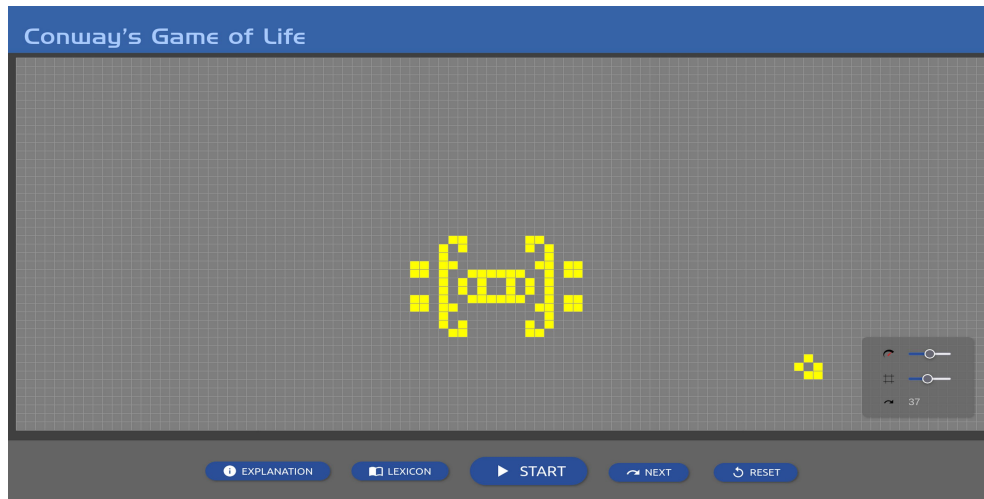


## Part 1: Conway's Game of Life

### Play with Conway's Game of Life

Play with an interactive version of Conway's Game of Life: <https://playgameoflife.com/>

You may start with an already defined array in the Lexicon or 'paint' your own pattern.



*Guiding questions:*

- What do you notice about the isolated bright 2 x 2 squares?
- From the Lexicon, try the “Barge” and run a few generations. Try painting or unpainting some blocks around the original Barge pattern. Write down some of your observations.

### Implementing Conway's Game of Life

Now, let us look at a small implementation of Conway's Game of Life using Python.

This implementation follows components of a tutorial from Real Python:

<https://realpython.com/conway-game-of-life-python/>

We will perform some exercises to understand the code a little bit. Most of the exercises will be about inferring how the code works. Areas marked as a “**Bonus**” are for the those who are curious about further details about the implementation or who would like ideas for additional modifications and simulations.

## Rules for Conway's Game of Life

Conway's Game of Life is a prototypical example of cellular automata, which are a class of discrete methods that share certain defining characteristics:

- The system is defined with a discrete regular, spatial grid where each cell has a finite number of discrete values associated with it.
- All cell values are updated deterministically in the same discrete time step
- The rules for changing cell values depend only the value of local neighboring cells

In the Game of Life, we have a two-dimensional square lattice, where each cell has an associated value of 'dead' or 'alive' (or some other binary representation such as 0 or 1).

The rules of the Game of Life can be summarized as follows:

1. Alive cells stay alive if they have **two** or **three** living neighbors.
2. Dead cells are revived to alive cells if there are **exactly three** living neighbors (imitating reproduction).
3. Alive cells die if there are **fewer than two** living neighbors (underpopulation) or **more than three** living neighbors (overpopulation).

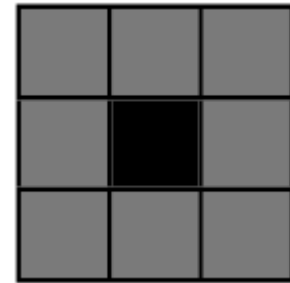


Figure 1: The Moore environment where each cell has eight neighbors (gray cells).

For a square lattice, we consider each cell to have eight neighbors as shown in Figure 1.

We can represent our rules mathematically. Each cell  $j$  is assigned a value  $a_j$  of 0 ('dead') or 1 ('alive'). The quantity

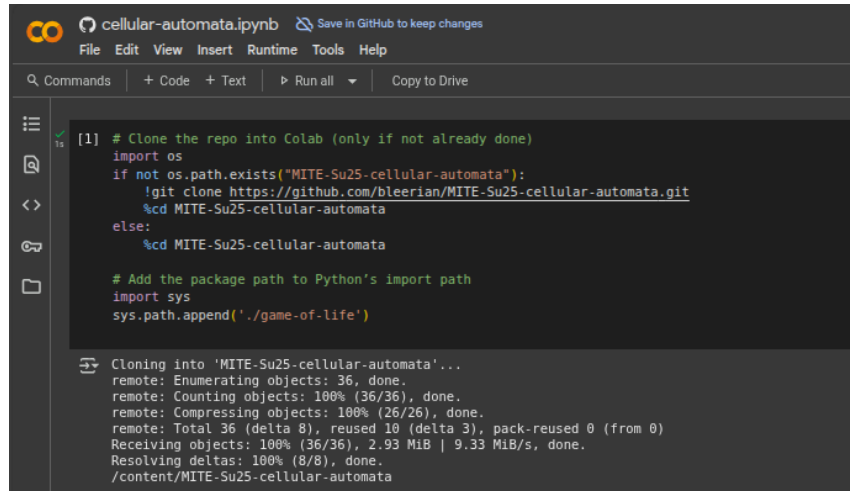
$$A_i = \sum_{j=1}^8 a_j$$

represents a sum over all neighboring cell values and is calculated at each time step to determine the value of the central cell in the next time step. Since each cell has a value of 0 or 1,  $A_i$  represents the number of neighboring cells that are alive. The rules for the Game of Life can thus be mathematically represented as

$$\begin{aligned} a_i(t+\Delta t) &= 0 & \text{if } A_i(t) > 3 \\ a_i(t+\Delta t) &= 1 & \text{if } A_i(t) = 3 \\ a_i(t+\Delta t) &= a_i(t) & \text{if } A_i(t) = 2 \\ a_i(t+\Delta t) &= 0 & \text{if } A_i(t) < 2 \end{aligned}$$

## About the code

This tutorial assumes you are working from the Google Colab environment setup earlier. Your screen should look something like the following:



```
[1] # Clone the repo into Colab (only if not already done)
import os
if not os.path.exists("MITE-Su25-cellular-automata"):
    !git clone https://github.com/bleerian/MITE-Su25-cellular-automata.git
    %cd MITE-Su25-cellular-automata
else:
    %cd MITE-Su25-cellular-automata

# Add the package path to Python's import path
import sys
sys.path.append('./game-of-life')
```

Cloning into 'MITE-Su25-cellular-automata'...

remote: Enumerating objects: 36, done.

remote: Counting objects: 100% (36/36), done.

remote: Compressing objects: 100% (26/26), done.

remote: Total 36 (delta 8), reused 10 (delta 3), pack-reused 0 (from 0)

Receiving objects: 100% (36/36), 2.93 MiB | 9.33 MiB/s, done.

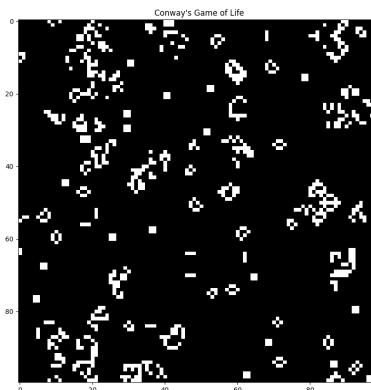
Resolving deltas: 100% (8/8), done.

/content/MITE-Su25-cellular-automata

In this series of activities, we will be working directly in this Python notebook. First begin by installing all of the necessary libraries by running the first code chunk labeled: “Clone the repo into Colab”

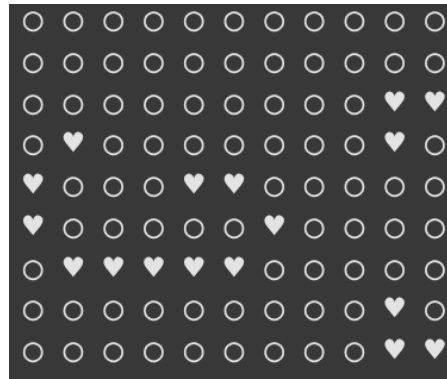
## Animating Conway’s Game of Life Using Python

After running the first code chunk, all of the necessary libraries should be loaded. In this section, we can see how Conway’s Game of Life can be coded in Python and animated. This first code chunk utilizes libraries such as numpy, matplotlib, and IPython to handle arrays of positions, visualize, and animate the famous Conway’s Game of Life. Run this code a couple of times and visualize some of the different preloaded patterns.



## Understanding the code

The code will build an animation for Conway's Game of Life that will display in the terminal from a set of available patterns. It will look something like the following, where the 'alive' cells are hearts and 'dead' cells are points:



Before we can build a visualization, we need to build the backend implementation.

1. In `pattern.py`, we define the pattern of 'alive' cells. We will use a list of tuples to represent 'alive' cells, in which their index coordinate is contained in `alive_cells` and the name of the pattern is contained in `name`.

```
from dataclasses import dataclass

@dataclass
class Pattern:
    name: str # name and alive_cells are
    alive_cells: set[tuple[int, int]] # placeholders for now
```

2. The next step is to implement how the grid will evolve with each iteration. This will be done with the `LifeGrid.evolve(pattern)` method in `grid.py`, which takes as input an instance of the `class Pattern`.

We first need a way to track the value of each cell. Instead of tracking the value of each cell individually, we will track just the coordinates of the 'alive' cells as a Python set of tuples (see the documentation for information on different types of data structures like sets and tuples: <https://docs.python.org/3/tutorial/datastructures.html>).

We also do not have to store coordinates of every cell in the whole grid. We take advantage of the fact that the only thing we care about is knowing the values of the neighboring cells to 'alive' cells, which correspond to the cases where the cell value changes between time steps.

For the Moore neighborhood and taking the central cell as reference coordinate (0,0), we can define the position of each neighbor as a *difference coordinate* from (0,0) in a Cartesian-like coordinate system, as shown below. We leave a few cells unspecified and leave as an exercise to be defined by you later.

	(0, 1)	
(-1, 0)	(0,0)	(1, 0)
	(0, -1)	

**Exercise:** Let's get a feel for how **Pattern** and **Grid** work together. Import the `patterns.py` and `grid.py` modules into this notebook as libraries, then load a "Blinker" pattern

```
# Load the pattern
blinker = patterns.get_pattern("Blinker")

# Create a grid with it
grid = grid_module.LifeGrid(blinker)

# Print the initial state
print("Generation 1 -", grid)

# Evolve the grid
grid.evolve()

# Print the updated pattern
print("Generation 2 -", grid) # same object; now updated

grid.evolve()

print("Generation 3 -", grid)
```

What is the output from this code snippet?

**Bonus:** Take a closer look at the implementation of the `evolve()` method in `grid.py`, specifically lines 18–29 (You can open up `grid.py` in the left hand folder navigation menu). This section calculates which cells will be "alive" in the next time step using a `defaultdict` to count neighbors and set operations to determine which cells stay alive or come alive. Try to follow how the neighbor counting works and how `stay_alive` and `come_alive` are computed. Inline comments have been provided to guide you through the logic. If anything is unclear, feel free to ask a helper or talk it over with your peers.

----

Great! At this point, we have implemented a way to identify all the neighbors of relevant cells for each time step and a procedure to propagate the values of each cell for each time step/iteration. When monitoring the grid with `print(grid)`, we see the name of the **Pattern** we created when setting up the grid and the set of tuples that tracks the 'alive' cells. This completes the implementation for the Game of Life. What remains is implementing a way to visualize each iteration.

3. In `grid.py`, the grid containing 'alive' and 'dead' cells will be represented as a string. We first need to define the grid size. In theory, the grid is infinite, but this is not possible to do on any computer (finite memory!). Instead, we will focus on a chunk of the grid and check that it is big enough for the pattern we are interested in.

The method `LifeGrid.as_string(bbox)` contains the way we can translate our set of tuples into a string that can be printed to the terminal for visualization. The variable `bbox` contains the dimensions of the grid and is itself also a tuple. Here, the cells in each row of the grid is represented with "♥" if the cell's coordinates are in the set of `pattern.alive_cells` and with "-" otherwise.

**Exercise:** Let's see what the string representation of the grid looks like for our example of the Blinker pattern. Move on to the next section and run the next code chunk

```
# Instantiate the grid with the pattern
life = grid_module.LifeGrid(blinker)

# Show initial state
print(life)

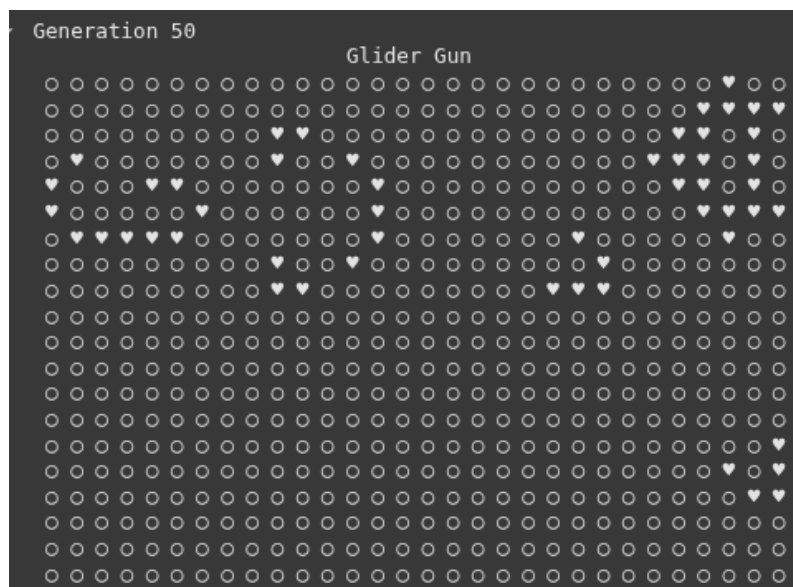
life.evolve() #This evolves each step to the next generation)
print(life.as_string((0, 0, 5, 5))) #Sets the bounding box: (start_row,
start_col, end_row, end_col)
life.evolve()
print(life.as_string((0, 0, 5, 5)))
```

This snippet of code instantiates the grid with a pattern called "blinker" then shows what it looks like using the `print` command. We can then evolve to see the next generation using the `life.evolve()` command. By repeating this process, we can continue to load newer generations of this pattern.

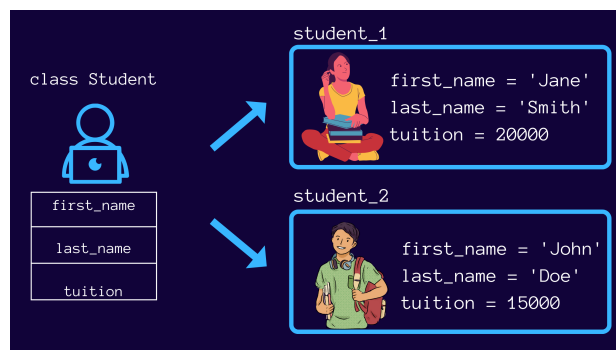
**Bonus:** The `toml` file is handled by the `tomllib` package (for Python >3.11) and its method is implemented into `Patterns.from_toml(...)`. You can retrieve all the patterns from `patterns.toml` with:

4. To visualize and animate the grid at each time step, we use the `IPython.display` module alongside `time.sleep` to create simple, notebook-friendly animations. A `LifeGrid` object is evolved over multiple generations and printed at regular intervals, allowing you to observe how the system changes over time. This connects the visual representation of the Game of Life with the simulation logic defined in `grid.py`. You can test this using patterns like the Glider Gun by calling the `notebook_view` function.

**Exercise:** Let's see how the visualization works. Run the code chunk where "`notebook_view`" is defined. You should see an animation like the image below. You can play around with different patterns and see how they are animated.



**Bonus:** The structure of the code uses [Python Classes](#) to bundle our user-defined objects with user-defined attributes. Classes are useful as templates for instantiating several of the same object. For example, consider the class `Student`, which has attributes `first_name`, `last_name`, and `tuition`. Then you can take `class Student` and instantiate two separate instances with specific information for each attribute, as found below. We make use of this kind of structure for both the Pattern of 'alive' cells and for the Grid on which the 'alive' cells reside.



## Modifying the code

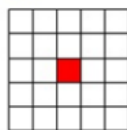
**Exercise:** Fill in the appropriate fields for the defining a neighborhood by changing the “custom\_neighbors” variable

```
# Step 1: Define the neighborhood
custom_neighbors = [
    (-1, 0), # top
    ( 0, -1), (0, 1), # left/right
    ( 1, 0), # bottom
]
```

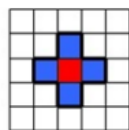
**Exercise:** Play around with different neighbor definitions. How do the generations of different patterns change with different neighbor values?

**Bonus Exercise:** Thus far, we have dealt with a specific version of the Moore neighborhood for updating the cell value where nearest neighbors are included. Several choices of the neighborhood are possible. A few common local environments are shown in Figure 2. Modify the code with different choices of neighborhood. Compare and contrast your observations with the nearest-neighbors Moore environment.

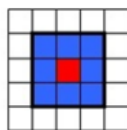
Neighbourhoods



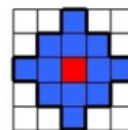
Empty  
 $N = 0$



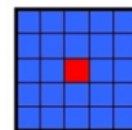
Von Neumann  
 $N = 4$



Moore  
 $N = 8$



MvonN  
 $N = 12$



Moore expanded  
 $N = 24$

Figure 2

**Exercise:** Create your own patterns and see how they evolve over time.

**Bonus Exercise:** Create your own patterns and implement them into `pattern.toml` and visualize it using

```
my_pattern_from_toml = patterns.get_pattern("My Pattern")
notebook_view(my_pattern_from_toml, generations=10, bbox=(0, 0, 10, 10), delay = 0.1)
```



MITE 2025 Summer  
Cellular Automata Demonstration

Wang Materials Group  
UT Austin