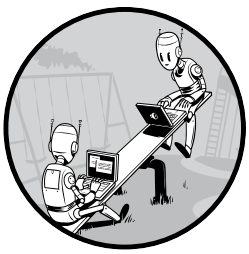
3

CONWAY'S GAME OF LIFE



You can use a computer to study a system by creating a mathematical model for that system, writing a program to represent the model, and then letting the model evolve over time. There are many kinds of computer simulations, but I'll focus on a famous one called Conway's Game of Life, the work of the British mathematician John Conway. The Game of Life is an example of a *cellular automaton*, a collection of colored cells on a grid that evolve through a number of time steps according to a set of rules defining the states of neighboring cells.

In this project, you'll create an $N \times N$ grid of cells and simulate the evolution of the system over time by applying the rules of Conway's Game of Life. You'll display the state of the game at each time step and provide ways to save the output to a file. You'll set the initial condition of the system to either a random distribution or a predesigned pattern.

This simulation consists of the following components:

- A property defined in one- or two-dimensional space
- A mathematical rule to change this property for each step in the simulation
- A way to display or capture the state of the system as it evolves

The cells in Conway's Game of Life can be either ON or OFF. The game starts with an initial condition, in which each cell is assigned one state and mathematical rules determine how its state will change over time. The amazing thing about Conway's Game of Life is that with just four simple rules the system evolves to produce patterns that behave in incredibly complex ways, almost as if they were alive. Patterns include "gliders" that slide across the grid, "blinkers" that flash on and off, and even replicating patterns.

Of course, the philosophical implications of this game are also significant, because they suggest that complex structures can evolve from simple rules without following any sort of preset pattern.

Here are some of the main concepts covered in this project:

- Using `matplotlib imshow` to represent a 2D grid of data
- Using `matplotlib` for animation
- Using the `numpy` array
- Using the `%` operator for boundary conditions
- Setting up a random distribution of values

How It Works

Because the Game of Life is built on a grid of nine squares, every cell has eight neighboring cells, as shown in Figure 3-1. A given cell (i, j) in the simulation is accessed on a grid $[i][j]$, where i and j are the row and column indices, respectively. The value of a given cell at a given instant of time depends on the state of its neighbors at the previous time step.

Conway's Game of Life has four rules.

1. If a cell is ON and has fewer than two neighbors that are ON, it turns OFF.
2. If a cell is ON and has either two or three neighbors that are ON, it remains ON.
3. If a cell is ON and has more than three neighbors that are ON, it turns OFF.
4. If a cell is OFF and has exactly three neighbors that are ON, it turns ON.

These rules are meant to mirror some basic ways that a group of organisms might fare over time: underpopulation and overpopulation kill cells by turning a cell OFF when it has fewer than two neighbors or more than three, and cells stay ON and reproduce by turning another cell from OFF to ON when the population is balanced. But what about cells at the edge of the grid? Which cells are their neighbors? To answer this question, you need to think about *boundary conditions*, the rules that govern what happens to cells at the edges or boundaries of the grid. I'll address this question by using *toroidal boundary conditions*, meaning that the square grid wraps around so that its shape is a torus. As shown in Figure 3-2, the grid is first warped so that its horizontal edges (A and B) join to form a cylinder, and then the cylinder's vertical edges (C and D) are joined to form a torus. Once the torus has been formed, all cells have neighbors because the whole space has no edge.

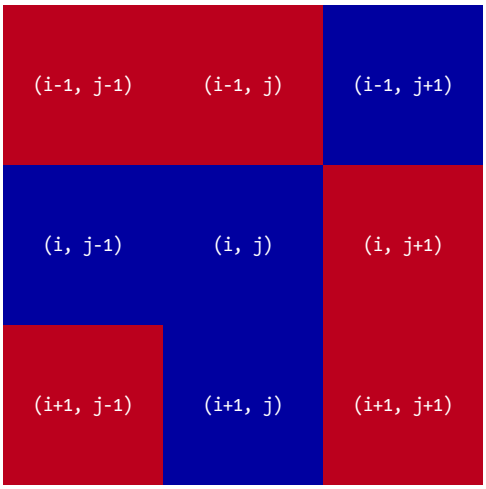


Figure 3-1: Eight neighboring cells

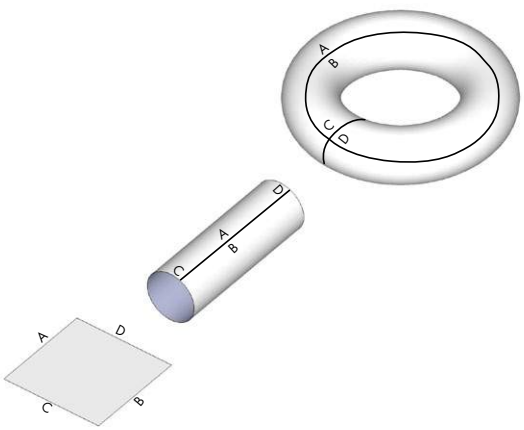


Figure 3-2: Conceptual visualization of toroidal boundary conditions

NOTE *This is similar to how boundaries work in Pac-Man. If you go off the top of the screen, you appear on the bottom. If you go off the left side of the screen, you appear on the right side. This kind of boundary condition is common in 2D simulations.*

Here's a description of the algorithm you'll use to apply the four rules and run the simulation:

1. Initialize the cells in the grid.
2. At each time step in the simulation, for each cell (i, j) in the grid, do the following:
   a. Update the value of cell (i, j) based on its neighbors, taking into account the boundary conditions.
   b. Update the display of grid values.

Requirements

You'll use `numpy` arrays and the `matplotlib` library to display the simulation output, and you'll use the `matplotlib animation` module to update the simulation. (See Chapter 1 for a review of `matplotlib`.)

The Code

You'll develop the code for the simulation bit by bit inside the Python interpreter by examining the pieces needed for different parts. To see the full project code, skip ahead to "The Complete Code" on page 49.

First, import the modules you'll be using for this project:

```
>>> import numpy as np
>>> import matplotlib.pyplot as plt
>>> import matplotlib.animation as animation
```

Now let's create the grid.

Representing the Grid

To represent whether a cell is alive (ON) or dead (OFF) on the grid, you'll use the values `255` and `0` for ON and OFF, respectively. You'll display the current state of the grid using the `imshow()` method in `matplotlib`, which represents a matrix of numbers as an image. Enter the following:

```
❶ >>> x = np.array([[0, 0, 255], [255, 255, 0], [0, 255, 0]])
❷ >>> plt.imshow(x, interpolation='nearest')
plt.show()
```

At ❶, you define a 2D `numpy` array of shape (3, 3), where each element of the array is an integer value. You then use the `plt.show()` method to display this matrix of values as an image, and you pass in the interpolation option as `'nearest'` at ❷ to get sharp edges for the cells (or they'd be fuzzy).

Figure 3-3 shows the output of this code.

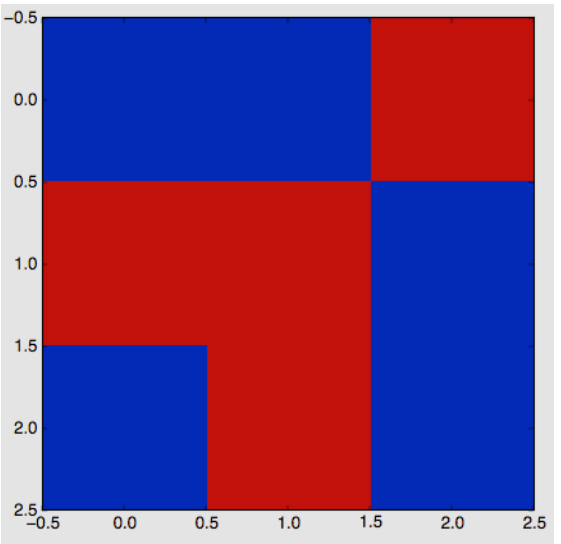


Figure 3-3: Displaying a grid of values

Notice that the value of `0` (OFF) is shown in dark gray and `255` (ON) is shown in light gray, which is the default colormap used in `imshow()`.

Initial Conditions

To begin the simulation, set an initial state for each cell in the 2D grid. You can use a random distribution of ON and OFF cells and see what kind of patterns emerge, or you can add some specific patterns and see how they evolve. You'll look at both approaches.

To use a random initial state, use the `choice()` method from the `random` module in `numpy`. Enter the following:

```
np.random.choice([0, 255], 4*4, p=[0.1, 0.9]).reshape(4, 4)
```

Here is the output:

```
array([[255, 255, 255, 255],
       [255, 255, 255, 255],
       [255, 255, 255, 255],
       [255, 255, 255,   0]])
```

`np.random.choice` chooses a value from the given list `[0, 255]`, with the probability of the appearance of each value given in the parameter `p=[0.1, 0.9]`. Here, you ask for 0 to appear with a probability of 0.1 (or 10 percent) and for 255 to appear with a probability of 90 percent. (The two values in `p` must add up to 1.) Because this `choice()` method creates a one-dimensional array of 16 values, you use `.reshape` to make it a two-dimensional array.

To set up the initial condition to match a particular pattern instead of just filling in a random set of values, initialize the two-dimensional grid to zeros and then use a method to add a pattern at a particular row and column in the grid, as shown here:

```
def addGlider(i, j, grid):
    """adds a glider with top left cell at (i, j)"""
❶   glider = np.array([[0, 0, 255],
                       [255, 0, 255],
                       [0, 255, 255]])
❷   grid[i:i+3, j:j+3] = glider
❸ grid = np.zeros(N*N).reshape(N, N)
❹ addGlider(1, 1, grid)
```

At ❶, you define the glider pattern (an observed pattern that moves steadily across the grid) using a `numpy` array of shape (3, 3). At ❷, you can see how you use the `numpy` slice operation to copy this pattern array into the simulation's two-dimensional grid, with its top-left corner placed at the coordinates you specify as `i` and `j`. You create an $N \times N$ array of zeros at ❸, and at ❹, you call the `addGlider()` method to initialize the grid with the glider pattern.

Boundary Conditions

Now we can think about how to implement the toroidal boundary conditions. First, let's see what happens at the right edge of a grid of size $N \times N$. The cell at the end of row i is accessed as `grid[i][N-1]`. Its neighbor to the right is `grid[i][N]`, but according to the toroidal boundary conditions, the value accessed as `grid[i][N]` should be replaced by `grid[i][0]`. Here's one way to do that:

```
if j == N-1:
    right = grid[i][0]
else:
    right = grid[i][j+1]
```

Of course, you'd need to apply similar boundary conditions to the left, top, and bottom sides of the grid, but doing so would require adding a lot more code because each of the four edges of the grid would need to be tested. A much more compact way to accomplish this is with Python's modulus (`%`) operator, as shown here:

```
>>> N = 16
>>> i1 = 14
>>> i2 = 15
>>> (i1+1)%N
15
>>> (i2+1)%N
0
```

As you can see, the `%` operator gives the remainder for the integer division by `N`. You can use this operator to make the values wrap around at the edge by rewriting the grid access code like this:

```
right = grid[i][(j+1)%N]
```

Now when a cell is on the edge of the grid (in other words, when `j = N-1`), asking for the cell to the right with this method will give you `(j+1)%N`, which sets `j` back to `0`, making the right side of the grid wrap to the left side. When you do the same for the bottom of the grid, it wraps around to the top.

Implementing the Rules

The rules of the Game of Life are based on the number of neighboring cells that are ON or OFF. To simplify the application of these rules, you can calculate the total number of neighboring cells in the ON state. Because the ON states have a value of 255, you can just sum the values of all the neighbors and divide by 255 to get the number of ON cells. Here is the relevant code:

```
            # apply Conway's rules
            if grid[i, j] == ON:
❶               if (total < 2) or (total > 3):
                    newGrid[i, j] = OFF
            else:
                if total == 3:
❷                   newGrid[i, j] = ON
```

At ❶, any cell that is ON is turned OFF if it has fewer than two neighbors that are ON or if it has more than three neighbors that are ON. The code at ❷ applies only to OFF cells: a cell is turned ON if exactly three neighbors are ON.

Now it's time to write the complete code for the simulation.

Sending Command Line Arguments to the Program

The following code sends command line arguments to your program:

```
# main() function
def main():
    # command line argumentss are in sys.argv[1], sys.argv[2], ...
    # sys.argv[0] is the script name and can be ignored
    # parse arguments
❶   parser = argparse.ArgumentParser(description="Runs Conway's Game of Life
        simulation.")
    # add arguments
❷   parser.add_argument('--grid-size', dest='N', required=False)
❸   parser.add_argument('--mov-file', dest='movfile', required=False)
❹   parser.add_argument('--interval', dest='interval', required=False)
❺   parser.add_argument('--glider', action='store_true', required=False)
    args = parser.parse_args()
```

The `main()` function begins by defining command line parameters for the program. You use the `argparse` class at ❶ to add command line options to the code, and then you add various options to it in the following lines. At ❷, you specify the simulation grid size *N*, and at ❸, you specify the filename for the saved *.mov* file. At ❹, you set the animation update interval in milliseconds, and at ❺, you start the simulation with a glider pattern.

Initializing the Simulation

Continuing through the code, you come to the following section, which initializes the simulation:

```
    # set grid size
    N = 100
    if args.N and int(args.N) > 8:
        N = int(args.N)

    # set animation update interval
    updateInterval = 50
    if args.interval:
        updateInterval = int(args.interval)

    # declare grid
❶   grid = np.array([])
    # check if "glider" demo flag is specified
    if args.glider:
        grid = np.zeros(N*N).reshape(N, N)
        addGlider(1, 1, grid)
    else:
        # populate grid with random on/off - more off than on
        grid = randomGrid(N)
```

Still within the `main()` function, this portion of the code applies any parameters called at the command line, once the command line options have been parsed. For example, the lines that follow ❶ set up the initial conditions, either a random pattern by default or a glider pattern.

Finally, you set up the animation.

```
    # set up the animation
❶   fig, ax = plt.subplots()
    img = ax.imshow(grid, interpolation='nearest')
❷   ani = animation.FuncAnimation(fig, update, fargs=(img, grid, N, ),
                                  frames=10,
                                  interval=updateInterval,
                                  save_count=50)

    # number of frames?
    # set the output file
    if args.movfile:
        ani.save(args.movfile, fps=30, extra_args=['-vcodec', 'libx264'])

    plt.show()
```

At ❶, you configure the `matplotlib` plot and animation parameters. At ❷, `animation.FuncAnimation()` calls the function `update()`, defined earlier in the program, which updates the grid according to the rules of Conway's Game of Life using toroidal boundary conditions.

The Complete Code

Here is the complete program for your Game of Life simulation. You can also download the code for this project from *https://github.com/electronut/pp/blob/master/conway/conway.py*.

```
import sys, argparse
import numpy as np
import matplotlib.pyplot as plt
import matplotlib.animation as animation

ON = 255
OFF = 0
vals = [ON, OFF]

def randomGrid(N):
    """returns a grid of NxN random values"""
    return np.random.choice(vals, N*N, p=[0.2, 0.8]).reshape(N, N)

def addGlider(i, j, grid):
    """adds a glider with top-left cell at (i, j)"""
    glider = np.array([[0,    0, 255],
                       [255,  0, 255],
                       [0,  255, 255]])
    grid[i:i+3, j:j+3] = glider

def update(frameNum, img, grid, N):
    # copy grid since we require 8 neighbors for calculation
    # and we go line by line
    newGrid = grid.copy()
    for i in range(N):
        for j in range(N):
            # compute 8-neghbor sum using toroidal boundary conditions
            # x and y wrap around so that the simulation
            # takes place on a toroidal surface
            total = int((grid[i, (j-1)%N] + grid[i, (j+1)%N] +
                         grid[(i-1)%N, j] + grid[(i+1)%N, j] +
                         grid[(i-1)%N, (j-1)%N] + grid[(i-1)%N, (j+1)%N] +
                         grid[(i+1)%N, (j-1)%N] + grid[(i+1)%N, (j+1)%N])/255)
            # apply Conway's rules
            if grid[i, j] == ON:
                if (total < 2) or (total > 3):
                    newGrid[i, j] = OFF
            else:
                if total == 3:
                    newGrid[i, j] = ON
```

```
    # update data
    img.set_data(newGrid)
    grid[:] = newGrid[:]
    return img,

# main() function
def main():
    # command line arguments are in sys.argv[1], sys.argv[2], ...
    # sys.argv[0] is the script name and can be ignored
    # parse arguments
    parser = argparse.ArgumentParser(description="Runs Conway's Game of Life
        simulation.")
  # add arguments
    parser.add_argument('--grid-size', dest='N', required=False)
    parser.add_argument('--mov-file', dest='movfile', required=False)
    parser.add_argument('--interval', dest='interval', required=False)
    parser.add_argument('--glider', action='store_true', required=False)
    parser.add_argument('--gosper', action='store_true', required=False)
    args = parser.parse_args()

    # set grid size
    N = 100
    if args.N and int(args.N) > 8:
        N = int(args.N)

    # set animation update interval
    updateInterval = 50
    if args.interval:
        updateInterval = int(args.interval)

    # declare grid
    grid = np.array([])
    # check if "glider" demo flag is specified
    if args.glider:
        grid = np.zeros(N*N).reshape(N, N)
        addGlider(1, 1, grid)
    else:
        # populate grid with random on/off - more off than on
        grid = randomGrid(N)

    # set up the animation
    fig, ax = plt.subplots()
    img = ax.imshow(grid, interpolation='nearest')
    ani = animation.FuncAnimation(fig, update, fargs=(img, grid, N, ),
                                  frames=10,
                                  interval=updateInterval,
                                  save_count=50)

    # number of frames?
    # set the output file
    if args.movfile:
        ani.save(args.movfile, fps=30, extra_args=['-vcodec', 'libx264'])

    plt.show()
```

```
# call main
if __name__ == '__main__':
    main()
```

Running the Game of Life Simulation

Now run the code:

```
$ python3 conway.py
```

This uses the default parameters for the simulation: a grid of 100×100 cells and an update interval of 50 milliseconds. As you watch the simulation, you'll see how it progresses to create and sustain various patterns over time, as in Figure 3-4.

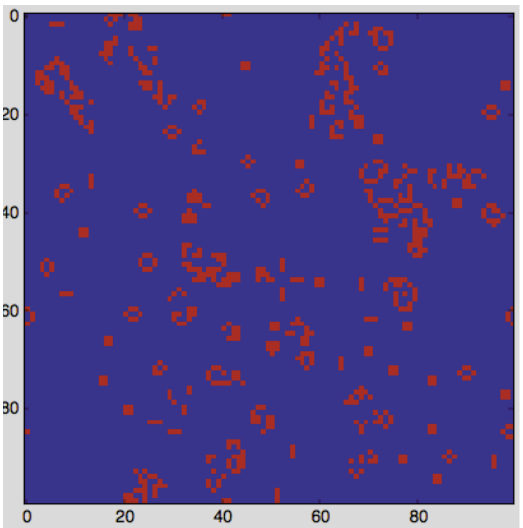


Figure 3-4: Game of Life in progress

Figure 3-5 shows some of the patterns to look for in the simulation. Besides the glider, look for a three-cell blinker and static patterns such as a block or loaf shape.

Now change things up a bit by running the simulation with these parameters:

```
$ python conway.py --grid-size 32 --interval 500 --glider
```

This creates a simulation grid of 32×32, updates the animation every 500 milliseconds, and uses the initial glider pattern shown in the bottom right of Figure 3-5.

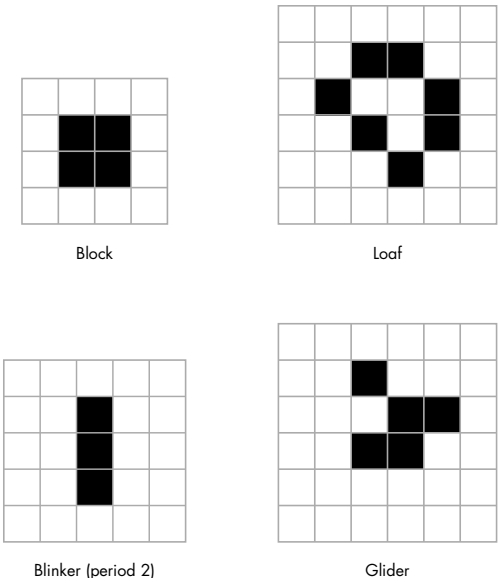


Figure 3-5: Patterns in Game of Life

Summary

In this project, you explored Conway's Game of Life. You learned how to set up a basic computer simulation based on some rules and how to use `matplotlib` to visualize the state of the system as it evolves.

My implementation of Conway's Game of Life emphasizes simplicity over performance. You can speed up the computations in Game of Life in many different ways, and a tremendous amount of research has been done on how to do this. You'll find a lot of this research with a quick Internet search.

Experiments!

Here are some ways to experiment further with Conway's Game of Life.

1. Write an `addGosperGun()` method to add the pattern shown in Figure 3-6 to the grid. This pattern is called the *Gosper Glider Gun*. Run the simulation and observe what the gun does.

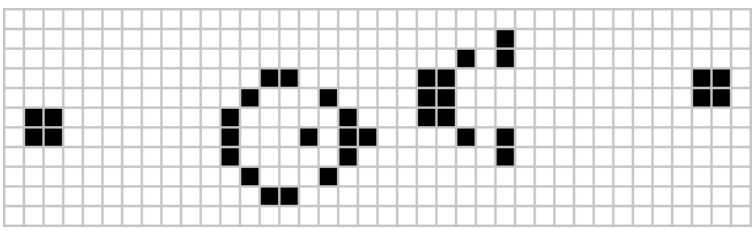


Figure 3-6: Gosper Glider Gun

2. Write a `readPattern()` method that reads in an initial pattern from a text file and uses it to set the initial conditions for the simulation. Here is a suggested format for this file:

```
8
0 0 0 255 ...
```

The first line of the file defines *N*, and the rest of the file is just *N*×*N* integers (0 or 255) separated by whitespace. You can use Python methods such as `open` and `file.read` to do this. This exploration will help you study how any given pattern evolves with the rules of the Game of Life. Add a `--pattern-file` command line option to use this file while running the program.