CA Workbook: Cellular Automata and the Game of Life

Dr. Yves J. Hilpisch[1]

December 11, 2025

[1]Get in touch: `https://linktr.ee/dyjh`. Web page `https://hilpisch.com`. Research, structuring, drafting, and visualizations were assisted by GPT 5.1 as a co-writing tool under human direction.

Contents

1 How to Use This Workbook

This workbook is a self-contained, hands-on breakout that extends the narrative of *A Short History of Computer Science: From Commodore C64 to Cloud AI* [1] by zooming in on cellular automata as a concrete case study. You do *not* need to have read every chapter of the main book, but it helps if you are comfortable with basic Python and with the idea of an array or matrix. The only hard requirement is curiosity: a willingness to run short programs, tweak parameters, and look closely at what simple rules do over many time steps.

There are three good ways to work through the material:

- **Sidecar to the main book:** read the relevant history chapter, then use the workbook sections to make the abstract story concrete by building and running your own models.
- **Standalone lab:** start directly here, treating each section as a guided mini-project. When the text mentions historical figures or themes, you can always circle back to the main book later for more context.
- **Experiment notebook:** if you already know the basic theory, skip ahead to the implementation and exploration sections and use the exercises as a structured way to run and record your own experiments.

For the practical work you only need a recent Python installation (version 3.9 or later is ideal) plus NumPy and Matplotlib. A minimal check that your environment is ready looks like this:

Quick environment check

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

print("NumPy version:", np.__version__)
print("Matplotlib version:", plt.__version__)
```

If those imports succeed and you see version numbers printed, you are ready to go. Everything else is introduced step by step inside the workbook.
The compiled PDF of this workbook is available at `https://hilpisch.com/automata.pdf`.

Target Reader and Prerequisites

The primary reader is a curious student with basic Python and linear-algebra literacy who wants to understand what cellular automata are, where they came from historically, and how to implement Conway's Game of Life from scratch. The workbook can also support mentors and instructors who use it as a structured lab for a history-of-computing or introductory complex-systems course.

What You Will Build

By the end of this workbook you will have:

- implemented a minimal but complete Game-of-Life simulator on a two-dimensional grid,
- reproduced classic patterns such as still lifes, oscillators, and gliders,
- experimented with alternative rules and neighbourhoods,
- and built an interactive viewer where you can pause, step, and adjust parameters such as grid size and update speed.

2 Concept Map and Learning Goals

This section makes the structure of the workbook explicit so you can keep track of why each activity matters. The concept map in Figure 1 places *cellular automata* in the middle and arranges four perspectives around it: *historical context*, *mathematical definition*, *implementation*, and *experiments*. The arrows are a reminder that the traffic runs both ways: historical questions motivate definitions, definitions suggest implementations, implementations enable experiments, and surprising experimental results often send you back to refine both the story and the mathematics.

In the **historical** quadrant you will meet names such as Stanislaw Ulam, John von Neumann, John Conway, and Stephen Wolfram. You will see how ideas that started as abstract thought experiments and pencil-and-paper sketches turned into computer programs that spread through magazines, early microcomputers, and, eventually, online communities.

In the **mathematical** quadrant you will learn to describe a cellular automaton precisely as:

$$\text{“grid”} + \text{“states”} + \text{“neighbourhood”} + \text{“local update rule”},$$

or, if you like symbols,

$$F : S^{\mathbb{Z}^2} \to S^{\mathbb{Z}^2}, \qquad c_{t+1} = F(c_t),$$

where S is a finite set of states (for example $S = \{0, 1\}$ for dead/alive cells), c_t is the configuration of the whole grid at time t, and F applies the same local update rule at every cell. The important point is not the notation itself but the discipline of saying exactly what depends on what.

In the **implementation** quadrant you will translate that abstract picture into concrete Python code. You will learn patterns such as:

- representing the grid as a two-dimensional NumPy array,
- computing neighbour counts efficiently using slices or convolution-like operations,
- and structuring your code so that you can easily swap in different rules or boundary conditions.

In the **experiments** quadrant you will treat your code as a small laboratory. You will design initial patterns, run simulations, and record what you see: which structures die out, which stabilise, which move, and how often small changes in the rules produce qualitatively new behaviour. Along the way you will practice speaking both the “microscopic” language of local updates and the “macroscopic” language of patterns, signals, and emergent structure.

By the time you reach the final project, you should be comfortable doing all of the following:

- describing a cellular automaton precisely enough that someone else could reimplement it from your specification,

- placing Conway's Game of Life in the broader historical development of computer science and complexity science,
- translating informal update rules into clear pseudo-code and idiomatic Python,
- and using experiments with simple rules to generate, and sometimes answer, your own questions about emergence and computation.

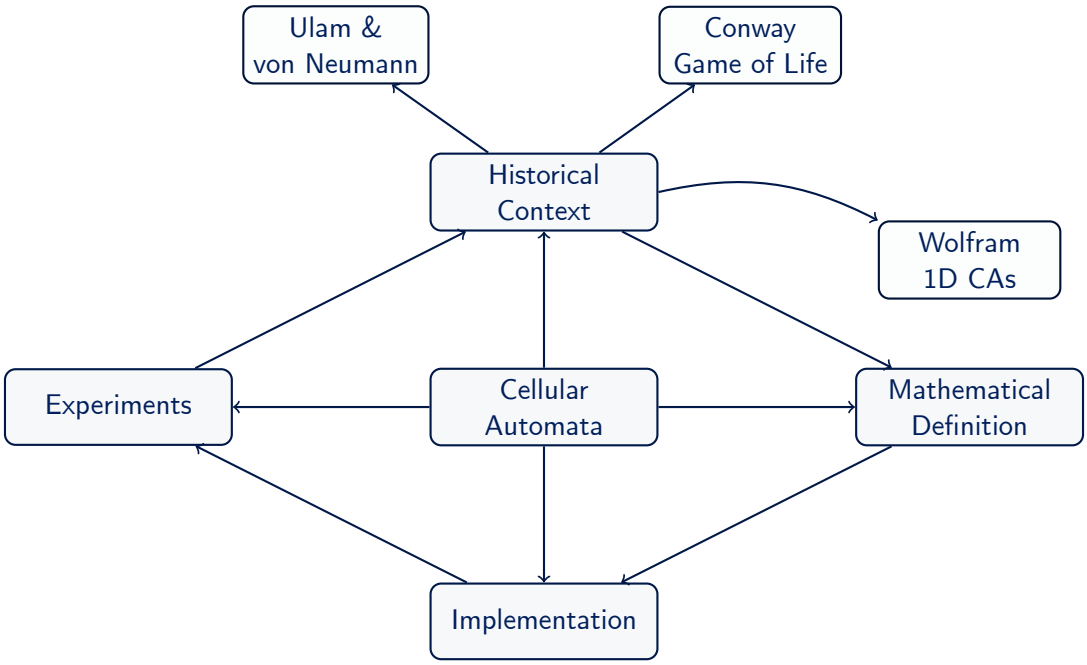

Figure 1: Concept map for the workbook: how historical context, mathematical definition, implementation, and experiments connect around the core idea of cellular automata.

3 A Short History of Cellular Automata

Cellular automata did not start life as programming toys. They emerged from serious attempts to think about self-reproduction, reliable computation, and the relationship between simple local rules and complex global behaviour. Only later did they become staples of recreational mathematics, early home-computer magazines, and, eventually, complex-systems research. The vertical sequence of milestones in Figure 2 offers a compact visual guide to this story, from the first lattice models to modern applications.

3.1 From Ulam and von Neumann to Conway

The basic picture—a grid of cells, each in a discrete state, updating according to a local rule—goes back to work by Stanislaw Ulam and John von Neumann in the 1940s and 1950s [2, 3]. Ulam used lattice models to study growth processes and percolation, while von Neumann asked whether a machine could *construct a copy of itself* using only local interactions on such a lattice. His self-reproducing automaton was defined on a two-dimensional grid with 29 possible states per cell and a carefully designed neighbourhood rule. By modern standards the construction is baroque, but the core idea is recognisably cellular-automaton-like.

At roughly the same time, digital computers were moving from theoretical blueprints to physical machines. Cellular automata provided one answer to the question "How can we reason about computation in a spatially extended medium?" Unlike Turing machines, which move a single tape head back and forth, cellular automata let every cell update in parallel, which made

them attractive both as abstract parallel models and as conceptual tools for thinking about distributed physical systems.

The picture changed dramatically when John Conway introduced the Game of Life around 1970. Life lives on a two-dimensional grid of square cells, each of which is either *alive* or *dead*. At each tick of the clock, the state of every cell is updated simultaneously according to four simple rules based on the number of live neighbours. Compared to von Neumann's original construction, Life uses:

- a much smaller state space ($S = \{0, 1\}$ instead of 29 states),
- a visually intuitive neighbourhood (the eight surrounding cells, often called the Moore neighbourhood),
- and rules that can be explained in a few lines to a non-specialist.

Conway's system was popularised by Martin Gardner's column "Mathematical Games" in *Scientific American* [4]. The combination of simple rules, rich behaviour, and easy pencil-and-paper experimentation made Life an instant hit. Very quickly, enthusiasts began implementing it on mainframes and minicomputers, then on early home computers. Life programs showed up in magazines, on university timesharing systems, and later as demonstrations of new graphical hardware. The same grid that once served von Neumann as a theoretical canvas for self-reproducing machines had become an accessible playground where students, hobbyists, and researchers could watch complexity unfold in real time.

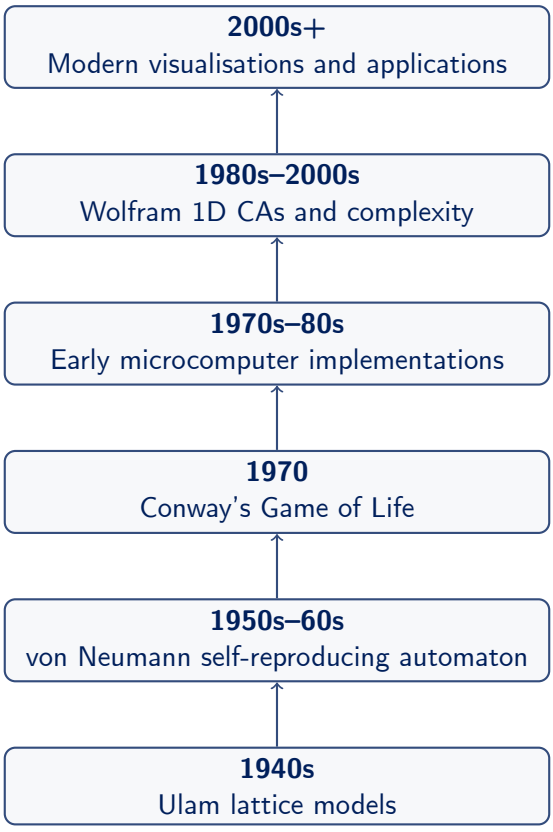

Figure 2: Timeline of key milestones in the history of cellular automata, from early lattice models and self-reproducing automata to Conway's Game of Life, one-dimensional rules, and modern applications.

3.2 Cellular Automata as Historical "Toy Models"

Very quickly, cellular automata also earned a second identity: that of *toy models*. A toy model is not a full-blown theory of its target system. Instead, it is a deliberately simplified setup that is easy to specify, easy to simulate, and rich enough to make at least one real question vivid. Toy models trade realism for clarity and playability.

In the 1970s and 1980s, Life became a staple of early computing clubs and magazines. Short BASIC listings for Life and its variants taught generations of programmers about arrays, nested loops, and simple graphics. A typical listing encoded the grid as a two-dimensional array `A(I,J)` of zeros and ones and used three nested loops to compute the next generation from the current one. The same idea translates neatly into modern pseudo-code:

Pseudocode for a generic 2D cellular automaton

```
for i in range(height):
    for j in range(width):
        neighbourhood = current[i-1:i+2, j-1:j+2]  # local patch
        s = neighbourhood.sum() - current[i, j]  # neighbours only
        next[i, j] = rule(current[i, j], s)  # same rule everywhere
```

Here `rule` encodes whatever local update logic we care about: Conway's Life, a forest-fire model, a simple traffic model, or something entirely new. The important point is that once you understand the pattern *"look at neighbours, compute a summary, apply a rule"*, you can swap in new rules without changing the computational skeleton.

Beyond programming education, cellular automata migrated into artificial-life research and complexity science. Simple one-dimensional rules such as Wolfram's Rule 30 and Rule 110 were used as microscopes for questions about randomness, structure, and computation in minimal systems [5, 6]. Historians and philosophers of science have been drawn to these models because they are unusually transparent: you can see every part of the mechanism, run the dynamics yourself, and still encounter behaviour that is hard to predict or summarise. That makes cellular automata ideal case studies for thinking about explanation, emergence, and the limits of simple rules—questions that the companion philosophy-of-science lab on cellular automata [7] takes up in more depth.

4 Core Ideas: Grids, States, and Rules

This section introduces the basic ingredients of a cellular automaton in a way that maps cleanly onto code you will write later. You will define grids, states, neighbourhoods, and local rules, and you will use these concepts to describe Conway's Game of Life precisely.

4.1 Defining a Cellular Automaton

At a high level, a cellular automaton is a discrete-time dynamical system with four ingredients: *a grid*, *a finite set of states*, *a neighbourhood*, and *a local update rule*. All cells update synchronously using the same rule.

One convenient way to write this down is:

- a dimension $d \in \{1, 2, 3, \dots\}$ and a grid of cells indexed by $\mathbb{Z}^d$,
- a finite state set $S = \{s_1, \dots, s_k\}$,
- a finite neighbourhood $N \subset \mathbb{Z}^d$ (for example, in two dimensions the Moore neighbourhood $N = \{(i, j) : i, j \in \{-1, 0, 1\}\}$),

- a local rule $f : S^N \to S$ that takes the states in a cell's neighbourhood and returns the new state for the centre cell.

If we write the configuration of the whole grid at time t as a function $c_t : \mathbb{Z}^d \to S$, then the global update rule is

$$c_{t+1}(x) = f\big((c_t(x+n))_{n \in N}\big) \qquad \text{for every cell } x \in \mathbb{Z}^d.$$

That compact formula just says: "to update a cell, look at its neighbours in N, feed those states into f, and use the result as the new state." Table 1 summarises the main symbols so you can keep the ingredients straight while reading the code and examples that follow.

Table 1: Key symbols in a cellular automaton definition.

Symbol	Informal meaning
d	Dimension of the grid (1D line, 2D sheet, …)
S	Finite set of states a cell can take (for example $\{0, 1\}$)
N	Neighbourhood offsets around a cell
f	Local rule mapping neighbour states to a new state
c_t	Configuration of the whole grid at time t
$c_{t+1}(x)$	State of cell x after one update step

In code, you can mirror this structure very directly. Before we zoom in on indices and neighbourhoods, it is helpful to have an intuitive picture in mind. Figure 3 invites you to think of a cellular automaton as a stylised map of city blocks: each block is a cell, its colour represents a simple state (for example "lights on" or "lights off"), and at each tick every block decides what to do next by "looking" at the pattern of lights in the neighbouring blocks.

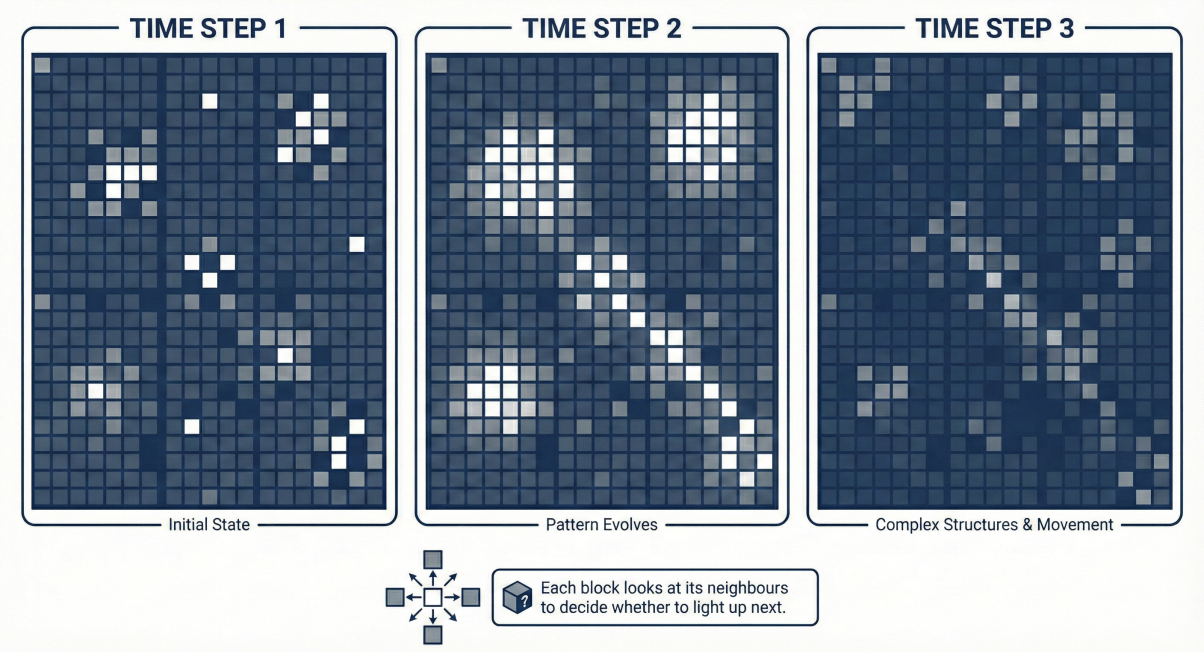

Figure 3: Intuitive picture of a cellular automaton. Think of this as a stylised map of city blocks where each block decides whether to light up in the next moment by "looking" at which neighbouring blocks are currently lit.

With that mental image in place, Figure 4 then translates the same idea into a minimal mathematical schematic with indices (i, j) and an explicit neighbourhood.

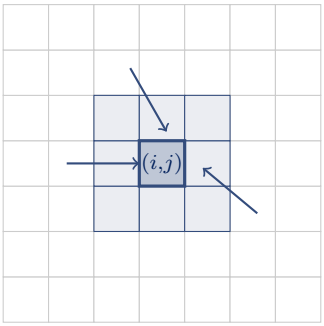

Figure 4: Schematic view of a two-dimensional cellular automaton: each cell lives on a regular grid, and the state of a central cell (i, j) is updated using information from a small local neighbourhood around it.

For a two-dimensional grid stored as a NumPy array `grid` we can write:

Local update rule as a Python function

```python
import numpy as np

def local_rule(current_state: int, live_neighbours: int) -> int:
    """Example of a binary-state local rule.

    This will later be specialised to Conway's Game of Life.
    """
    # Placeholder: by default, keep the current state.
    return current_state

def step_generic(grid: np.ndarray) -> np.ndarray:
    """Apply the same local_rule to every cell in the grid."""
    height, width = grid.shape
    next_grid = grid.copy()
    for i in range(1, height - 1):
        for j in range(1, width - 1):
            neighbourhood = grid[i-1:i+2, j-1:j+2]
            live_neighbours = int(neighbourhood.sum() - grid[i, j])
            next_grid[i, j] = local_rule(int(grid[i, j]), live_neighbours)
    return next_grid
```

The code simply unpacks the mathematics into a nested loop: for each cell, read a neighbourhood, compute a summary (here, "number of live neighbours"), and then call the same local rule everywhere.

4.2 Conway's Game of Life as a Worked Example

Conway's Game of Life is a special case of this general framework:

- Dimension $d = 2$.
- State set $S = \{0, 1\}$, where 0 means "dead" and 1 means "alive".
- Neighbourhood N is the 3×3 Moore neighbourhood around each cell (the cell itself and its eight immediate neighbours).
- The local rule f depends only on the current state and the number of live neighbours.

If we write the state of a cell as $x \in \{0, 1\}$ and the number of live neighbours as $n \in \{0, 1, \ldots, 8\}$, the Life rule can be written compactly as:

$$f(x, n) = \begin{cases} 1, & \text{if } x = 1 \text{ and } n \in \{2, 3\} \quad \text{(survival)}, \\ 1, & \text{if } x = 0 \text{ and } n = 3 \quad \text{(birth)}, \\ 0, & \text{otherwise} \quad \text{(death or staying dead)}. \end{cases}$$

In words:

- a live cell with two or three live neighbours stays alive,
- a dead cell with exactly three live neighbours becomes alive,
- all other cells are dead in the next step.

We can now specialise the generic code skeleton to the Life rule:

Conway's Game of Life local rule

```
def life_rule(current_state: int, live_neighbours: int) -> int:
    """Conway's Game of Life local rule."""
    if current_state == 1 and live_neighbours in (2, 3):
        return 1
    if current_state == 0 and live_neighbours == 3:
        return 1
    return 0
```

You can now drop this local rule into the generic update loop from Table 1 by replacing `local_rule` with `life_rule`.

It is often helpful to see a small concrete example. Consider the following 5×5 configuration at time t, with $\bullet$ indicating a live cell and $\circ$ a dead cell:

$$\begin{array}{ccccc} \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \bullet & \circ & \circ \\ \circ & \circ & \bullet & \circ & \circ \\ \circ & \circ & \bullet & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \end{array}$$

This is a vertical line of three live cells (a "blinker"). Counting neighbours shows that each of the three live cells has exactly two live neighbours, while the two central neighbours to the left and right each see exactly three live neighbours. After one update step we obtain the configuration shown in Figure 5:

$$\begin{array}{ccccc} \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \\ \circ & \bullet & \bullet & \bullet & \circ \\ \circ & \circ & \circ & \circ & \circ \\ \circ & \circ & \circ & \circ & \circ \end{array}$$

The pattern has rotated into a horizontal line: an example of a simple *oscillator* with period two.

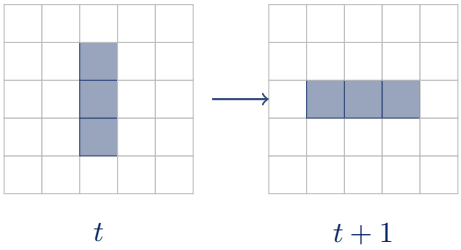

Figure 5: Example of one Game-of-Life update step from time t to $t+1$: a vertical three-cell line (a "blinker") turns into a horizontal three-cell line.

Qualitative Behaviours to Watch For

As you implement and experiment with Life, look for three recurring behaviours:

- **Extinction**: patterns that quickly die out and leave an empty grid.
- **Stability and oscillation**: patterns that settle into a fixed shape or cycle through a small number of configurations.
- **Motion**: patterns such as gliders that appear to move across the grid over time.

Recognising these behaviours by eye will make later experiments much more informative.

5 Hands-On Implementation: Life in Python

This section turns the conceptual model into working code. The goal is a small, well-documented implementation that you can run in any standard Python environment.

5.1 Environment Setup and Minimal Loop

To make the examples concrete we assume:

- Python 3.9 or later,
- NumPy for array-based computation,
- Matplotlib for simple visualisation.

If you like working in virtual environments, a typical setup on the command line looks like:

Example shell commands for setup (type in a terminal)

```
python -m venv ca-env
source ca-env/bin/activate  # or ca-env\Scripts\activate on Windows
pip install --upgrade pip
pip install numpy matplotlib
```

We represent the Life grid as a two-dimensional NumPy array of zeros and ones:

$$\texttt{grid[i, j]} = \begin{cases} 1, & \text{if the cell at row } i, \text{ column } j \text{ is alive,} \\ 0, & \text{if it is dead.} \end{cases}$$

The core of the implementation is a function that applies one Life update step to such a grid.

A first implementation of one Life step

```
import numpy as np

def step_life(grid: np.ndarray) -> np.ndarray:
    """Single Game-of-Life update step on a 2D NumPy array."""
    height, width = grid.shape
    next_grid = grid.copy()

    for i in range(1, height - 1):
        for j in range(1, width - 1):
            neighbourhood = grid[i-1:i+2, j-1:j+2]
            live_neighbours = int(neighbourhood.sum() - grid[i, j])
            next_grid[i, j] = life_rule(int(grid[i, j]), live_neighbours)

    return next_grid
```

This version ignores the outermost border cells for simplicity. Later you can experiment with different boundary conditions (fixed borders, wrap-around, absorbing edges).

To see whether the update rule behaves sensibly, it helps to have a tiny driver loop that prints a crude text representation of the grid:

Text-based Life loop for quick checks

```
def print_grid(grid: np.ndarray) -> None:
    """Print the grid as ASCII art using '.' and '#'."""
    for i in range(grid.shape[0]):
        row = "".join("#" if cell else "." for cell in grid[i])
        print(row)
    print()

def run_life(initial_grid: np.ndarray, steps: int = 5) -> None:
    """Run Life for a fixed number of steps and print each grid."""
    grid = initial_grid.copy()
    for t in range(steps):
        print(f"Step {t}")
        print_grid(grid)
        grid = step_life(grid)
```

With this in place you can define a small initial pattern in a Python session and watch its evolution in plain text before investing in nicer graphics.

5.2 Visualising Patterns

Once you trust your update rule, the next step is to make the patterns visible. The simplest approach is to use Matplotlib's `imshow` to display the grid as a small black-and-white image, where 1 (alive) corresponds to a dark square and 0 (dead) to a light square.

Minimal Matplotlib visualisation

```
import matplotlib.pyplot as plt

def show_grid(grid: np.ndarray) -> None:
    """Display the Life grid as a black-and-white image."""
    plt.imshow(grid, cmap="Greys", interpolation="nearest")
    plt.axis("off")
```

```
    plt.show()
```

For interactive exploration, you can place this inside a loop and update the figure in place. One simple option is to use `plt.pause`:

Animated visualisation in a simple loop

```python
def animate_life(initial_grid: np.ndarray, steps: int = 100, pause: float = 0.1) -> None:
    """Animate Life in a Matplotlib window."""
    grid = initial_grid.copy()
    plt.ion()
    fig, ax = plt.subplots()
    img = ax.imshow(grid, cmap="Greys", interpolation="nearest")
    ax.axis("off")

    for _ in range(steps):
        img.set_data(grid)
        fig.canvas.draw()
        fig.canvas.flush_events()
        plt.pause(pause)
        grid = step_life(grid)

    plt.ioff()
    plt.show()
```

Later, if you wish, you can replace this with more polished animations or export sequences of frames to create GIFs. For now the goal is simply to make it easy to see whether still lifes stay still, oscillators cycle, and gliders glide.

6 Exploration: Classic Patterns and Experiments

This section invites you to play systematically with well-known Life patterns and to record what you see. The emphasis is on observation, measurement, and connecting behaviour back to the underlying rules.

6.1 Still Lifes, Oscillators, and Gliders

Classic Life patterns make excellent test cases for your implementation and excellent "characters" for telling the story of the model. Three broad families are especially important: *still lifes*, *oscillators*, and *spaceships* (moving patterns such as gliders).

Here are small binary arrays for a few canonical examples, using 1 for live cells and 0 for dead ones; Figure 6 then collects them into a visual "pattern gallery" you can use as a quick reference while testing your code:

- Block (a simple still life):

$$\begin{bmatrix} 1 & 1 \\ 1 & 1 \end{bmatrix}$$

- Blinker (the period-two oscillator from above, centred in a 5×5 grid):

$$\begin{bmatrix} 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 \end{bmatrix}$$

- Glider (a small spaceship that moves diagonally):

$$\begin{bmatrix} 0 & 1 & 0 \\ 0 & 0 & 1 \\ 1 & 1 & 1 \end{bmatrix}$$

In code you can embed these as NumPy arrays:

Defining a few classic patterns

```
block = np.array([[1, 1],
                  [1, 1]], dtype=int)

blinker = np.array(
    [[0, 0, 0, 0, 0],
     [0, 0, 1, 0, 0],
     [0, 0, 1, 0, 0],
     [0, 0, 1, 0, 0],
     [0, 0, 0, 0, 0]],
    dtype=int,
)

glider = np.array(
    [[0, 1, 0],
     [0, 0, 1],
     [1, 1, 1]],
    dtype=int,
)
```

Suggested experiments:

- Place each pattern into a larger empty grid (for example, 30×30) and verify that: blocks remain unchanged, blinkers cycle with period two, and gliders move diagonally.
- Gently "poke" each pattern by flipping a single cell from dead to alive (or vice versa) and observing whether the qualitative behaviour survives.
- Measure simple quantities over time, such as the total number of live cells or the bounding box of the pattern, and plot them to get a more quantitative feel for the dynamics.

Figure 6: Representative Game-of-Life patterns: still lifes, oscillators, and gliders. Illustrator prompt: compact multi-panel figure with clear labels and minimal decoration.

6.2 Design-Your-Own Experiments

This subsection turns you into the experiment designer. Treat your Life implementation as a small laboratory in which you can pose questions, run trials, and record what happens.

A simple checklist for formulating questions:

- **Lifetime**: "How long does this pattern remain active before it dies out or settles into a stable configuration?"
- **Reach**: "How far, in grid cells, does activity spread from the initial region?"
- **Robustness**: "Does a small change in the initial condition qualitatively change the long-term behaviour?"

One way to keep your experiments organised is to use a small table or notebook template with four lines per run:

- *Initial pattern*: a short description and, if relevant, a saved NumPy array or seed value.
- *Parameters*: grid size, number of steps, boundary conditions.
- *Observations*: what you saw (for example, "block remained stable", "glider collided with debris and broke apart").
- *Questions*: follow-up ideas the run suggests.

As you collect runs, look for families of behaviour rather than isolated curiosities. For example, you might discover that many small random initial conditions either die out quickly or generate a handful of gliders, or that certain configurations reliably produce fractal-like debris patterns. These qualitative impressions will be useful when you read the companion philosophy-of-science lab [7], which asks how such toy experiments can inform broader discussions about emergence and explanation.

7 Project: Interactive Life with Controls

The final project of the workbook is to build an interactive Life viewer with basic controls for grid size, simulation speed, and (optionally) rule selection. The implementation sketched here uses Matplotlib and plain Python only; if you prefer a GUI toolkit or a browser-based front end, you can treat this as a reference design and adapt it.

Conceptually, the project has three modules:

- **State update**: functions such as `life_rule` and `step_life` that know nothing about graphics or user input.
- **Rendering**: code that turns a grid into pixels on screen.
- **User input**: code that responds to key presses or button clicks by pausing, stepping, randomising, or resetting the grid.

Keeping these pieces separate makes the system easier to understand and to extend.

7.1 Minimal Interactive Viewer

The simplest interactive viewer is a window that:

- shows the current grid,
- advances the simulation automatically when a "running" flag is true,
- and reacts to a few keys such as `Space` (pause/resume), `N` (single step), and `R` (reset to a random pattern).

Here is a compact implementation that follows this design:

Interactive Life viewer with keyboard controls

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

class LifeViewer:
    def __init__(self, height: int = 60, width: int = 80, p_alive: float = 0.2):
        self.height = height
        self.width = width
        self.p_alive = p_alive
        self.grid = self._random_grid()
        self.running = False

        self.fig, self.ax = plt.subplots()
        self.img = self.ax.imshow(self.grid, cmap="Greys", interpolation="nearest")
        self.ax.axis("off")
        self.fig.canvas.mpl_connect("key_press_event", self.on_key)

    def _random_grid(self) -> np.ndarray:
        return (np.random.rand(self.height, self.width) < self.p_alive).astype(int)

    def on_key(self, event) -> None:
        if event.key == " ":  # space toggles run/pause
            self.running = not self.running
        elif event.key == "n":  # single step
            self.step()
        elif event.key == "r":  # reset
            self.grid = self._random_grid()
            self._draw()

    def step(self) -> None:
        self.grid = step_life(self.grid)
        self._draw()

    def _draw(self) -> None:
        self.img.set_data(self.grid)
        self.fig.canvas.draw_idle()

    def run(self, pause: float = 0.05) -> None:
        plt.ion()
        while plt.fignum_exists(self.fig.number):
            if self.running:
                self.step()
            plt.pause(pause)
        plt.ioff()
```

To use this class in a Python session, you can write:

Starting the interactive viewer

```
viewer = LifeViewer(height=60, width=80, p_alive=0.2)
viewer.run(pause=0.05)
```

Once the window appears, use:

- `Space` to toggle between running and paused,
- `n` to advance a single step while paused,
- `r` to reset the grid to a fresh random configuration.

Even this minimal set of controls already turns the model into a genuine exploration tool rather than a fixed animation.

7.2 Possible Extensions

Once the basic viewer works, you can treat it as a platform for experiments:

- Add keyboard shortcuts or simple on-screen buttons to switch between different rules (for example, Life and a few Life-like variants).
- Let the user "paint" live cells with the mouse before starting the simulation, so that custom initial conditions are easy to create.
- Display simple statistics (live-cell count, number of distinct connected components, bounding box of activity) in a small status bar.
- Allow saving and loading patterns using plain-text files or NumPy `.npy` arrays.

If you build larger extensions, keep the separation between update logic, rendering, and input in mind. That separation will make it much easier to reason about the behaviour of your system—a theme that reappears in the main history book and the companion philosophy-of-science lab [7].

Where This Connects Next

In the companion philosophy-of-science lab on cellular automata [7], you will step back from code and ask what systems like Life teach us about emergence, modelling, and explanation. This workbook prepares you for that discussion by giving you concrete experiences and artefacts to refer to.

References

[1] Y. J. Hilpisch. *A Short History of Computer Science: From Commodore C64 to Cloud AI*. 2025.

[2] S. Ulam. On some mathematical problems connected with patterns of growth of figures. In *Proceedings of Symposia in Applied Mathematics*, volume 14, pages 215–224. American Mathematical Society, 1962.

[3] J. von Neumann. *Theory of Self-Reproducing Automata*. Edited and completed by A. W. Burks. University of Illinois Press, 1966.

[4] M. Gardner. Mathematical games: The fantastic combinations of John Conway's new solitaire game "Life". *Scientific American*, 223(4):120–123, 1970.

[5] S. Wolfram. Statistical mechanics of cellular automata. *Reviews of Modern Physics*, 55(3):601–644, 1983.

[6] M. Cook. Universality in elementary cellular automata. *Complex Systems*, 15(1):1–40, 2004.

[7] Y. J. Hilpisch. *POS Lab: Cellular Automata, Emergence, and Explanation*. 2025. Companion lab article to *Introduction to the Philosophy of Science: Concepts, Practice, and Case Studies*.