Danny Rehl, Programming I - Python

Exercise sheet 05

Deadline: Together with the last exercise sheet of the semester

---

You will find a template for this exercise sheet to download from Moodle.

The functions you complete in the upper part of the template count towards the evaluation of your submission. Only write your solutions here, no other commands outside of the functions.

In the lower part you will find an `if name == " main "` section, which already contains some example calls to the functions. Here you can add to or replace the commands as required to test your functions.

Note: Despite all tests, the uploaded .py file must of course still remain executable. For example, a `SyntaxError` in the `if name == " main "` part will always lead to a Pytest evaluation of 0 %.

Bonus exercise sheet

In the course of the tasks below, you will write your own implementation of Four Wins. Even though this may sound very time-consuming at first, don't be discouraged: Breaking it down into sub-problems significantly reduces the perceived difficulty of the project. In the end, you will only need code that you have already used or will use in a similar way on the previous or upcoming exercise sheets.

The submission deadline for this exercise sheet is the same as the submission deadline for the last exercise sheet of the semester. You can therefore work on a new exercise throughout the semester and upload the latest version to Tutron.

As some of the tasks build on each other, it is important that you complete the tasks in the order given.

In addition to the usual sections, this time the template also contains the following extensive part:

```
# Default code (do not change anything here)
```

You should not change anything in the code in this section, as stated in the comment. However, this does not mean that you should or can ignore the section. It will be worthwhile looking through the code carefully to understand how to work through the tasks.

Task 1 (1 bonus point)

We first need a representation of the game data so that we can work on it with our code. The game board consists of a grid with 6 rows and 7 columns. It can therefore be represented very clearly by a 6x7 matrix. We save this matrix in the code as a multidimensional list `grid`.

Complete the function `empty_grid`, which has a multidimensional list as its return value. The elements of the first dimension are the 6 rows of the game board, each row (i.e. the second dimension) in turn contains 7 elements of type `int` with value `0`.

The list return value should therefore look like this:

```
[[0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0],
[0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0], [0, 0, 0, 0, 0, 0, 0]]
```

Make sure that you select the correct number of rows and columns. With `grid[0][0]` you should be able to access the value of the top leftmost field, with `grid[5][6]` the value of the bottom rightmost field. (As the index starts at 0, this is the 6th row, 7th column).

If you swap the number of rows and columns by mistake, `grid[5][6]` will result in a `IndexError`.

Task 2 (1 bonus point)

Complete the function `print_grid`. It should print the game data stored in `grid`. The value `0` should be represented by a space, player 1 with value `1` by an `X` and player 2 with value `2` by an `O`. Use `|`, `+` and `-` to draw a border around the fields. The individual columns have a spacing of 1 space. Under the bottom The columns are labeled with the numbers `1` to `7` along the edge of the playing field.

Take a look at the preliminary output of the function and expand it bit by bit until it fulfills all the required demands.

Note: You use the `end` parameter of the `print` function at this point:

```
print(text, end="")
```

This outputs the variable `text` without automatically appending a line break `\n` afterwards. If necessary, take another look at the lecture slides or the Python documentation for print for more information.

To test your function, you can use the examples `EXAMPLE_GRID_1` contained in the template, `EXAMPLE_GRID_2` and `EXAMPLE_GRID_3`. You should receive the following output:

```
|                     |
|                     |
|                     |
|                     |
|                     |
|                     |
+---------------------+
  1 2 3 4 5 6 7
```

```
|       X             |
|       O             |
|       X             |
|O      O             |
|X      X X       O   |
|X O O X       O X    |
+---------------------+
  1 2 3 4 5 6 7
```

```
|O O O X X O O O O O |
|O X O O X X X X     |
|X O X O X O O O     |
|O X O X X X X O     |
|  X O O O X O X O   |
|X X X O O X X X X   |
+---------------------+
  1 2 3 4 5 6 7
```

Task 3 (1 bonus point)

The values of the matrix, which represents the game board, are stored in rows in the `grid`. A function is required to obtain a list with the values of a column of the matrix.

Complete the `get_column` function so that the column of the matrix specified with `column_number` is returned. (Values from top to bottom.)

Example: For `get_column(0, EXAMPLE_GRID_3)` we expect we expect the return value `[2, 2, 1, 2, 1, 1]`. (See also the output of `EXAMPLE_GRID_3` above.)

Task 4 (1 bonus point)

6 values fit into one column. When a player has chosen a column for his "disk", he must check whether there is still a free space in this column and, if so, in which position. (After all, the disks must be arranged from bottom to top).

Complete the `free_space` function so that the column specified with `column_number` is checked for a free field. If no field is free, `None` is returned. If a field is still free, the index with which this row can be accessed in `grid` is returned.
(In other words, the index of the row that has the first free field in the specified column when viewed from below, or the last free field when viewed from above).

The function uses `get_column` from task 3 to get the values of the specified column. (Note: The function returns the values from top to bottom. However, free fields must be searched for from bottom to top).

Examples: For `free_space(1, EXAMPLE_GRID_2)` we expect `4`. For `free_space(2, EXAMPLE_GRID_2)` we expect `None`. (See also the output of `EXAMPLE_GRID_2` above.)

Task 5 (1 bonus point)

To find out whether the game has ended in a draw, we check whether there are any empty spaces left on the board.

Complete the `grid_is_full` function so that it returns `True` if no field is free - and `False` if at least one field is still free. To do this, write a loop over all columns: apply `free_space` from task 4 to each column. Should `free_space` is `False` for all columns, no field is free. Or to put it the other way round: if `free_space` is `True` for at least one column, at least one field is still free.

Examples: `grid_is_full(EXAMPLE_GRID_1)` and `grid_is_full(EXAMPLE_GRID_2)` are `False`. On the other hand, we expect `True` for `grid_is_full(EXAMPLE_GRID_3)`. (Compare also output of `EXAMPLE_GRID_1`, `EXAMPLE_GRID_2` and `EXAMPLE_GRID_3` above).

Task 6 (1 bonus point)

Complete the drop_disc function so that a "disc" is dropped into the column specified with column_number. The current player (i.e. the value that you must enter in grid) is passed as the parameter player.

Use free_space from task 4 to find out in which row you need to change the grid. However, if you get None back when calling free_space, this means that there is no free space in the column selected by the player. In this case, the drop_disc function should return False. If, on the other hand, the value could be saved successfully, return
True back.

Example: drop_disc(2, 1, EXAMPLE_GRID_2) is False we expect On the other hand we expect for
drop_disc(1, 1, EXAMPLE_GRID_2) a True and can display the modified EXAMPLE_GRID_2 afterwards with print_grid.

Task 7 (1 bonus point)

Complete the function `all_elements_equal`. This function should check whether all elements of the list passed as parameter `sequence` are equal. If yes, `True` is returned

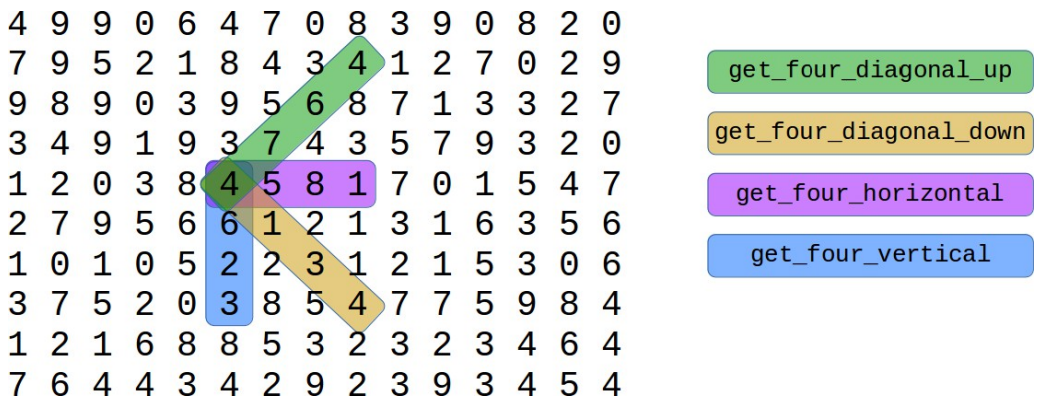
- if not, `False` accordingly. In the event that the parameter contains `sequence` is not a list, but the value `None`, `False` should also be returned.

Examples: For `all_elements_equal([1, 2, 3])` we expect `False`. The special case `all_elements_equal(None)` should also `False` as well. On the other hand are `all_elements_equal([1, 1, 1])`, `all_elements_equal([True, True, True])` or `all_elements_equal([False, False, False])` all `True`.

Task 8 (1 bonus point)

To check whether a player has won, we need the values of the four adjacent fields in four directions starting from a specific playing field - diagonally up and down, as well as horizontally and vertically.

The outgoing playing field is defined via the parameters `row_number` and `column_number`. In the example below, `row_number == 4` and `column_number == 5`. (Reminder: The indices of lists start at 0).



(Incidentally, the example is only intended to provide a better illustration. Our game board in `grid` has other dimensions and only contains values `0`, `1` or `2`).

To get the individual values of the 4 fields, there are the functions `get_four_diagonal_up`, `get_four_diagonal_down`, `get_four_horizontal` and `get_four_vertical`.

The `get_four_diagonal_up` function is already correctly implemented in the template code **and must not be changed**. However, it serves as a template for the remaining functions `get_four_diagonal_down`, `get_four_horizontal` and `get_four_vertical`. These functions still all access the same positions at the moment. How must the indices in `get_four_diagonal_down`, `get_four_horizontal` and `get_four_vertical` be changed so that the accesses are correct as shown in the graphic above?

Note: If `row_number` and `column_number` define a field that is at the edge of the board, it may not be possible to check all 4 directions. Therefore you will see a `try` - `except` block, which intercepts the appropriate `IndexError` if it occurs. You do not need to do anything more.

Examples: The functions are required for `player_has_won` so that you can test whether your changes were successful. For `player_has_won(EXAMPLE_GRID_1)`, `player_has_won(EXAMPLE_GRID_2)` and `player_has_won(EXAMPLE_GRID_3)` becomes `False` expected. For `player_has_won(EXAMPLE_GRID_4)`, `player_has_won(EXAMPLE_GRID_5)`, `player_has_won(EXAMPLE_GRID_6)` and `player_has_won(EXAMPLE_GRID_7)` on the other hand `True`.

Task 9 (1 bonus point)

Complete the function `next_player`. If the passed parameter `player` contains the value `1`, `2` should be returned. In **all** other cases `1`.

Examples: For `next_player(3)` or `next_player(None)` we expect the return value to be `1`. For `next_placer(1)` on the other hand `2`.

Task 10 (1 bonus point)

If you have already successfully completed all other tasks, the game should already be working. You can test this by calling the `game_loop` function.

However, the user input for selecting the column still lacks suitable error handling in the event that the user does not type in a number between 1 and 7, but something else such as `42` or `hello`.

Complete the `player_choice` function so that incorrect entries do not cause the program to terminate. Tip: As a rule, you need both a `try` - except block and at least one `if` query.
