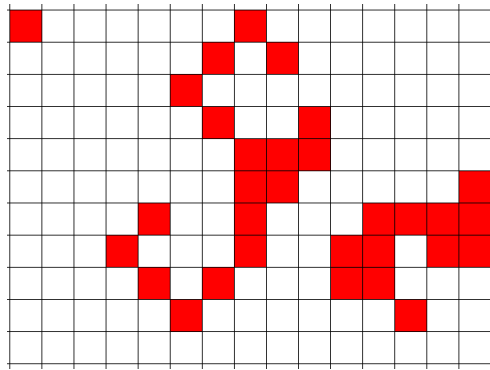# Game of life

The game of life *is a simple model of the evolution of a population of cells that split and die over time. The "game" consists of finding initial configurations that give interesting evolution: some groups of cells disappear, others stabilize, some move. . .*
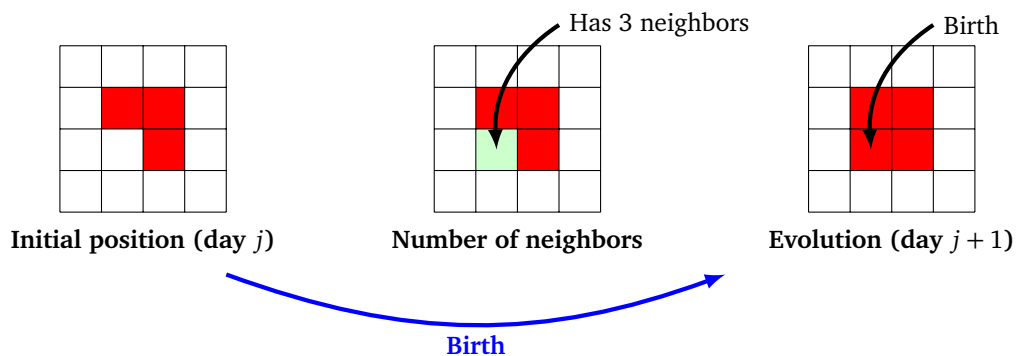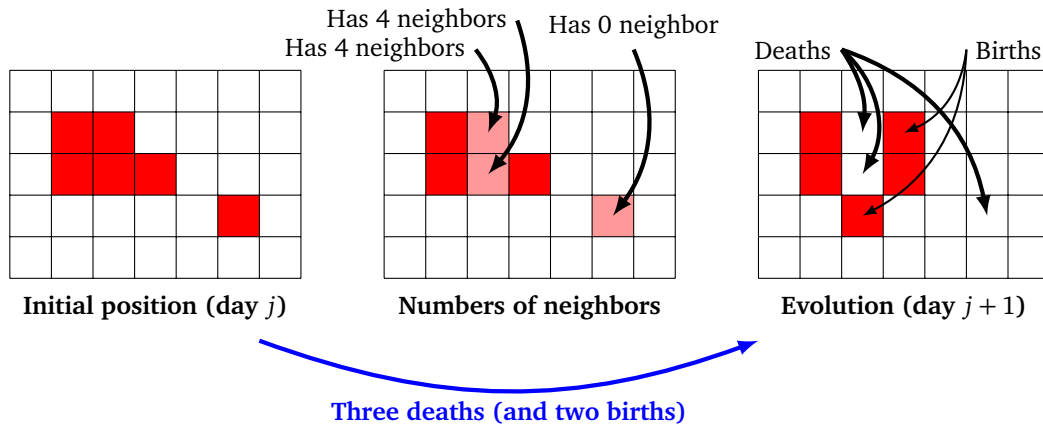

Lesson 1 (Rules of the game of life)**.**
The *game of life* takes place on a grid. Each square can contain one cell. Starting from an initial configuration, each day cells will be born and others will die depending on the number of its neighbors.
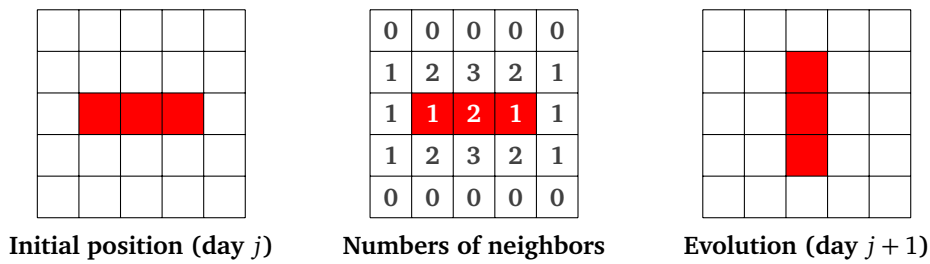Here are the rules:

- For an empty square on day j and having exactly 3 neighboring cells: a cell is born on day $j + 1$.

| Has 3 neighbors | Birth |

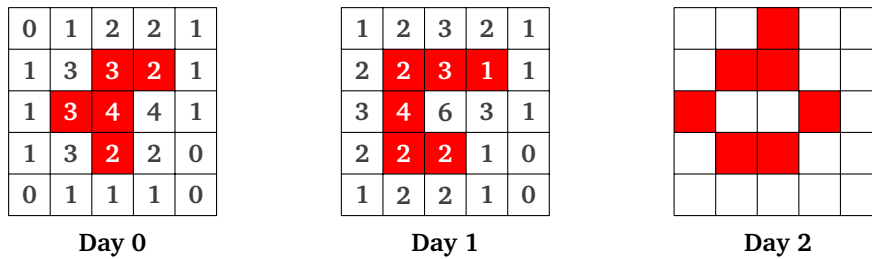| **Initial position (day j)** | **Number of neighbors** | **Evolution (day $j + 1$)** |

Birth

- If a square containing a cell at day j, has either 2 or 3 neighboring cells: then the cell continues to live. In other cases the cell dies (with 0 or 1 neighbors, it dies of isolation, with more than 4 neighbors, it dies of overpopulation).

Has 4 neighbors
Has 4 neighbors
Has 0 neighbor
Deaths
Births

Initial position (day j)   **Numbers of neighbors**   **Evolution (day $j+1$)**

Three deaths (and two births)

Here is a simple example, the "blinker".

0	0	0	0	0
1	2	3	2	1
1	1	2	1	1
1	2	3	2	1
0	0	0	0	0

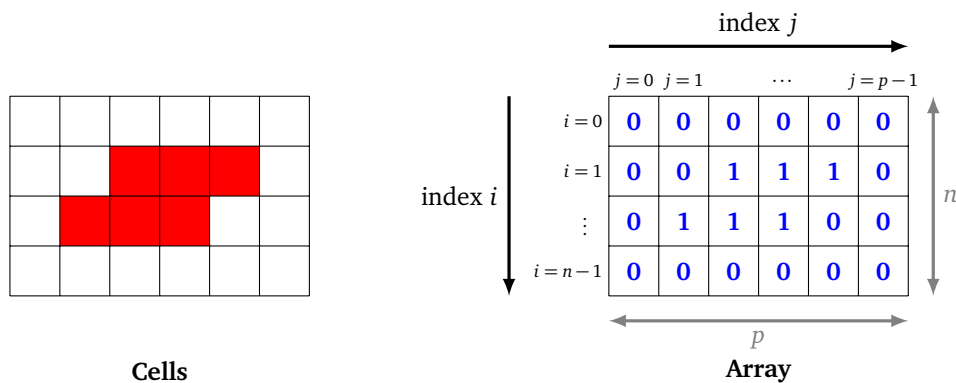**Initial position (day j)**   **Numbers of neighbors**   **Evolution (day $j+1$)**

Here is another example (with the number of neighbors written in each case), the first evolution and the evolution of the evolution.

0	1	2	2	1
1	3	3	2	1
1	3	4	4	1
1	3	2	2	0
0	1	1	1	0

1	2	3	2	1
2	2	3	1	1
3	4	6	3	1
2	2	2	1	0
1	2	2	1	0

Day 0   **Day 1**   **Day 2**

Activity 1 (Display).

Goal: define and display arrays.

We model the living space of the cells by a double entry table, containing integers, 1 to indicate the presence of a cell, 0 otherwise. Here is an example of the "toad" configuration and its array:

index j

	$j=0$	$j=1$		$\cdots$		$j=p-1$
$i=0$	0	0	0	0	0	0
$i=1$	0	0	1	1	1	0
$\vdots$	0	1	1	1	0	0
$i=n-1$	0	0	0	0	0	0

index i

n

p

Cells   **Array**

1.
- Initialize two variables: `n` (the height of the table) and `p` (the width) (for example at 5 and 8).
- Define a two-dimensional table filled with zeros by the command:
   `array = [[0 for j in range(p)] for i in range(n)]`
- By using instructions of the type `array[i][j] = 1`, fill in the table to define the configurations of the blinker, the toad ...

2. Program the display of a given array on the screen. For example, the blinker is displayed as follows:

```
00000000
00000000
00111000
00000000
00000000
```

Hint. By default the `print()` function starts a new line on each call (it effectively adds `"\n"` which is the end of line character). It can be specified not to do this by the option `print("My text",end="")`.

Activity 2 (Graphic display).

Goal: create a graphical display of a cell configuration.
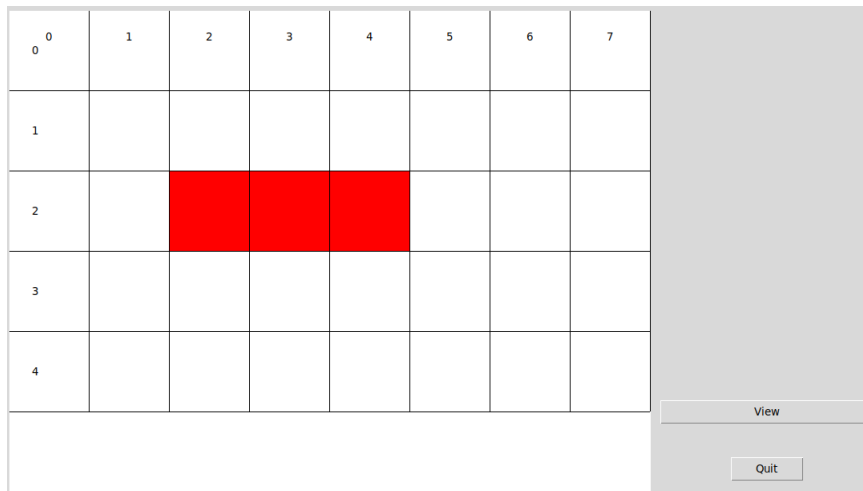
1. Program the creation of a `tkinter` window (see the "Statistics – Data visualization" chapter). Write a `draw_grid()` function, without parameters, that displays a blank grid of coordinates. You will need a constant `scale` (for example `scale = 50`) to transform the indices i and j into graphic coordinates x and y, depending on the relationship: $x = s \times j$ and $y = s \times i$ (s being the scale).
   Bonus. You can also mark the values of the indices i and j at the top and the left to make it easier to read the grid.
2. Build a `draw_array(array)` function that graphically displays the cells from a table.
   Bonus. Add a "View" button and a "Quit" button (see the "Statistics – Data visualization" chapter).

Activity 3 (Evolution).

Goal: calculate the evolution of a configuration from one day to the next.

1. Program a `number_neighbors(i,j,array)` function that calculates the number of living neighboring cells to the cell (i, j).
   Hints.
   - There is a maximum of 8 possible neighbors. The easiest way is to test them one by one!
   - For counting, it is necessary to be very careful with the cells that are placed near an edge (and have less than 8 possible neighbors).

2. Program an `evolution(array)` function that receives a table as an input and returns a new array corresponding to the situation on the next day, according to the rules of the game of life described at the beginning. For example, if the table on the left corresponds to the input, then the output corresponds to the table on the right:

<table>
<tr><td>00000000</td><td></td><td>00000000</td></tr>
<tr><td>00000000</td><td></td><td>00010000</td></tr>
<tr><td>00111000</td><td>evolves in</td><td>00010000</td></tr>
<tr><td>00000000</td><td></td><td>00010000</td></tr>
<tr><td>00000000</td><td></td><td>00000000</td></tr>
</table>

Hints. To define a new table, use the command:
```
new_array = [[0 for j in range(p)] for i in range(n)]
```
and then modify the table as desired.

Activity 4 (Iterations).

Goal: complete the graphic program so that the user can define configurations and make them evolve with a simple click.

1. Improve the graphics window to make the user's life easier:
   - An "evolve" button which displays the next evolution with each click.
   - Buttons to display predefined configurations (in the screenshot below is the configuration "pentadecathlon").
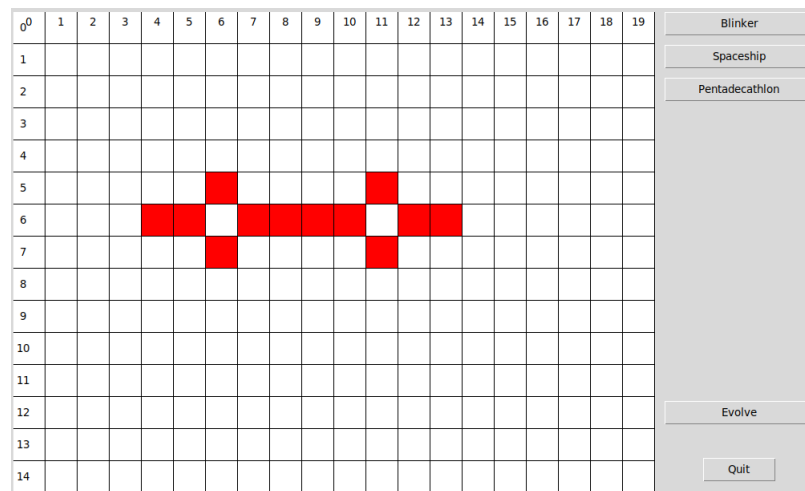
2. Perfect your program so that the user can draw the configuration he wants with mouse clicks. Clicking on an extinguished cell turns it on, clicking on an living cell turns it off. You can break this task down into three functions:

- on_off(i,j), which switches the cell (i,j).
- xy_to_ij(x,y) that converts graphic coordinates (x,y) into integers coordinates (i,j) (use the scale variable and the integer division).
- action_click_mouse(event) to retrieve the coordinates (x,y) with a mouse click (see course below) and switch the clicked box.
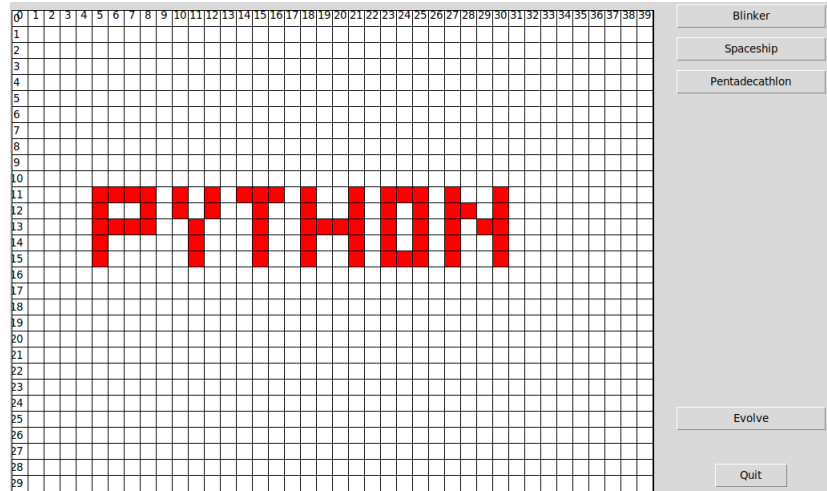
Lesson 2 (Mouse click).

Here is a small program that displays a graphic window. Each time the user clicks (with the left mouse button) the program displays a small square (on the window) and displays "Click at $x = \ldots$, $y = \ldots$" (on the console).

```python
from tkinter import *

# Window
root = Tk()
canvas = Canvas(root, width=800, height=600, background="white")
canvas.pack(side=LEFT, padx=5, pady=5)

# Catch mouse click
def action_mouse_click(event):
    canvas.focus_set()
    x = event.x
    y = event.y
    canvas.create_rectangle(x,y,x+10,y+10,fill="red")
    print("Click at x =",x,", y =",y)
    return

# Association click/action
canvas.bind("<Button-1>", action_mouse_click)

# Launch
root.mainloop()
```

Here are some explanations:

- The creation of the window is usual. The program ends with the launch of the `mainloop()` command.

- The first key point is to associate a mouse click to an action, that's what this line does:
  ```
  canvas.bind("<Button-1>", action_mouse_click)
  ```
  Each time the left mouse button is clicked, `Python` executes the function `action_mouse_click`. (Note that there are no brackets for the call to the function.)

- Second key point: the `action_mouse_click` function retrieves the click coordinates and then does two things here: it displays a small rectangle at the click location and prints the (x, y) coordinates to the terminal window.

- The coordinates x and y are expressed in pixels; $(0, 0)$ refers to the upper left corner of the window (the area delimited by `canvas`).

Here are some ideas to go further:

- Make sure that the grid automatically adapts to the screen width, i.e. calculate `scale` based on `n` and `p`.

- Add an intermediate step before evolving: color a cell that will be born in green and a cell that will die in black. For this purpose the elements of `array` can take values other than 0 and 1.

Here are some interesting configurations.

Square   **Spaceship**   **Pulsar**
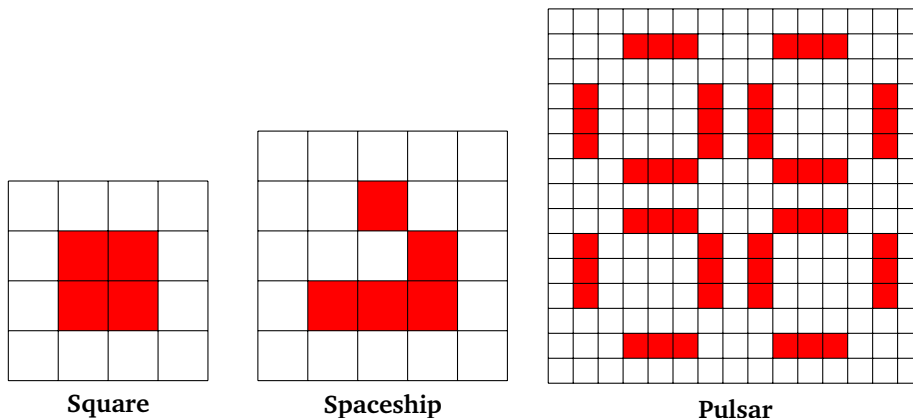
You will find many more on the Internet but above all have fun discovering new ones!

In particular, find configurations:

- which remain fixed over time;

- which evolve, then become fixed;

- which are periodic (the same configurations come back in loops) with a period of 2, 3 or more;

- which travel;

- which propel cells;

- whose population is increasing indefinitely!