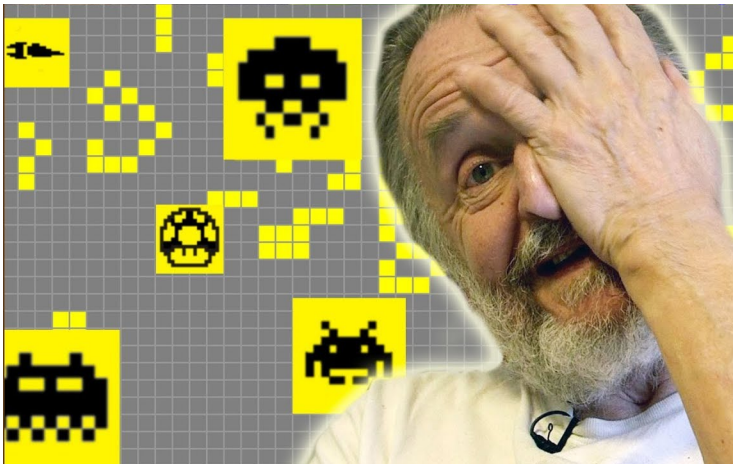
# Project: The Game of Life
# Mathematical Programming with Python

**MATH 2604: Advanced Scientific Computing 4**
**Spring 2025**
**Monday/Wednesday/Friday, 1:00-1:50pm**
**Room A202 Langley Hall**

https://people.sc.fsu.edu/~jburkardt/classes/python_2025/life/life.pdf

---



*The Game of Life was invented by John Conway.*

"Life"

- *John Horton Conway invented a grid simulation called the Game of Life;*
- *Life is an example of a 2D Cellular Automaton.*
- *It is a "zero person" game; once it is initialized, there is no interaction with a player.*
- *The grid of black and white squares evolves one time step at a time.*
- *Simple rules determine whether a living cell will die, or an empty cell will come alive.*

The game of Life is a possible project topic.

## 1 The Game of Life

In 1970, an article appeared in the Mathematical Games section of Scientific American, describing a peculiar kind of game invented by mathematician John Horton Conway, called Life. It wasn't really a game, since no players were needed; instead, it essentially played itself. Mathematicians and computer scientists soon realized that this simple game was capable of a profound range of unexpected behaviors worth studying.

For more information on Life, you can look at the many books and online discussion, or refer to the original article:

```
Martin Gardner,
Mathematical Games:
The Fantastic Combinations of John Conway's new solitaire game "Life",
```

Scientific American,
Volume 223, Number 4, October 1970, pages 120-123.

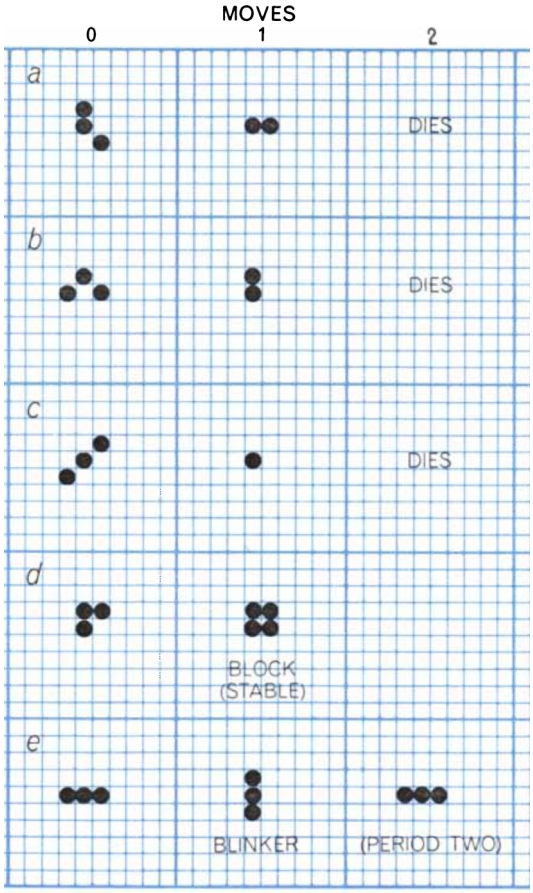
The game started with an arbitrary pattern of markers on a grid. A new pattern was then determined by letting some markers "die" and others be "born". Depending on the initial condition, the markers might quickly all disappear, or proliferate, or fall into steady, repeating patterns.

The results were suprisingly complex, but the rules were simple which updated the current pattern to determine the next one. The playing board was a grid of squares, which were called "cells". Each cell with a marker was said to be "living", the others were "empty". Except along the borders, each cell has four neighbors, north, south, east and west. The fate of a cell in the next pattern was determined by its current state and that of these neighbors:

1. An empty cell with exactly three living neighbors becomes living;
2. Otherwise, an empty cell stays empty;
3. A living cell with less than two or more than three living neighbors dies;
4. Otherwise, a living cell stays alive;

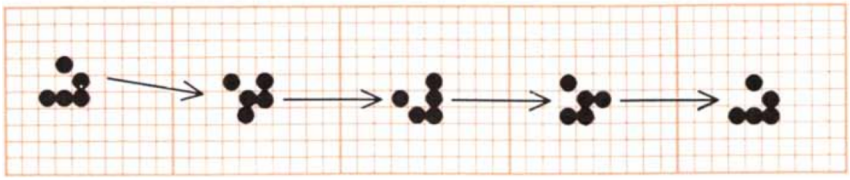
## 2 Interesting Initial Conditions

Here, for example, are some of the possible outcomes if our starting setup involves a triplet of living cells.

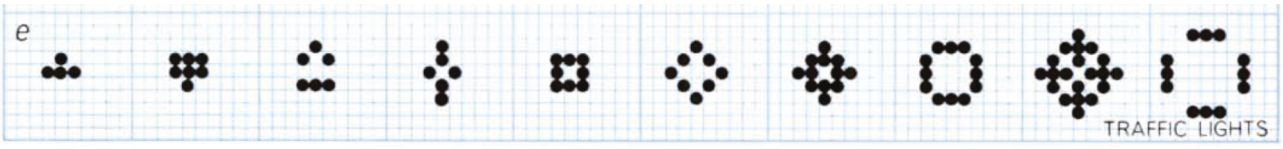


The first three cases die out, the fourth reaches a permanent unchanging state, while the fifth behaves in a periodic fashion.

People investigating the game soon made a number of discoveries. One of the first was a pattern called "the glider", which twisted and turned and then reappeared on the fourth step, shifted one cell down and right. This means that, on an infinite grid, it would simply continue to sail away forever.



If we start with four living cells in the following position, the pattern quickly becomes symmetric, then seems to get complicated, and finally resolves into an alternating design known as "traffic lights".




Perhaps one of the strangest cases of a tiny initial condition that goes wild is known as the "R" pentomino. seems to grow wildly for hundreds of steps, creating gliders that shoot off in different directions, and generating small patches of stability that later get "eaten" by other blobs.


Since the simplest initial conditions can vary so much in their later evolution, this makes Life a fruitful playground for investigations about games on grids, growth simulation and cellular automata.

# 3 A Python implmmentation

To explore the possible behaviors of this simulation of life, we can set up an $m \times n$ grid of cells, scatter living cells across it at random, apply the evolution rules and display each stage of the development.

Try writing a program *life.py* that carries out the evolution steps for a given initial condition. To get started, you can use a small $20 \times 20$ grid, and test some of the starting configurations discussed above. Once you are satisfied, you should try a larger grid, and random initial conditions.

# 4 Initial Conditions

Your first version of the code can set the initial condition by setting the entire grid to zero (dead) and then specifying the few living cells in your pattern. In an improved version, you might want to set each cell at random, perhaps using commands like:

```
grid = np.random.choice ( [0,1], m * n, p = [ 0.8, 0.2 ] )
grid = np.reshape ( grid, [ m, n ] )
```

If you are very ambitious, you can try to learn about the `pygame` library, for game design. One of its features lets you use mouse clicks to initialize individual cells on the grid.

# 5 Counting neighbors

To update the grid, you need a function that returns the number of living neighbors of each cell. Your first version can use a `for()` loop. Remember that cells along the boundary don't have all four neighbors, so be careful to treat them correctly.

For an improved version of your code, you could look at periodic boundary conditions. This just means that when you count neighbors for a cell along the boundary, you wrap around to the other side to get the missing neighbor.

If you can set up the periodic boundary conditions, and you think about it carefully and write your code using the modulo function, you can count the neighbors of all the cells in a single Python statement.

# 6 Display or animation

You can display the grid at each step using `matplotlib`. You will probably have to manually close the current plot in order to see the next one.

As an improvement, you could save each plot to a separate file, then figure out how to merge those files into a single GIF or MPEG animation.

If you are ambitions, you can try to figure out how to use Python's FuncAnimation feature so that as you run your program, the sequence of states shows up automatically on the screen.