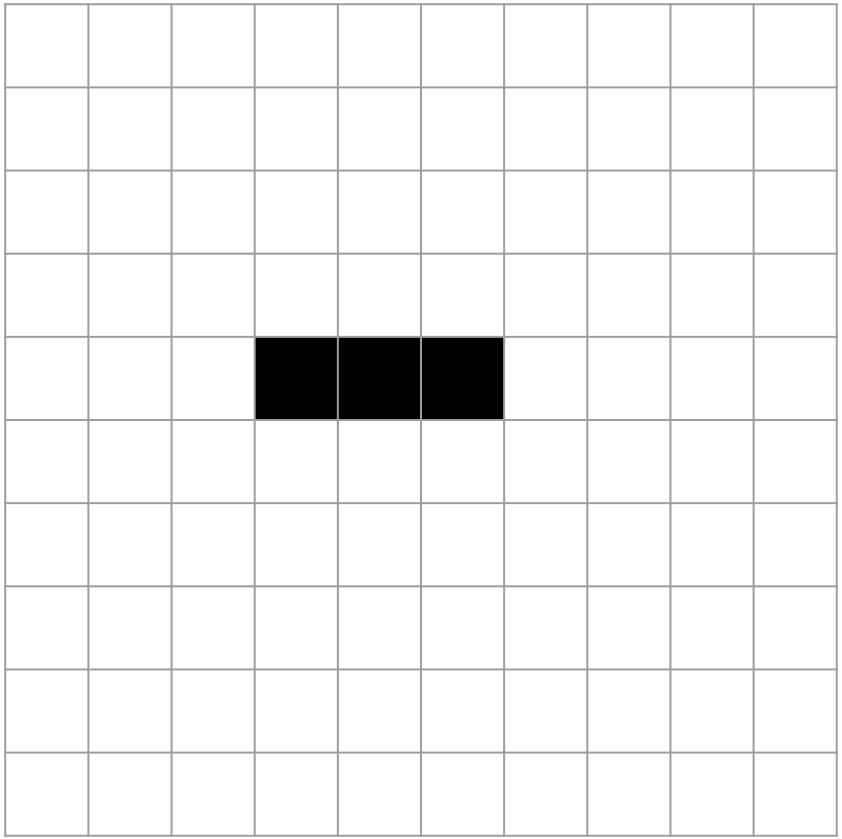
Game of life

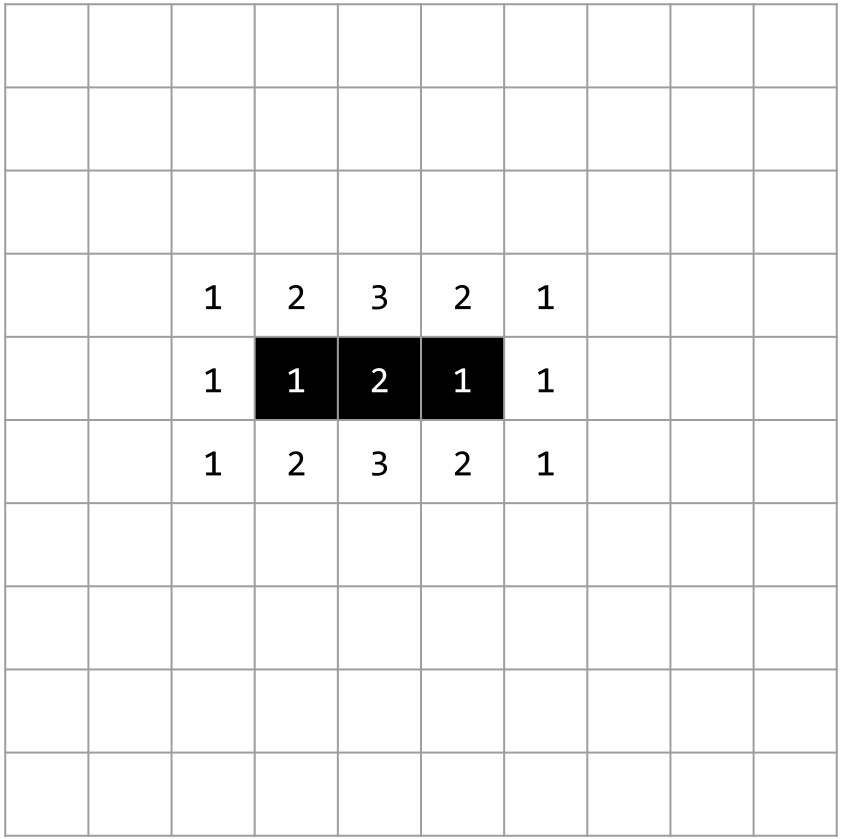
Python project

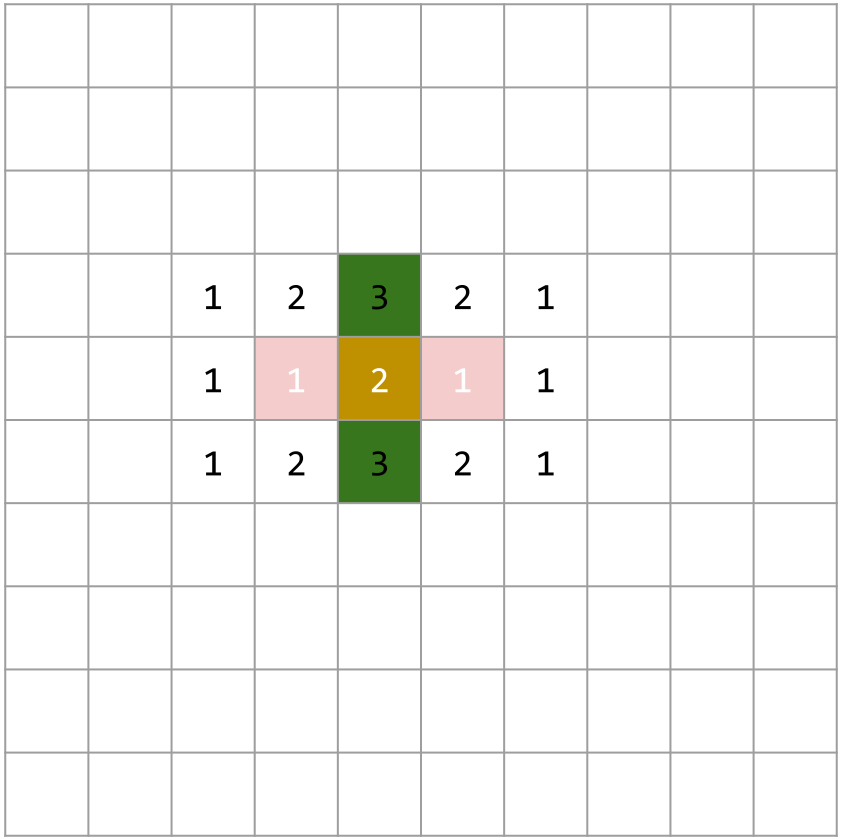
Rules

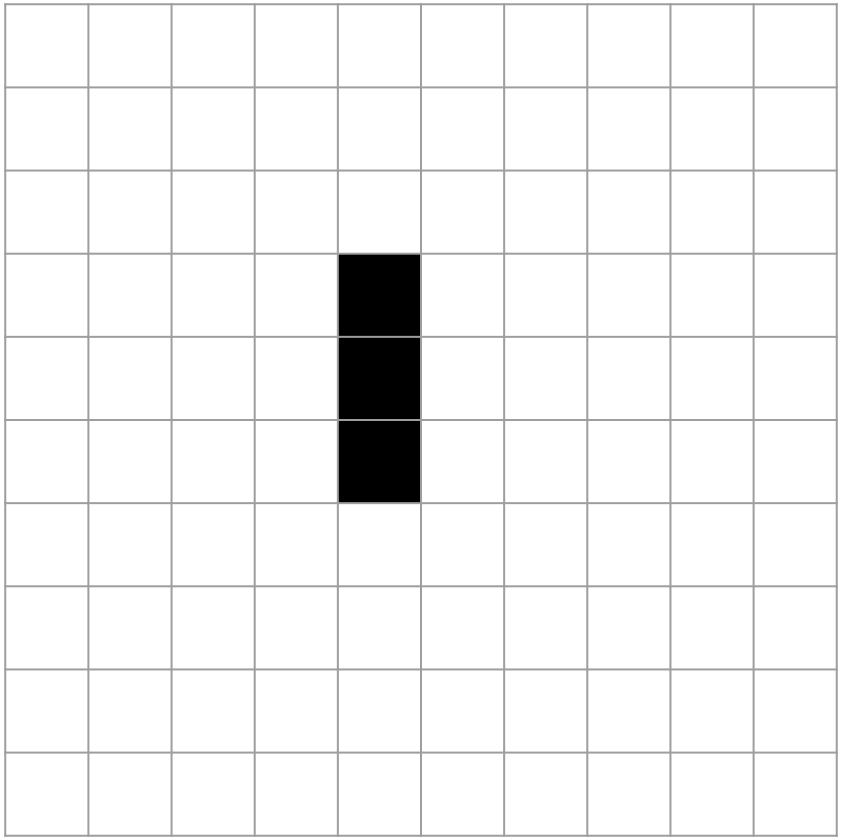
The universe of the Game of Life is an infinite, **two-dimensional orthogonal grid of square *cells***, each of which is in one of two possible states, ***live*** or ***dead***, (or *populated* and *unpopulated*, respectively). Every cell interacts with its eight neighbours, which are the cells that are horizontally, vertically, or diagonally adjacent. At each step in time, the following transitions occur:

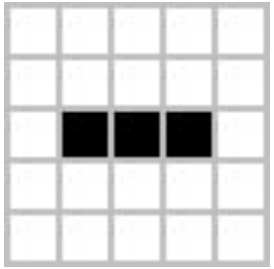
1. Any live cell with two or three live neighbours survives.
2. Any dead cell with three live neighbours becomes a live cell.
3. All other live cells die in the next generation. Similarly, all other dead cells stay dead.



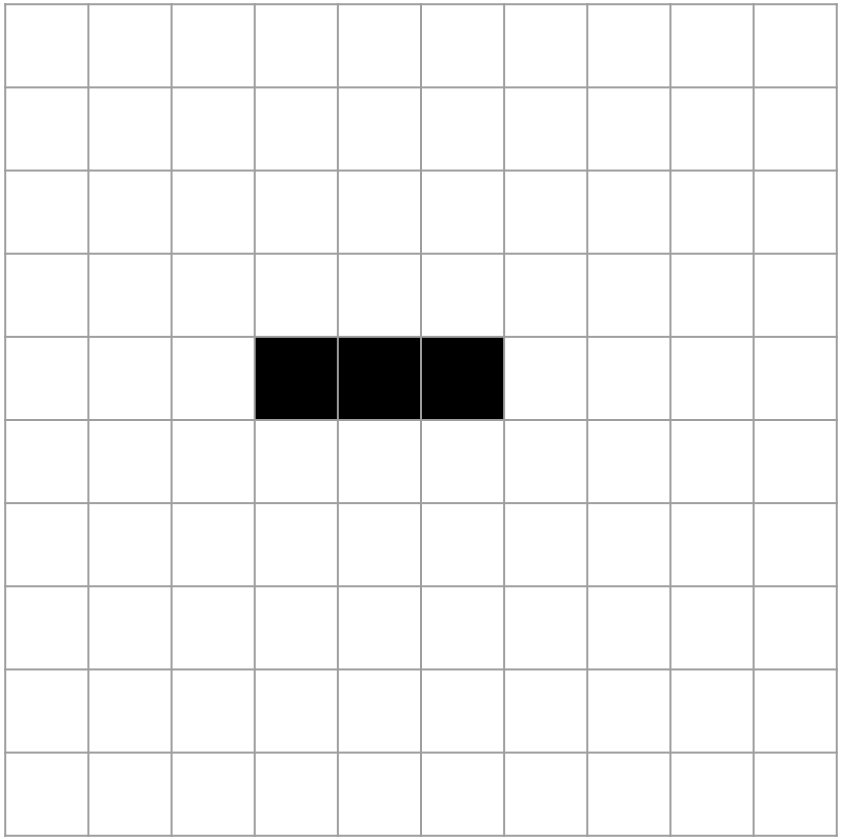

1 2 3 2 1
1 1 2 1 1
1 2 3 2 1


1 2 3 2 1
1 1 2 1 1
1 2 3 2 1



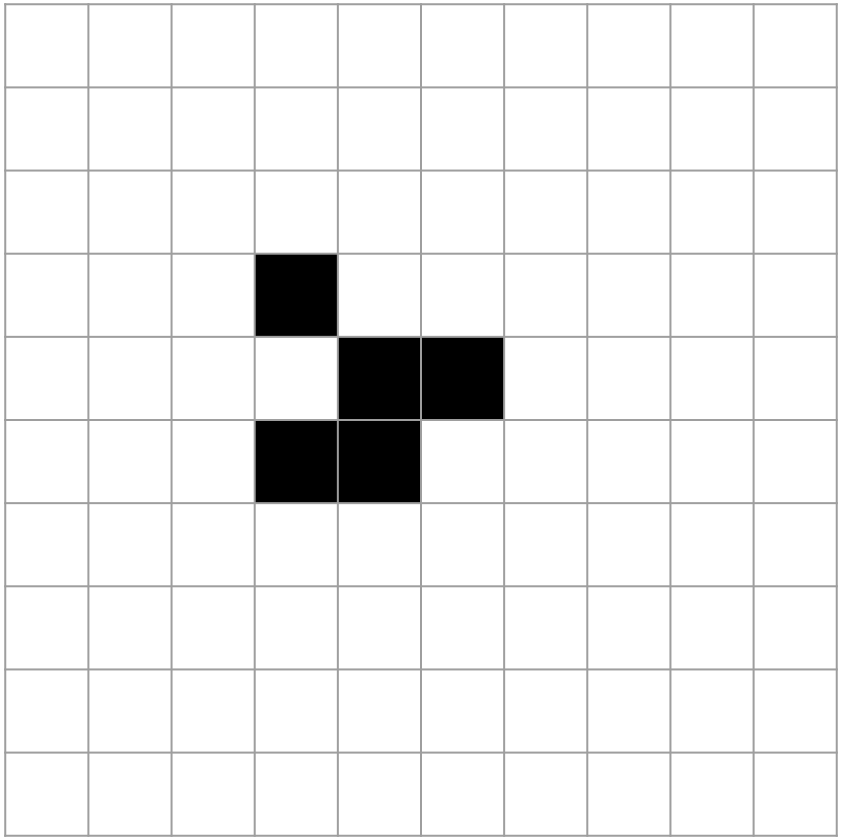


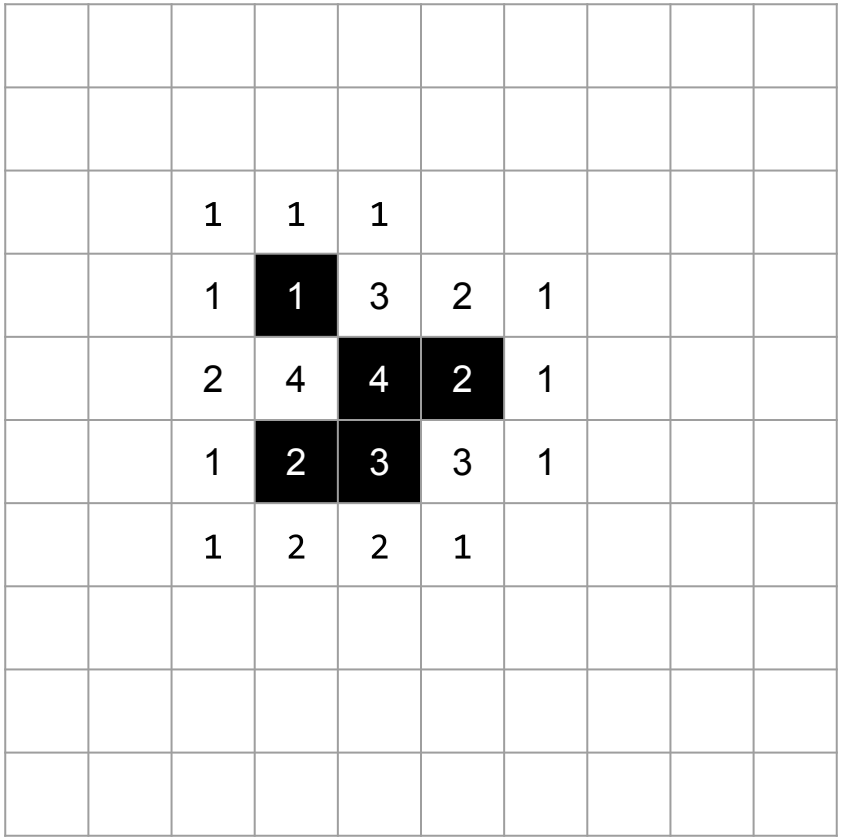
blinker

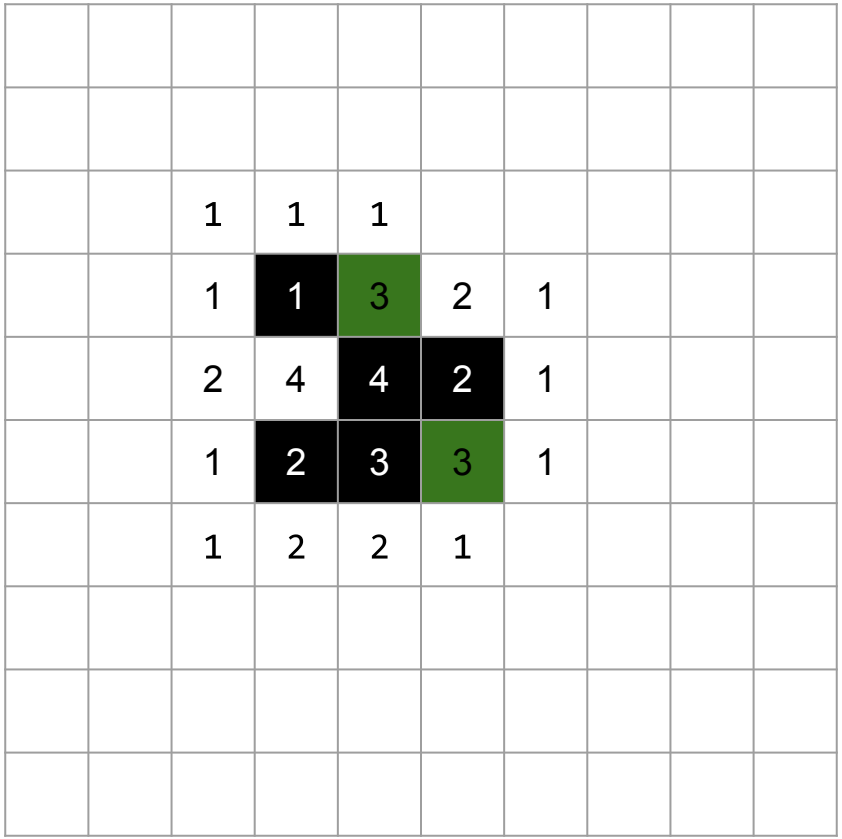


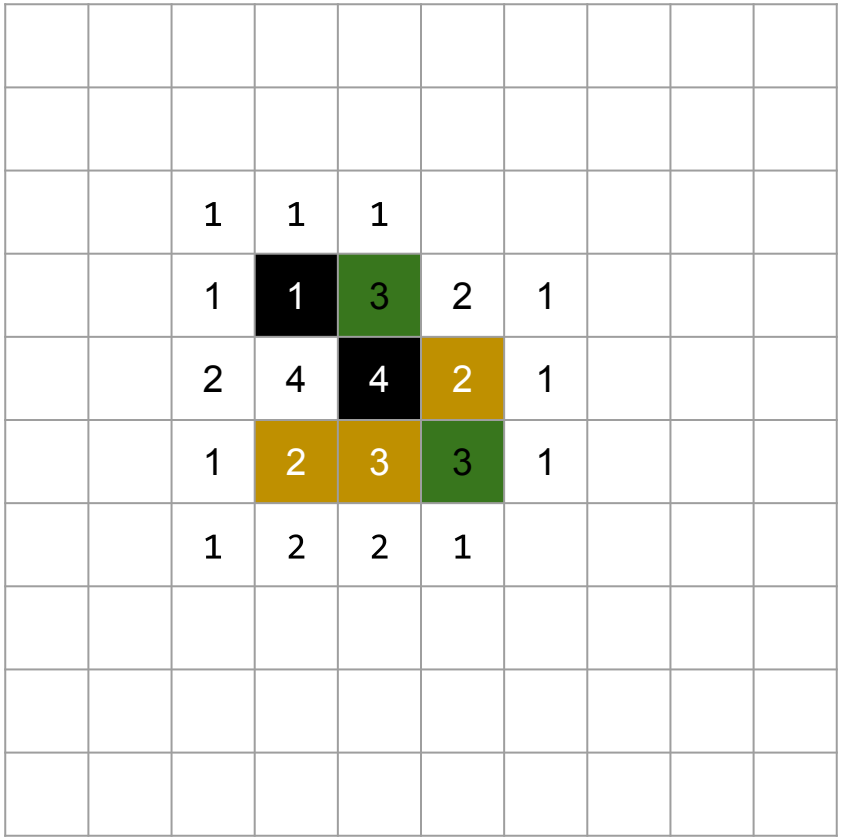
Task 1.

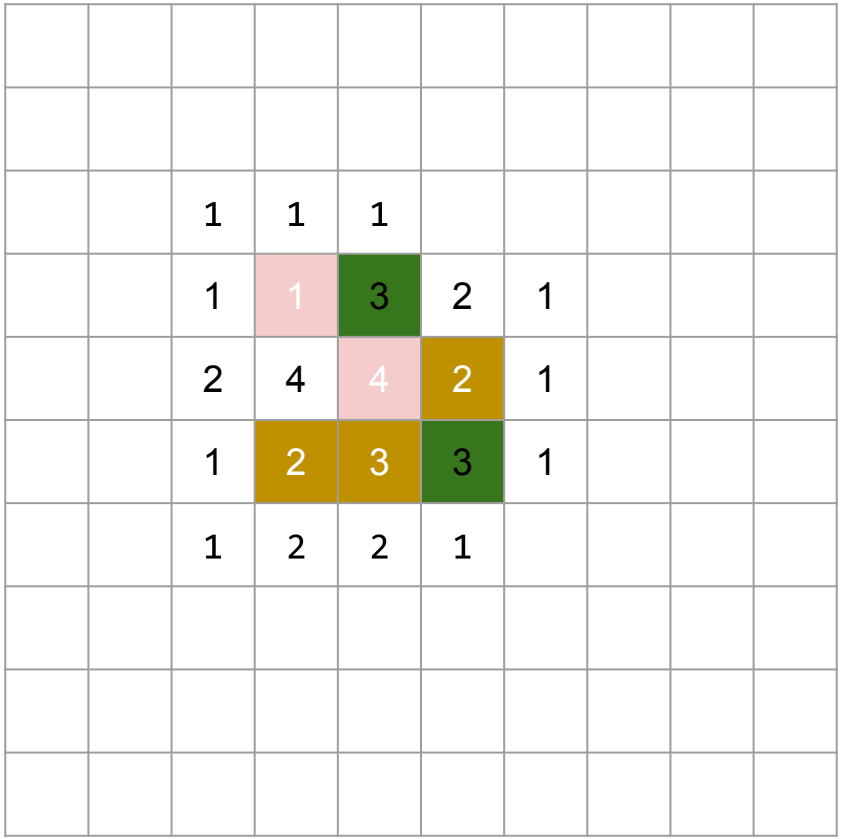
Predict the state of the game in the next generation.

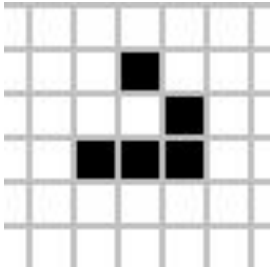



1 1 1
1 1 3 2 1
2 4 4 2 1
1 2 3 3 1
1 2 2 1

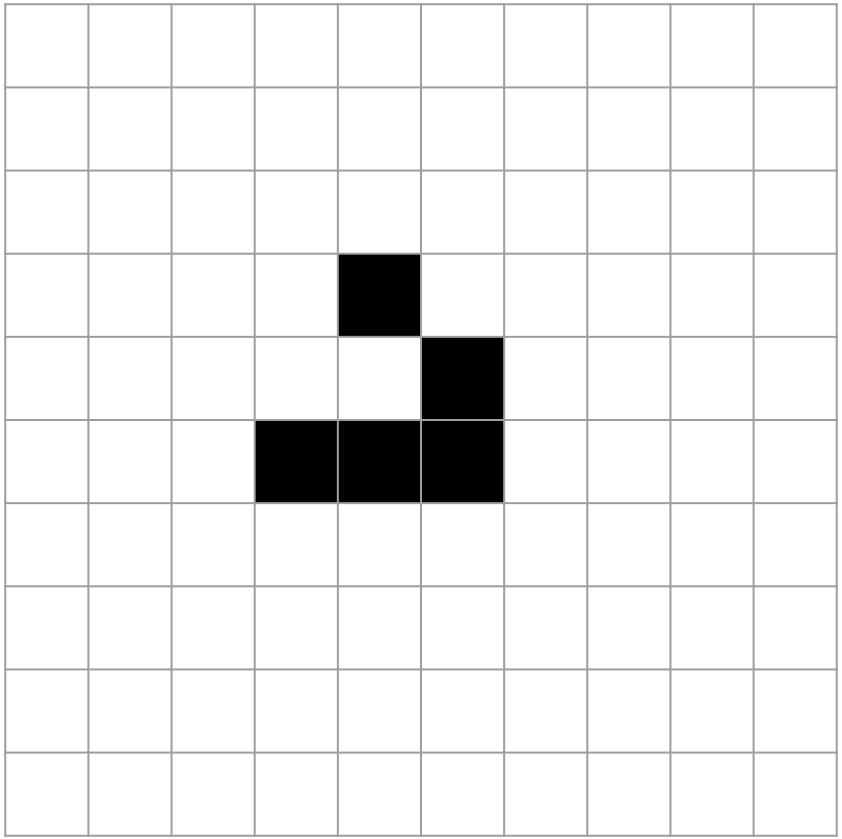

1 1 1
1 1 3 2 1
2 4 4 2 1
1 2 3 3 1
1 2 2 1


1 1 1
1 1 3 2 1
2 4 4 2 1
1 2 3 3 1
1 2 2 1


1 1 1
1 1 3 2 1
2 4 4 2 1
1 2 3 3 1
1 2 2 1



glider



Structuring our project

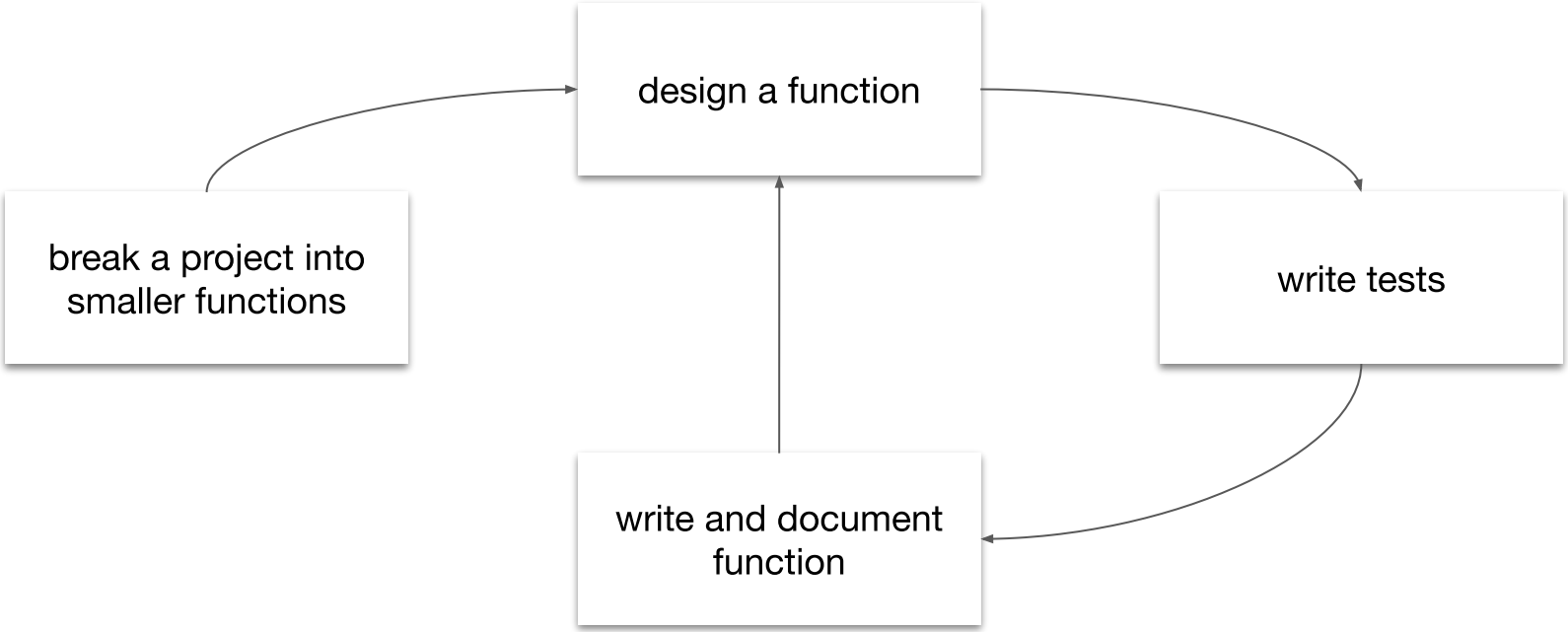
Representation:

- state of the game will be represented as 2D array (list of lists) with 0 if the cell is dead and 1 if the cell is alive

Assumptions:

- we will try to keep functions pure (not directly modyfing the state)
- we will create many small functions that usually take state as an argument and return either new state or something else
- we will create tests for all of our functions (TDD)
- we will document all our functions

Development process



Plan

Function name	What will do?
`create_grid`	Creates empty state.
`convert_to_string`	Creates string representation of the state.
`display`	Prints the state to the console.
`get_neighbors`	Determine neighbors of a cell.
`next_status`	Based on a cell status (*live* or *dead*) and its neighbors determine next status.
`next_state`	Makes one step of evolution.
`state_from_file`	Loads state from text file.

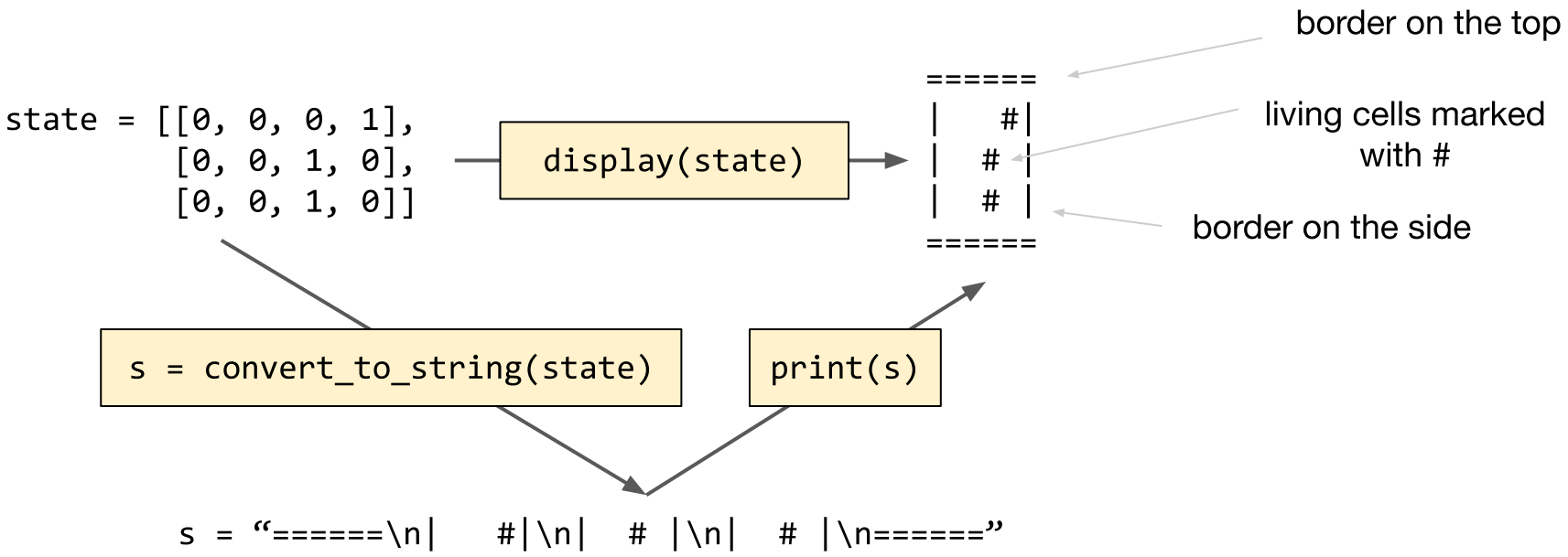
Create empty state

- `create_grid(n_rows, n_columns)`
  - return empty 2D array filled with zeros with desired size

`create_grid(5, 6)`

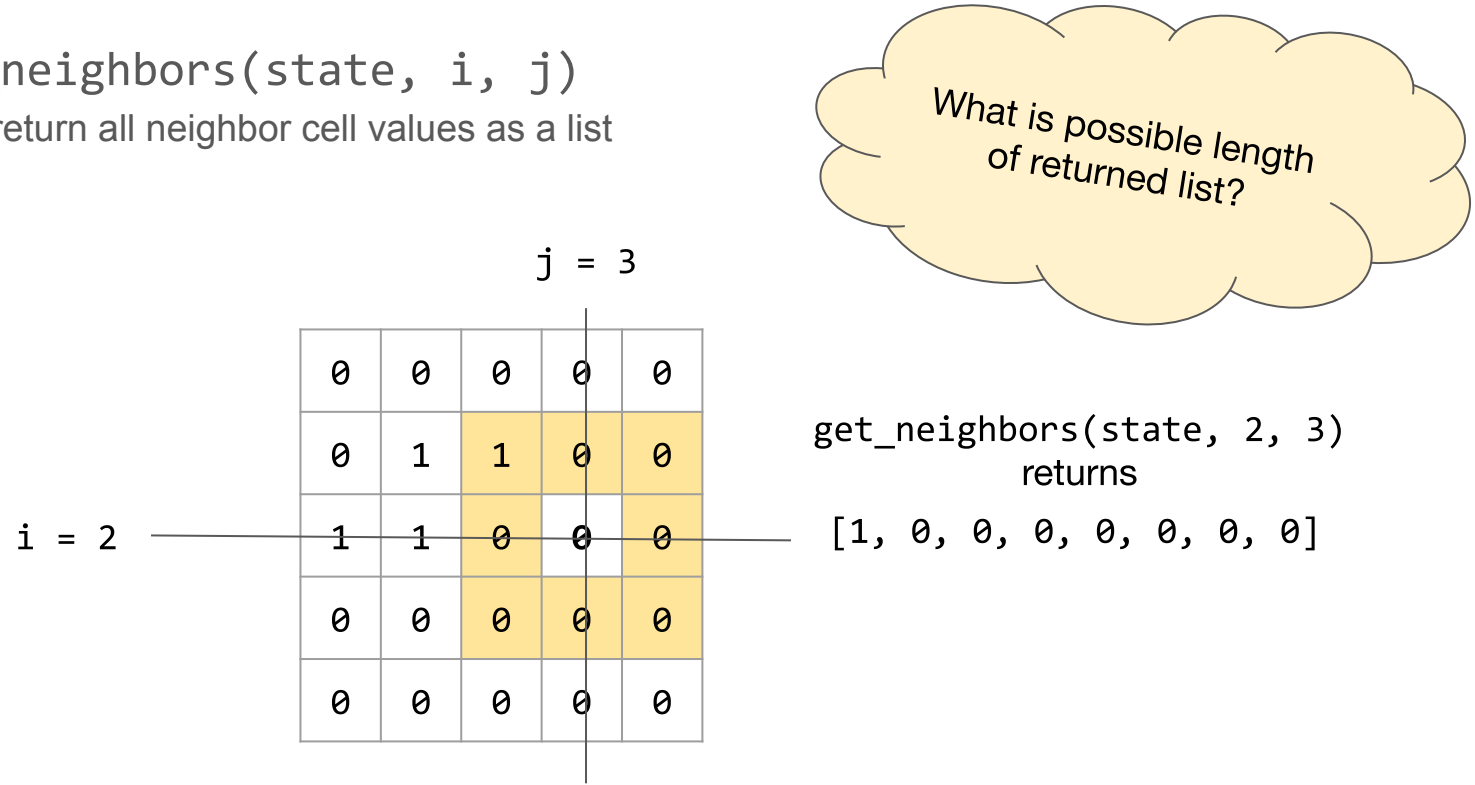
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0
0	0	0	0	0	0

Converting state to string



Get neighbors

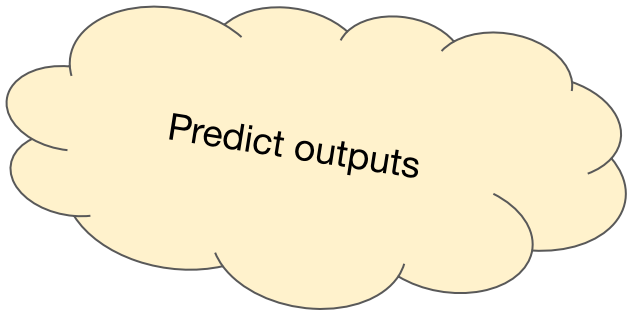
- `get_neighbors(state, i, j)`
  - return all neighbor cell values as a list



Next status

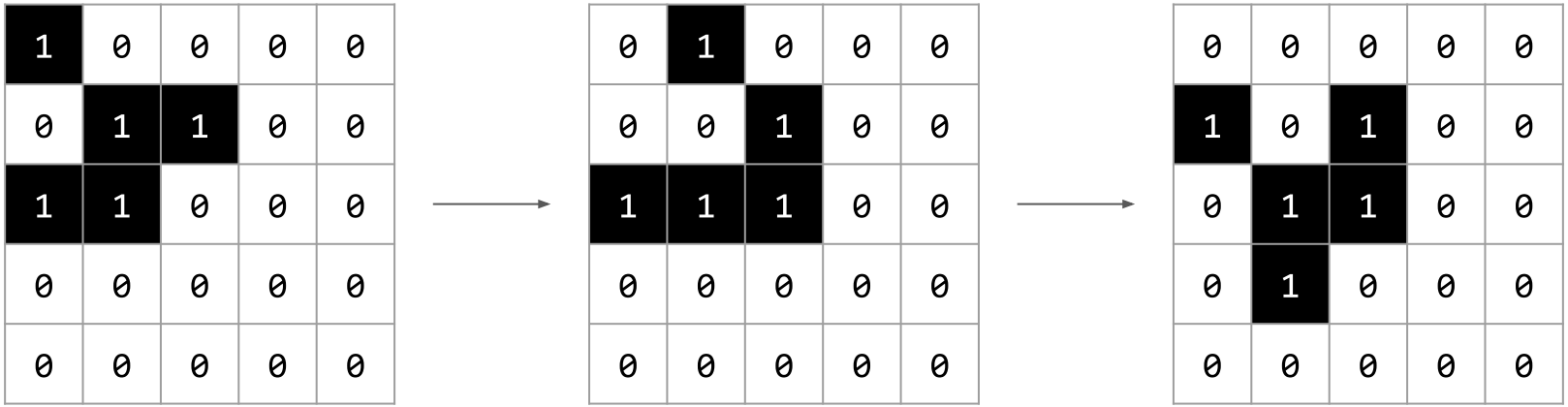
- `next_status(current, neighbors)`
  - implements the rules of the game
  - if current status is 0 (dead) and neighbors contains 3 living cells, it should return 1

```
next_status(0, [1, 0, 1, 0, 0, 0, 0, 0]) # returns ...
next_status(0, [1, 0, 1, 0, 0, 0, 0, 1])
next_status(1, [0, 0, 0, 0, 1, 0, 0, 0])
next_status(1, [0, 0, 0, 1, 1])
next_status(1, [1, 1, 1])
```



Next state

- `next_state(state)`
  - return new state after one generation of evolution



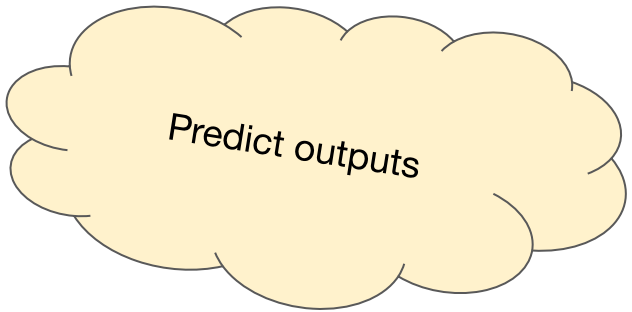
Next state implementation

1. create empty grid of matching size (to keep the function pure)
2. loop over all squares and fill them with new cell status
    - for each square grab current value and get all neighbors
    - then decide new status based on that

Load state from file

- `next_status(current, neighbors)`
  - implements the rules of the game
  - if current status is 0 (dead) and neighbors contains 3 living cells, it should return 1

```
next_status(0, [1, 0, 1, 0, 0, 0, 0, 0]) # returns ...
next_status(0, [1, 0, 1, 0, 0, 0, 0, 1])
next_status(1, [0, 0, 0, 0, 1, 0, 0, 0])
next_status(1, [0, 0, 0, 1, 1])
next_status(1, [1, 1, 1])
```



Put it all together

Using existing function create some initial state and run the simulation.