), which player will play next turn (`_current_marker`) and whether the game is still running (`_is_running`).

    When buttons are clicked on the interface, the class takes care of running the right functions, that will update the board. This type of programming is called *event-driven programming*. For example, when the user clicks on a button, the method `clicked` is called. When the user presses on the button "Restart", the method `restart_clicked` is called, which creates a new board.

    Take a look in the *window_tk.py*. The window class calls functions to update the board, when buttons are pressed. Try to understand roughly what happens in the methods `clicked`, `restart_clicked` and `update` methods.

    Creating graphical interfaces is not included in the course and often requires reading a fair amount of documentation. There are several different libraries in Python for creating interfaces, and tkinter is one of them. Other libraries, such as Qt, have a somewhat more modern appearance but are somewhat more complicated to install.