Programming in Python[1]

Mattia Monga

Dip. di Informatica
Università degli Studi di Milano, Italia
mattia.monga@unimi.it

Academic year 2022/23, I semester

Lecture XV: A game of life

Using the notebook in a virtual environment

Since we are now interested in graphics, Jupyter notebooks can be very convenient to see pictures together with the code.

1. We set up a virtual environment as usual
2. With `pip install notebook` we have the Jupyter notebook machinery available
3. I normally want to have also a clean `.py` file, since `.ipynb` do not play well with configuration management (git) and other command line tools like the type checker or doctest: thus I suggest to install `jupytext`; it needs a `jupytext.toml` text file telling `.ipynb` and `.py` files are **paired**, *i.e.*, they are kept synchronized.

   ```
   # Always pair ipynb notebooks to py files
   formats = "ipynb,py"
   ```
4. lunch the notebook with `jupyter notebook`

A game of life

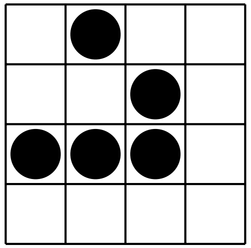
In 1970, J.H. Conway proposed his Game of Life, a simulation on a 2D grid:

1. Every cell can be *alive* or *dead*: the game start with a population of alive cells (*seed*)
2. any alive cell with less of 2 alive neighbours dies (*underpopulation*)
3. any alive cell with more than 3 alive neighbours dies (*overpopulation*)
4. any dead cell with exactly 3 alive neighbours becomes alive (*reproduction*)

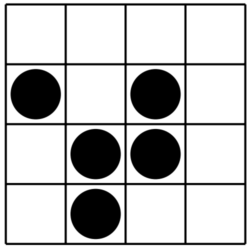
The game is surprisingly rich: many mathematicians, computer scientists, biologists. . . spent their careers on the emerging patterns!

Life forms

There are names for many "life forms": *still lifes*, *oscillators*, *starships*. . .
A famous starship is the glider:



1	1	2	1
3	5	3	2
1	3	2	2
2	3	2	1



The glider repeats itself in another position after 4 generations.

Python implementation

To implement a Game of Life simulation in Python, we can:

- use a `ndarray` for the grid
- each cell contains 0 (dead) or 1 (alive)
- for simplicity we can add a "border" of zeros

0	0	0	0	0
0	1	1	1	0
0	1	0	1	0
0	1	1	0	0
0	0	0	0	0

Avoiding loops

For a 1-D array `X`

0	**1**	**1**	**0**	**1**	0

All the neighbours on the right `X[2:]`

0	**1**	**1**	**0**	**1**	0

All the neighbours on the left `X[:-2]`

What does `X[2:] + X[:-2]` represent? The sum is (yellow) element by (yellow) element, the result is: `[1,1,2,0]`
Can you think to a similar solution for the 2-D case?

Avoiding loops

0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	0
0	**0**	**0**	**1**	**0**	0
0	**1**	**1**	**1**	**0**	0
0	**0**	**0**	**0**	**0**	0
0	0	0	0	0	0

`X[1:-1, 2:]`

0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	0
0	**0**	**0**	**1**	**0**	0

Avoiding loops

`X`

0	0	0	0	0	0
0	**0**	**1**	**0**	**0**	0
0	**0**	**0**	**1**	**0**	0
0	**1**	**1**	**1**	**0**	0
0	**0**	**0**	**0**	**0**	0
0	0	0	0	0	0

`N`

0	0	0	0	0	0
0	1	1	2	1	0
0	3	5	3	2	0
0	1	3	2	2	0
0	2	3	2	1	0
0	0	0	0	0	0

`X == 1` `N > 3`

Death by overpopulation: `X[(X == 1) & (N > 3)] = 0` (empty in this case!)

Homework

- `https://classroom.github.com/a/bmOfyQYC`