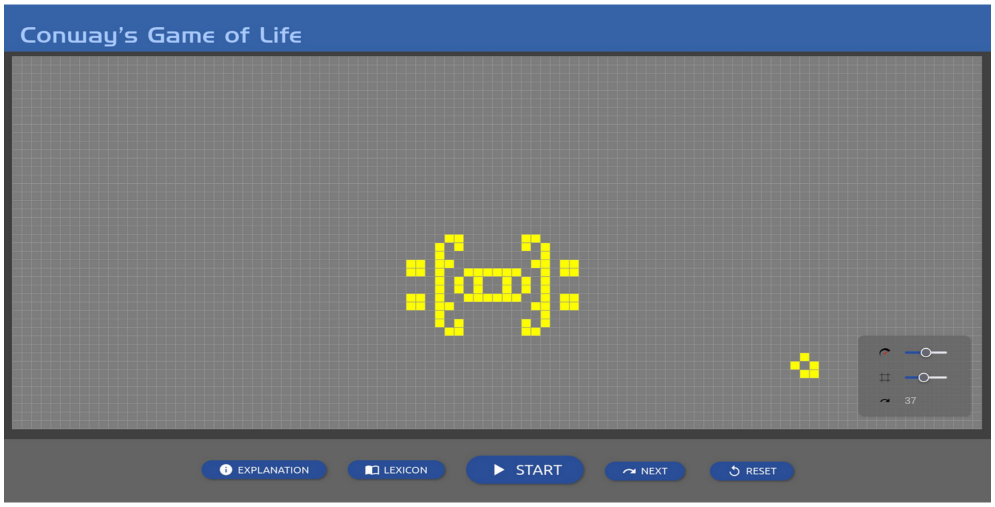
Part 1: Conway's Game of Life

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 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).

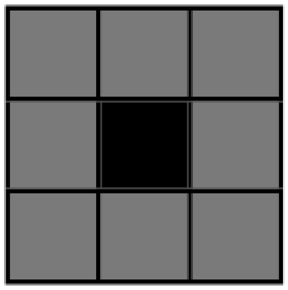


Figure 1: The Moore environment where each cell has eight neighbors (gray cells).

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

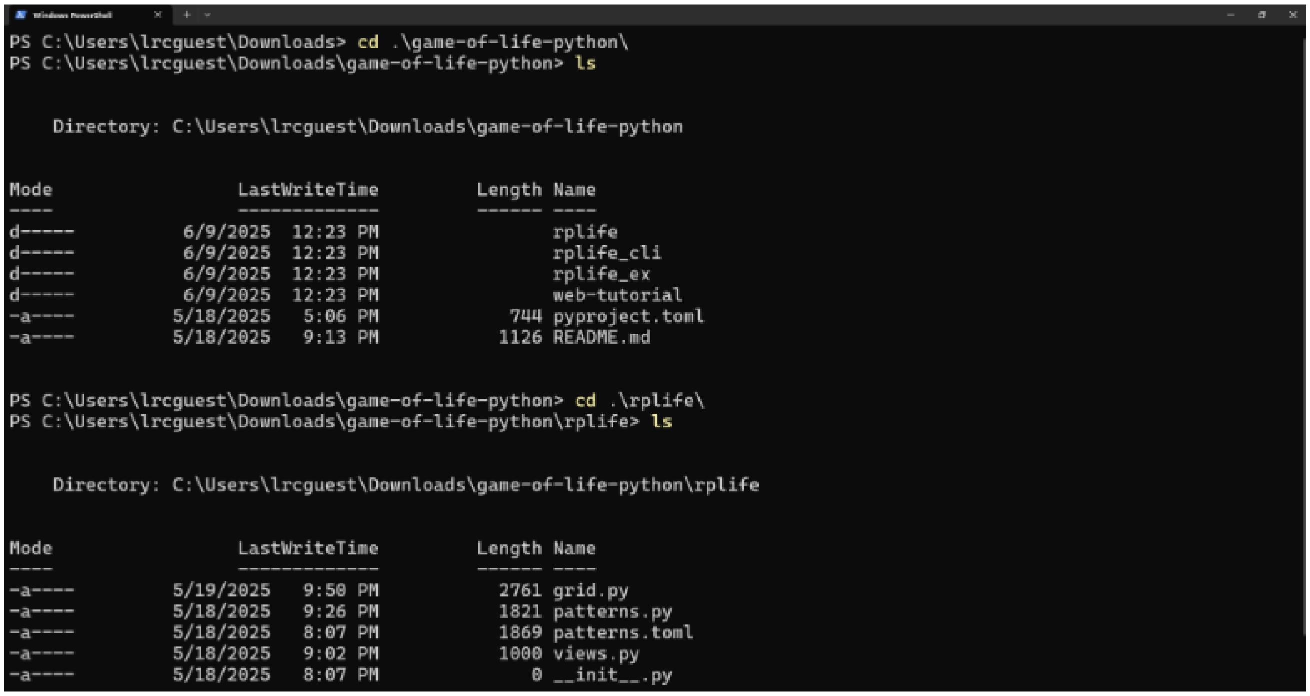
$$A_i = \sum_{j=1}^{8} a_j$$

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

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

About the code

This tutorial assumes you are working from the Downloads folder. Your terminal environment will look something like the following.



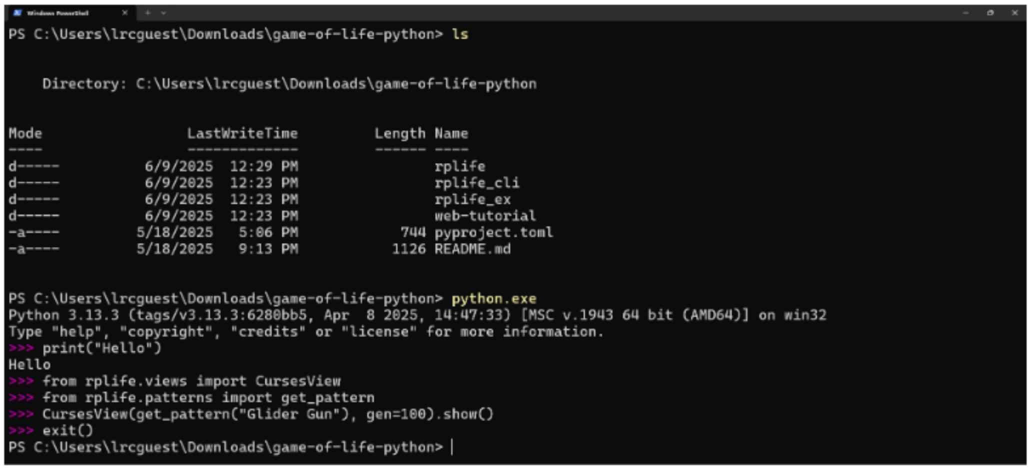
```
PS C:\Users\lrcguest\Downloads> cd .\game-of-life-python\
PS C:\Users\lrcguest\Downloads\game-of-life-python> ls


    Directory: C:\Users\lrcguest\Downloads\game-of-life-python


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          6/9/2025  12:23 PM                rplife
d-----          6/9/2025  12:23 PM                rplife_cli
d-----          6/9/2025  12:23 PM                rplife_ex
d-----          6/9/2025  12:23 PM                web-tutorial
-a----         5/18/2025   5:06 PM            744 pyproject.toml
-a----         5/18/2025   9:13 PM           1126 README.md


PS C:\Users\lrcguest\Downloads\game-of-life-python> cd .\rplife\
PS C:\Users\lrcguest\Downloads\game-of-life-python\rplife> ls


    Directory: C:\Users\lrcguest\Downloads\game-of-life-python\rplife


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
-a----         5/19/2025   9:50 PM           2761 grid.py
-a----         5/18/2025   9:26 PM           1821 patterns.py
-a----         5/18/2025   8:07 PM           1869 patterns.toml
-a----         5/18/2025   9:02 PM           1000 views.py
-a----         5/18/2025   8:07 PM              0 __init__.py
```

In this series of activities, we will be working from the `rplife` and `rplife_ex` directories.

Bonus: The `rplife_cli` folder contains additional python scripts for creating a command line interface (cli) where you can run the game of life directly from the terminal instead of through an interactive Python session.

Important! For editing files, you will want to access the `rplife` and `rplife_ex` directories.

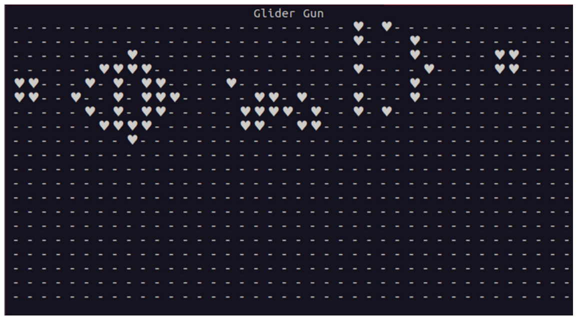
For running the python code, you will want to be in the `.\game-of-life-python\` directory.



```
PS C:\Users\lrcguest\Downloads\game-of-life-python> ls


    Directory: C:\Users\lrcguest\Downloads\game-of-life-python


Mode                 LastWriteTime         Length Name
----                 -------------         ------ ----
d-----          6/9/2025  12:29 PM                rplife
d-----          6/9/2025  12:23 PM                rplife_cli
d-----          6/9/2025  12:23 PM                rplife_ex
d-----          6/9/2025  12:23 PM                web-tutorial
-a----         5/18/2025   5:06 PM            744 pyproject.toml
-a----         5/18/2025   9:13 PM           1126 README.md


PS C:\Users\lrcguest\Downloads\game-of-life-python> python.exe
Python 3.13.3 (tags/v3.13.3:6280bb5, Apr  8 2025, 14:47:33) [MSC v.1943 64 bit (AMD64)] on win32
Type "help", "copyright", "credits" or "license" for more information.
>>> print("Hello")
Hello
>>> from rplife.views import CursesView
>>> from rplife.patterns import get_pattern
>>> CursesView(get_pattern("Glider Gun"), gen=100).show()
>>> exit()
PS C:\Users\lrcguest\Downloads\game-of-life-python> |
```

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. Open up a Python in the terminal and type or copy/paste the following prompts (indicated with ">>>"). You should see output that looks like the gray text.

```
from rplife import grid, patterns
blinker = patterns.Pattern("Blinker", {(2, 1), (2, 2), (2, 3)})
grid = grid.LifeGrid(blinker)
print(grid)
 Blinker:
 Alive cells -> [(2, 1), (2, 2), (2, 3)]
grid.evolve()
print(grid)
 Blinker:
 Alive cells -> [(1, 2), (2, 2), (3, 2)]
grid.evolve()
print(grid)
 Blinker:
 Alive cells -> [(2, 1), (2, 2), (2, 3)]
```

Tip: Exit out of Python by typing "`exit()`"

Bonus: Take a closer look at the implementation for determining the set of 'alive' cells in the next time step in lines 32 – 53 in `grid.py` and see if you can follow the logic of operations. Some in-line comments have been included to help you along. Feel free to also ask a helper or 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. Open up a Python in the terminal and type or copy/paste the following prompts. You should see output that looks like the gray text.

```
from rplife import grid, patterns

blinker = patterns.Pattern("Blinker", {(2, 1), (2, 2), (2, 3)})

grid = grid.LifeGrid(blinker)

print(grid.as_string((0, 0, 5, 5)))
 Blinker
  •  •  •  •  •
  •  •  •  •  •
  •  ♥  ♥  ♥  •
  •  •  •  •  •
  •  •  •  •  •

grid.evolve()
print(grid.as_string((0, 0, 5, 5)))
 Blinker
  •  •  •  •  •
  •  •  ♥  •  •
  •  •  ♥  •  •
  •  •  ♥  •  •
  •  •  •  •  •
```

4. Instead of having to instantiate the Blinker pattern every time, we can save a separate file that contains a set of pre-defined patterns. This is the `patterns.toml` files.

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:

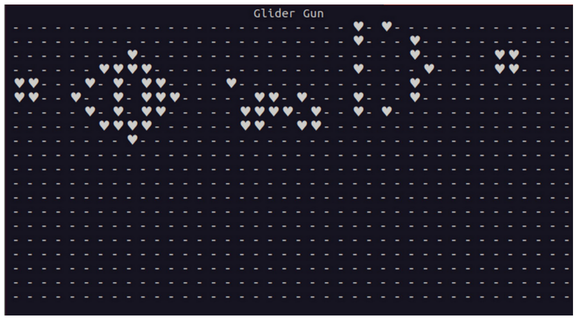
```
from rplife import patterns
patterns.get_pattern("Blinker")
 Pattern(name='Blinker', alive_cells={(2, 3), (2, 1), (2, 2)})
patterns.get_all_patterns()
 [
     Pattern(name='Blinker', alive_cells={(2, 3), (2, 1), (2, 2)}),
     ...
 ]
```

5. To visualize and animate the string representation of our grid at each time step, the code makes use of a package called `curses`, which was created for generating character-cell display terminals. The implementation is found in `views.py`. `LifeGrid` is imported into `views.py` and evolved for each generation; this connects the terminal-based rendering of the grid with the back-end implementation of the Game of Life.

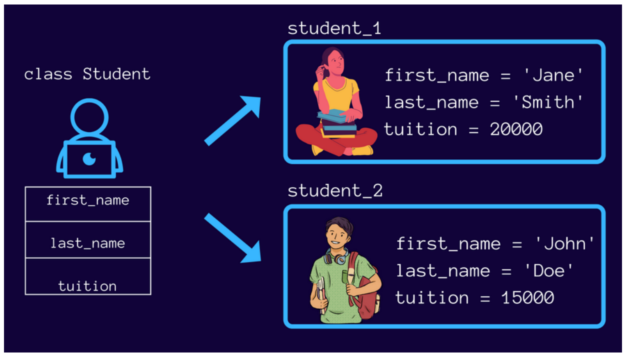
Exercise: Let's see how the visualization works. Open up a Python in the terminal and type or copy/paste the following prompts. You should see output that looks like the gray text.

```
from rplife.views import CursesView
from rplife.patterns import get_pattern

CursesView(get_pattern("Glider Gun"), gen=100).show()
```

In the last command, you request 100 iterations of the pattern "Glider Gun" using the `CursesView` class. You should see something that looks like the following:



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 missing neighbors in `rplife_ex/grid.py`.

Exercise: Play around with the code.

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.

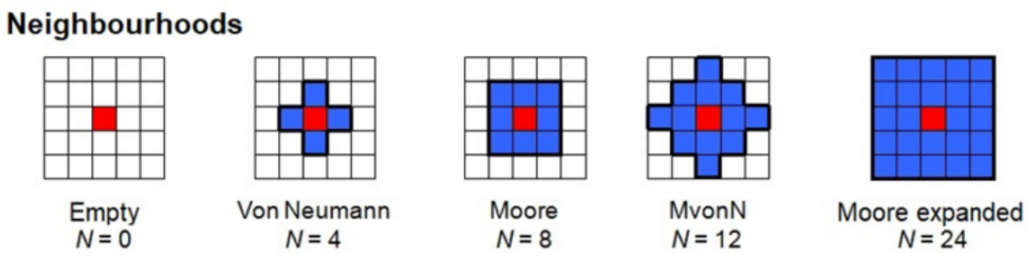


Figure 2

Bonus Exercise: Create your own patterns and implement them into `pattern.toml`. A blank square grid is provided for sketching out your ideas.