CONWAY'S GAME OF LIFE

CSC 109 - Introduction to programming in Python - Summer 2023

Marcus Birkenkrahe

June 14, 2023

Contents

1 README

- More on cellular automata in Python (Ilievski, 2021)
- More on cellular automata (Wikipedia)
- Conway's Game of Life (Wikipedia)
- Code source: Sweigart (2019)/OpenAI. For more games, see GitHub.

2 Mathematical basis: cellular automata

- Automata are self-operating state-machines or iterative arrays
- History: discovered by Ulam/von Neumann (1940s), popularized by Conway (1970s), analyzed by Wolfram (1980s).
- *Applications* include: fluid dynamics simulations in **physics**, plant growth processes in **biology**, pseudo random number generation in **cryptography**, traffic flow modeling, city growth patterns in **urban planning**, spread of forest fires in **ecology**, chemical reaction models in **chemistry**, terrain generation in **game development**, etc.
- Automata are conveniently represented as directed (cyclic) graphs: here is an example of a 2 state finite automaton.
- In the depicted automaton, S1 and S2 are states, and the edges are transition lines that are triggered by the state: e.g. 0 on S1 triggers a state change from S1 to S2.
- The language of graphs is also important in network analysis, database and algorithm design.

3 Example: a 'Rule 30' automaton in Python

- Here's a simple Python implementation of a one-dimensional cellular automaton, specifically the 'Rule 30 automaton'.
- Despite its simplicity, it generates complex, seemingly random patterns (e.g. on shells).
- The code:

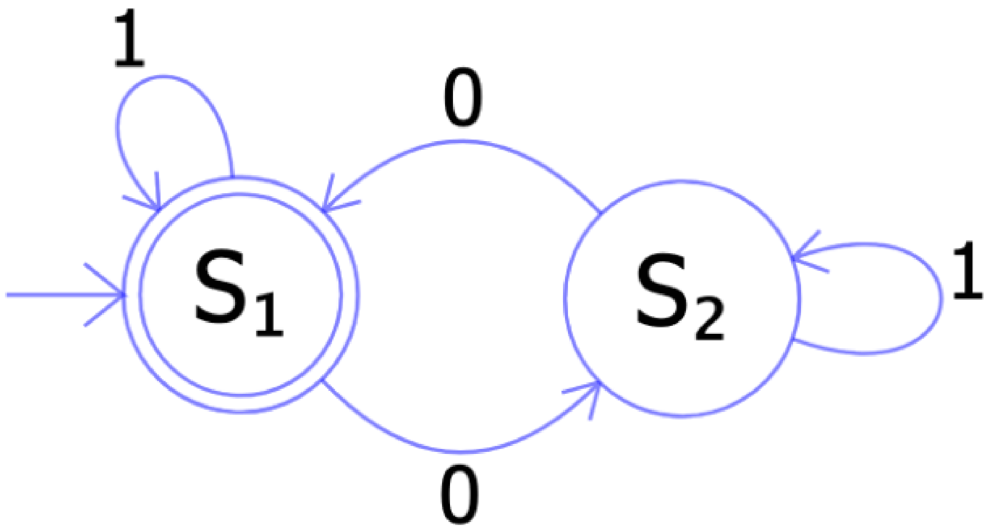


Figure 1: Simple automaton with two states (Ilievski, 2021)

```
# function that applies rule 30 to a list of cells
def rule30(cells):
    # Applies Rule 30 to the given list of cells
    new_cells = []
    for i in range(len(cells)):
        left = cells[i - 1] if i > 0 else 0
        center = cells[i]
        right = cells[i + 1] if i < len(cells) - 1 else 0
        new_cells.append(left ^ (center or right))  # Rule 30 logic
    return new_cells

# Initial configuration
cells = [0, 0, 0, 0, 0, 1, 0, 0, 0, 0, 0]

# Run the automaton for 10 steps
for _ in range(10):
    print(''.join('#' if cell else ' ' for cell in cells))
    cells = rule30(cells)
```

- How the program works:

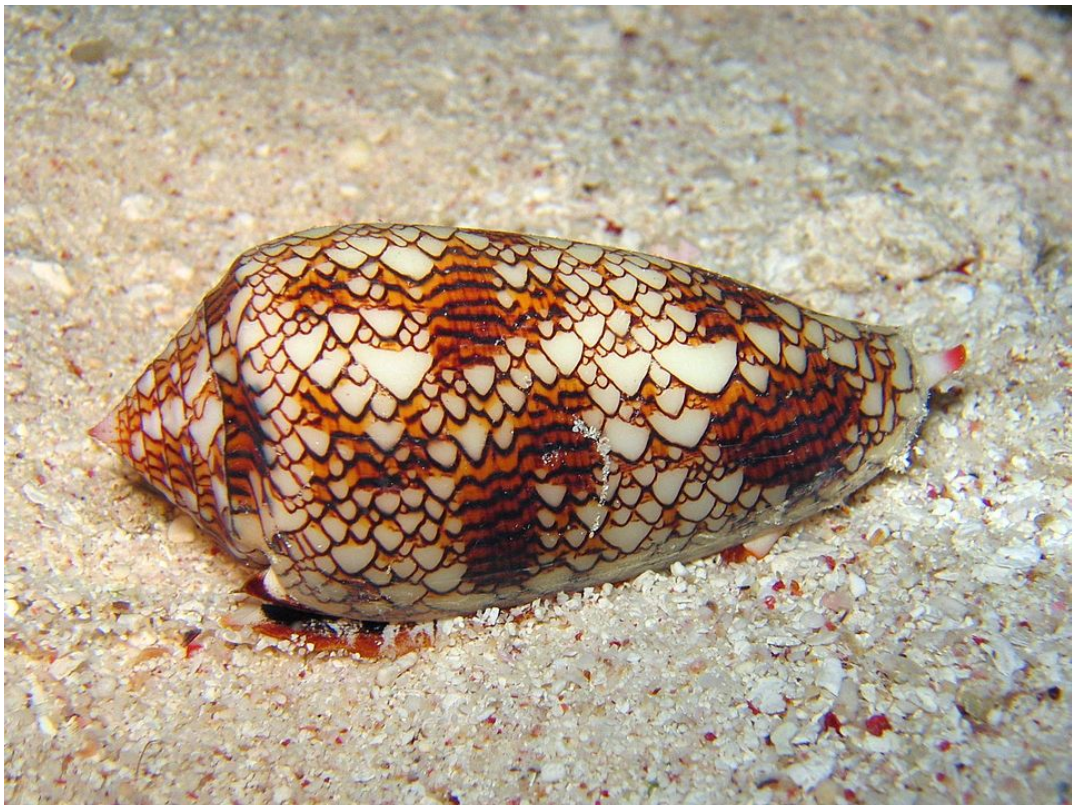
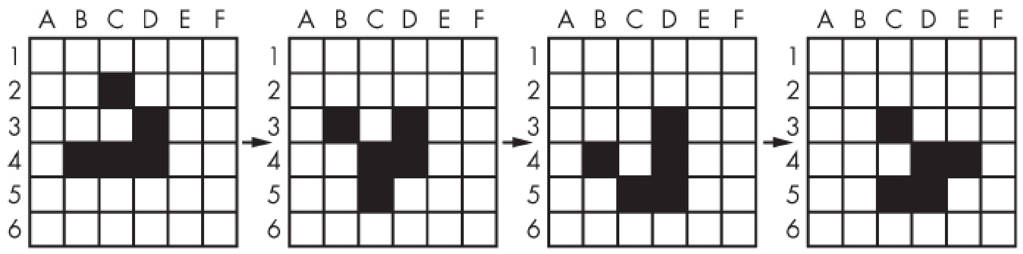


Figure 2: A Conus textile shell similar in appearance to Rule 30 (Wikipedia)

1. `rule30` is a function that takes a list of cells as input and applies Rule 30 to generate the new state of these cells. Rule 30 is one of the elementary cellular automaton rules introduced by Stephen Wolfram. It's named "Rule 30" because the rule number, 30, translates to `00011110` in binary, which dictates the new state for a given triplet of cells.
2. Inside the `rule30` function, a new list called `new_cells` is created to store the new state of the cells.
3. The function then iterates over each cell in the input list (`cells`). For each cell, it determines the state of its left neighbor, itself, and its right neighbor. Based on these states, it computes the new state using the Rule 30 logic: `new_state = left XOR (center OR right)`, where XOR (or `^`) is the bitwise exclusive OR operation.
4. After iterating through all the cells, the function returns the `new_cells` list, which represents the new state of the cells after applying Rule 30.
5. The main part of the program initializes the cells with a specific configuration (11 cells, with the sixth cell being in state 1 and the rest in state 0).
6. The program then runs the cellular automaton for 10 steps. In each step, it prints the current state of the cells as a string of `#` characters for state 1 and spaces for state 0. After printing, it calls the `rule30` function to compute the new state of the cells for the next step.
7. The underscore `_` in the `for` loop is a conventional placeholder or throwaway variable since it is not actually needed anywhere: the loop is there only to execute the clause 10 times.
8. Which *methods* are used in this program?
    - `list.append`: appends item to list in place
    - `str.join`: joins any number of strings
    - Other functions: `range`, `len`, `print`.

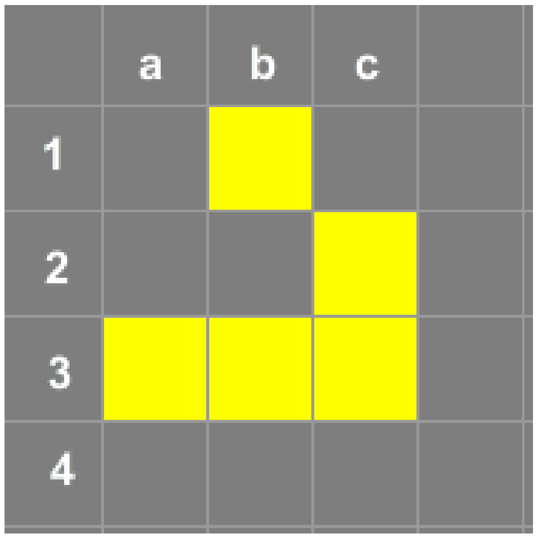
4 Extended example: Conway's 'Game of Life'

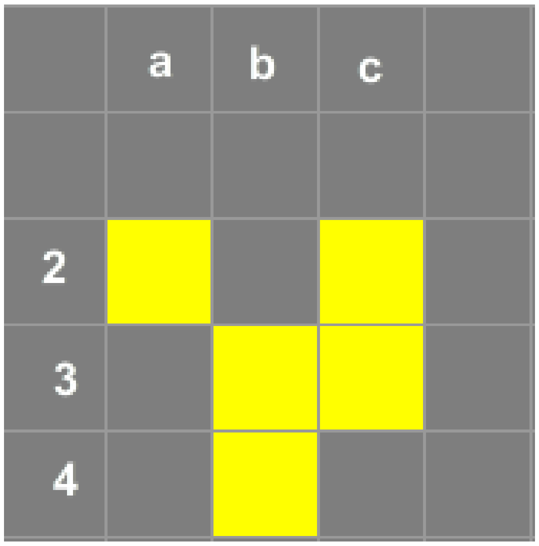
#caption: Four steps in a Conway's Game of Life simulation (Sweigart, 2019)



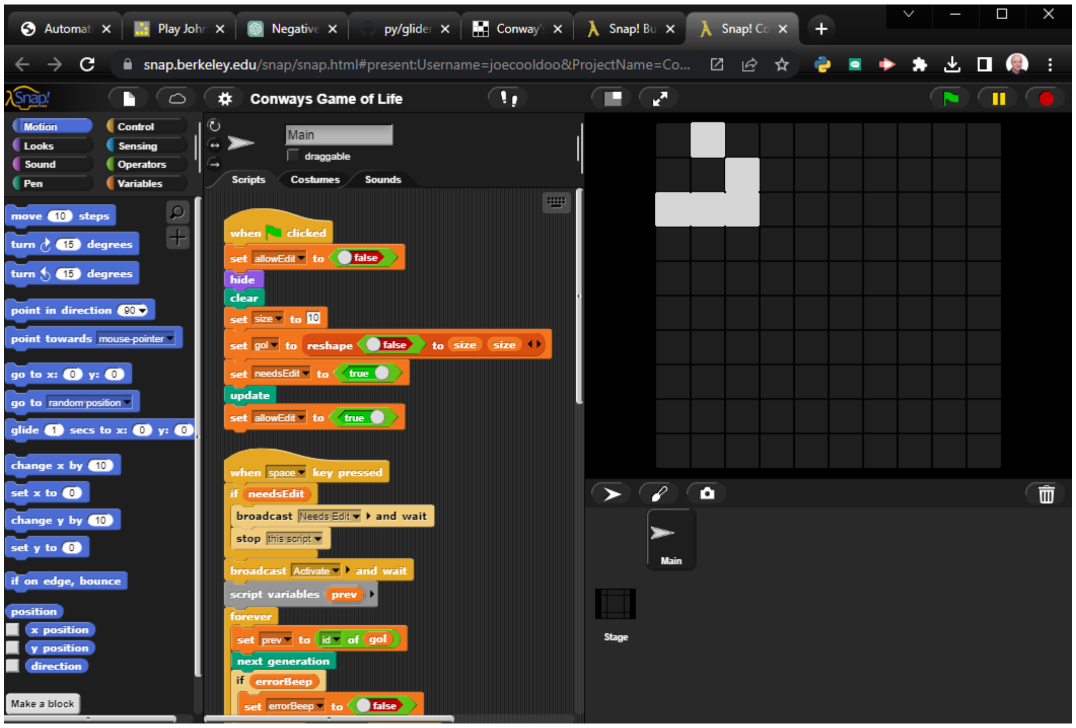
- In the graphical illustration, an empty square is 'dead', and a filled-in one is 'alive'.
- Conway's Game of Life simulation (CGOL) has four simple rules:
  1. each live cell with one or no neighbors dies.
  2. each live cell with four or more neighbors dies.
  3. each live cell with two or three neighbors survives.
  4. each dead cell with three neighbors becomes populated.
- In a 4 x 4 state space, the two automatons below describe the transition of 5 live cells and reproduction of 2 live cells:

cell	next	state 1	state 2	fate
b1	1	1	0	dies
c2	3	1	1	lives
c3	2	1	1	lives
b3	3	1	1	lives
a3	1	1	0	dies
a2	3	0	1	comes alive
b4	3	0	1	comes alive





- You can play CGOL online at playgameoflife.com and check out the sequence above. On the site, click on 'explanation' to see the set of rules illustrated with examples.
- A nice project for my new Snap! course: Game of Life simulation in in Snap! here at U Berkeley:
- For a (free) 500p. book on the Game of Life and the mathematics behind it, see Johnston/Greene (2022): conwaylife.com/book/.



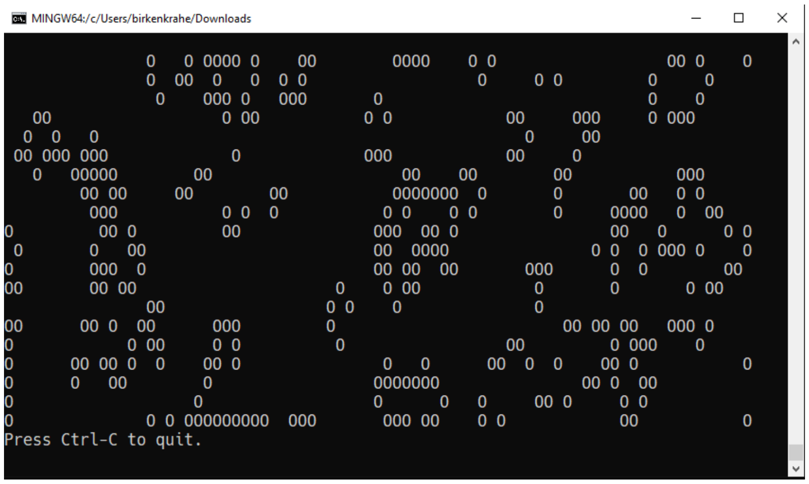
```
when [flag] clicked
set allowEdit to false
hide
clear
set size to 10
set gol to reshape false to size size
set needsEdit to true
update
set allowEdit to true

when space key pressed
if needsEdit
    broadcast Needs Edit and wait
    stop this script
broadcast Activate and wait
script variables prev
forever
    set prev to id of gol
    next generation
    if errorBeep
        set errorBeep to false
```

Figure 3: Conway Game of Life simulation 10 x 10 in Snap!

- In the code, a list of lists is used to represent a 2-dimensional field. The inner list represents each column of squares and stores one character (like 'O') for living, and a space for dead cells.

5 Conway's Game of Life using lists

- Save the file from here as `conwaysgameoflife.py`: `https://inventwithpython.com/projects/conwaysgameoflife.py`
- Save the `gameOfLife.py` from GitHub: github.com/birkenkrahe/py/blob/main/src/gameOfLife.p
- Open a CMD terminal on Windows, go to the `Downloads` directory and run the simulation with `python conwaysgameoflife.py`:

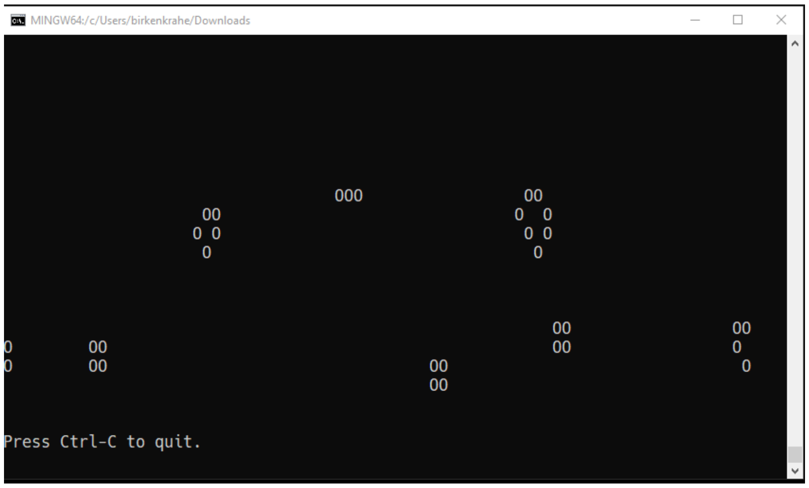


```
               O   O OOOO O    OO        OOOO    O O                  OO O    O
               O  OO  O   O  O O                  O     O O         O     O
                O    OOO O   OOO       O                            O    O
   OO                  O OO           O O            OO     OOO     O OOO
  O  O   O                                             O     OO
 OO OOO OOO             O             OOO            OO     O
   O   OOOOO        OO                    OO    OO        OO           OOO
        OO OO     OO        OO           OOOOOOO  O       O       OO   O O
         OOO           O O  O           O O    O O        O     OOOO   O  OO
O         OO O         OO              OOO  OO O                OO   O      O O
 O       O   OO                        OO  OOOO               O O  O OOO O    O
O        OOO  O                        OO OO  OO       OOO      O  O        OO
OO       OO OO                     O    O OO            O       O       O OO
               OO                 O O    O              O
OO      OO O  OO      OOO         O                        OO OO OO   OOO O
O            O OO     O O          O                 OO         O OOO    O
O      OO OO O  O    OO O               O   O      OO  O  O    OO O           O
O      O   OO        O                 OOOOOOO               OO O  OO
O                   O                  O      O   O     OO O     O O
O              O O OOOOOOOOO  OOO       OOO OO    O O            OO           O
Press Ctrl-C to quit.
```

Figure 4: Conway's Game of Life at the start and after stabilizing.

- Here's the code followed by an analysis:

```
# Conway's Game of Life
# By Al Sweigart - inventwithpython.com (2016)
import random, time, copy

WIDTH = 79   # x range
HEIGHT = 20  # y range
ALIVE = 'O'  # symbol for live cells
DEAD = ' '   # symbol for dead cells
```



```
                                   000                 00
                     00                               0  0
                    0 0                                0 0
                     0                                  0



                                                          00                 00
0        00                                               00                 0
0        00                                  00                               0
                                             00


Press Ctrl-C to quit.
```

Figure 5: Conway's Game of Life at the start and after stabilizing.

```
# create a list of lists for the cells:
nextCells = []
for x in range(WIDTH):
    column = [] # create a new column
    for y in range(HEIGHT):
        if random.randint(0,1) == 0:
            column.append(ALIVE)  # add a living cell
        else:
            column.append(DEAD)  # add a dead cell
    nextCells.append(column)  # nextCells is a list of column lists

while True: # main program loop
    print('\n\n\n\n\n')  # separate each step with newlines
    currentCells = copy.deepcopy(nextCells)

    # print currentCells on screen:
    for y in range(HEIGHT):
        for x in range(WIDTH):
            print(currentCells[x][y], end=' ') # print hash or space
        print()   # print newline at the end of the row

    # calculate the next step's cells based on current step's cells
```

```
for x in range(WIDTH):
    for y in range(HEIGHT):
        # get neighboring coordinates
        leftCoord = (x - 1) % WIDTH
        rightCoord = (x + 1) % WIDTH
        aboveCoord = (y - 1) % HEIGHT
        belowCoord = (y + 1) % HEIGHT

        # count number of living neighbors:
        numNeighbors = 0
        if currentCells[leftCoord][aboveCoord]==ALIVE:
            numNeighbors += 1 # top-left neighbor alive
        if currentCells[x][aboveCoord]==ALIVE:
            numNeighbors += 1 # top neighbor alive
        if currentCells[rightCoord][aboveCoord]==ALIVE:
            numNeighbors += 1 # top-right neighbor alive
        if currentCells[leftCoord][y]==ALIVE:
            numNeighbors += 1 # left neighbor alive
        if currentCells[rightCoord][y]==ALIVE:
            numNeighbors += 1 # right neighbor alive
        if currentCells[leftCoord][belowCoord]==ALIVE:
            numNeighbors += 1 # bottom-left neighbor alive
        if currentCells[x][belowCoord]==ALIVE:
            numNeighbors += 1 # bottom neighbor alive
        if currentCells[rightCoord][belowCoord]==ALIVE:
            numNeighbors += 1 # bottom-right neighbor alive

        # set cell based on Conway's Game of Life rules:
        if currentCells[x][y] == ALIVE:
            if numNeighbors==2 or numNeighbors==3:
                # living cells with 2-3 neighbors live:
                nextCells[x][y] = ALIVE
            else:
                nextCells[x][y] = DEAD
        else:
            # dead cells with 3 neighbors become alive:
            if numNeighbors == 3:
                nextCells[x][y] = ALIVE
            else:
                nextCells[x][y] = DEAD
    # add a 1-sec pause to reduce flickering
    time.sleep(1)
```

5.1 Import modules needed:

1. `random.randint` to populate the grid randomly with cells
2. `time.sleep` to delay execution by a second between screens
3. `copy.deepcopy` to copy a list (instead of only a reference)

```
import random, time, copy
```

5.2 Create random cell population

- We want to simulate life on a 2-dimensional canvas. You can do that with a list inside a list. We call it `nextCells` and add `WIDTH` columns of length `HEIGHT` to it using `list.append`:

```
import random
WIDTH=10
HEIGHT=5
nextCells = []
for x in range(WIDTH):
    column = []
    for y in range(HEIGHT):
        if random.randint(0,1) == 0:
            column.append('O')
        else:
            column.append(' ')
            nextCells.append(column)
```

- Print the lists to reveal the 2-dimensional structure: the list items that are lists are the columns of the 10 x 5 grid:

```
<<nextCells>>  # create random population
print('first column: ',nextCells[0][:])
print('last column:  ',nextCells[9][:],end='\n\n')
for y in range(HEIGHT):
    for x in range(WIDTH):
        print(nextCells[x][y], end=' ') # print hash or space
        print()   # print newline at the end of the row
```

```
first column:  ['O', ' ', 'O', 'O', ' ']
last column:   ['O', 'O', ' ', 'O', ' ']

O     O O O O   O O
  O O O       O   O
O O     O
O O   O O O O O O O
  O O O O
```

- What is the address of the last cell (lower right corner)?

```
nextCells[WIDTH-1][HEIGHT-1]
```

5.3 Copy cells and print them

- Each iteration of the main program loop is a single step of the cellular automata: all cells are traversed and re-evaluated to see if they live or die, or become alive:

```
while True: # main program loop
  print('\n\n\n\n\n')  # separate each step with newlines
  currentCells = copy.deepcopy(nextCells)
```

- We put this list of lists into `nextCells`, then on each step we copy `nextCells` into `currentCells`, print it to the screen and then use the cells in `currentCells` to calculate the next in `nextCells`.

```
# print currentCells on screen:
for y in range(HEIGHT):
    for x in range(WIDTH):
        print(currentCells[x][y], end=' ') # print hash or space
    print()   # print newline at the end of the row
```

5.4 Calculate indices of cells around each cell

- The living or dead state of each cell depends on its neighbors, so we calculate the indices of the cells to the left, right, above and below the current x and y coordinates:

```
# calculate the next step's cells based on current step's cells
```

```
for x in range(WIDTH):
    for y in range(HEIGHT):
        # get neighboring coordinates
        leftCoord  = (x - 1) % WIDTH
        rightCoord = (x + 1) % WIDTH
        aboveCoord = (y - 1) % HEIGHT
        belowCoord = (y + 1) % HEIGHT
```

- The `%` operator makes the index wrap around at the edges of the grid: The leftmost column `0` would be `0 - 1 = -1`. To identify this with the rightmost column `WIDTH - 1 = 59`, take `(0 - 1) % WIDTH = 59`.
- How does this work? It's called 'floored division' in Python, rounding down a number to the nearest integer that is less or equal to that number: drop the decimal part of the number and keep the integer part unchanged if it's positive or moving towards negative infinity if it's negative:

```
from math import floor, ceil
print(floor(5.8))  # 5 is the largest integer <= 5.8
print(ceil(5.8)) # 6 is the next integer >= 5.8
print(floor(2))  # 2 is already 'floored'
print(floor(-2.3)) # -3 is the largest integer <= -2.3
print(floor(-7)) # -7 is already 'floored'
```

- When you divide -1 by 60, you get approximately -0.0167. If you round this towards negative infinity, you get -1:

```
print(floor(-1/60))
```

- Now, if you plug this into the formula for modulo:

```
-1 % 60 = -1 - (60 * floor(-1/60))
        = -1 - (60 * -1)
        = -1 + 60
        = 59
```

- Print some coordinate values to see the wraparound for a 2 x 2 grid:

```
# calculate the next step's cells based on current step's cells
WIDTH  = 2
```

```
HEIGHT = 2
for x in range(WIDTH):
    for y in range(HEIGHT):
        print(f'(x,y) = ({x},{y}): ',end='')
        leftCoord =  (x - 1) % WIDTH
        rightCoord = (x + 1) % WIDTH
        print(f'left  = {leftCoord}, x = {x}, right = {rightCoord}')
        aboveCoord = (y - 1) % HEIGHT
        belowCoord = (y + 1) % HEIGHT
        print(f'{"":15}above = {aboveCoord}, y = {y}, below = {belowCoord}')
```

5.5 Count the number of living neighbors

- The rules relate to the number of living neighbors. We need to count them for every `currentCell[x][y]`. We use the coordinates we just computed to look at everyone one of the eight neighbors:

```
# count number of living neighbors:
numNeighbors = 0
if currentCells[leftCoord][aboveCoord]=='#':
    numNeighbors += 1 # top-left neighbor alive
if currentCells[x][aboveCoord]=='#':
    numNeighbors += 1 # top neighbor alive
if currentCells[rightCoord][aboveCoord]=='#':
    numNeighbors += 1 # top-right neighbor alive
if currentCells[leftCoord][y]=='#':
    numNeighbors += 1 # left neighbor alive
if currentCells[rightCoord][y]=='#':
    numNeighbors += 1 # right neighbor alive
if currentCells[leftCoord][belowCoord]=='#':
   numNeighbors += 1 # bottom-left neighbor alive
if currentCells[x][belowCoord]=='#':
   numNeighbors += 1 # bottom neighbor alive
if currentCells[rightCoord][belowCoord]=='#':
   numNeighbors += 1 # bottom-right neighbor alive
```

- The variable `numNeighbors` now contains the number of living neighbors of each cell.

5.6 Apply Conway's rules for the next generation

- `nextCells` contains the next generation's cells. We apply three rules to `currentCells[x][y]` for both currently living or dead cells and copy the result to `nextCells`:

  1. Living cells with 2 or 3 neighbors stay alive
  2. Dead cells with 3 neighbors become alive
  3. Every other cell either dies or stays dead

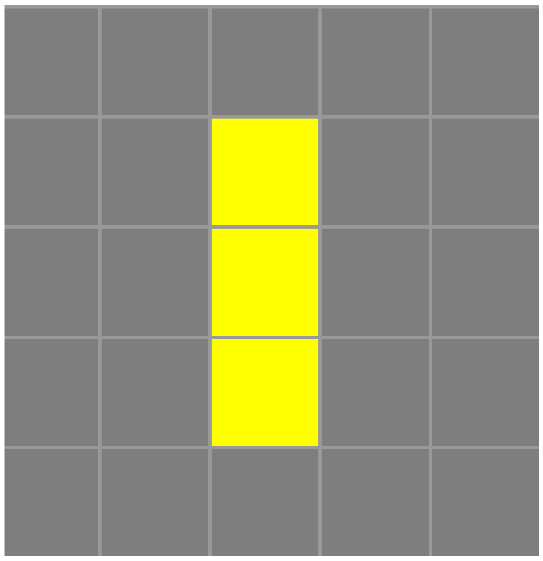
```
# set cell based on Conway's Game of Life rules:
if currentCells[x][y] == 'O':
    if numNeighbors==2 or numNeighbors==3:
        # living cells with 2-3 neighbors live:
        nextCells[x][y] = 'O'
    else:
        nextCells[x][y] = ' '
else:
    # dead cells with 3 neighbors become alive:
    if numNeighbors == 3:
        nextCells[x][y] = 'O'
    else:
        nextCells[x][y] = ' '
```

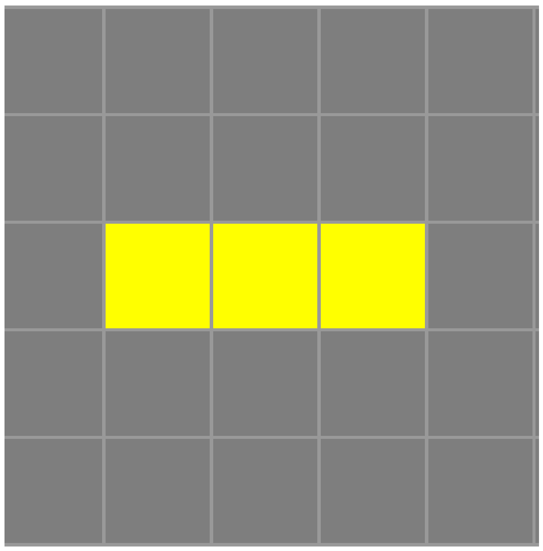
5.7 Take a short time out

- Before the next run through all cells, still within the infinite loop, we pause execution for 1 second to suppress flickering:

```
# add a 1-sec pause to reduce flickering
  time.sleep(1)
```

- Otherwise, there is no exit condition, the automata will live, die and replicate forever until they stabilize and the rules will not lead to a change anymore, or only to small changes:





5.8 Moving patterns: 'glider'

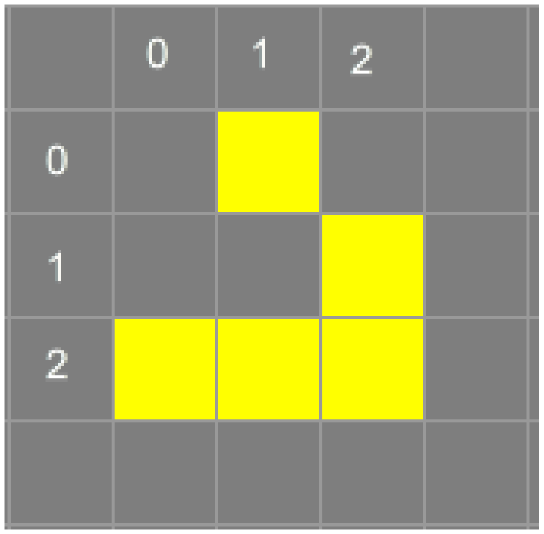
- To create the pattern shown at the start, the 'glider', which goes through 4 states before it repeats, replace

  ```
  if random.randint(0,1) == 0
  ```

  with this line:

```
if (x, y) in ((1, 0), (2, 1), (0, 2), (1, 2), (2, 2)):
```

- This brings only the cells of the 'glider' starting state to life:



- Download `glider.py` and run it in a terminal: github.com/birkenkrahe/py/blob/main/src/glider.p
- It's fun to experiment with other patterns. The free book by Johnston and Greene (2022) contains a lot of patterns for reproduction.

6 Conway's Game of Life using dictionaries

- Here's Sweigart's code using dictionaries. You can download it from here, or follow the author while he's coding on YouTube (Sweigart, 2021):

```
"""Conway's Game of Life, by Al Sweigart al@inventwithpython.com
The classic cellular automata simulation. Press Ctrl-C to stop.
More info at: https://en.wikipedia.org/wiki/Conway%27s_Game_of_Life
This code is available at https://nostarch.com/big-book-small-python-programming
Tags: short, artistic, simulation"""

import copy, random, sys, time

# Set up the constants:
```

```
WIDTH = 79   # The width of the cell grid.
HEIGHT = 20  # The height of the cell grid.

# (!) Try changing ALIVE to '#' or another character:
ALIVE = 'O'  # The character representing a living cell.
# (!) Try changing DEAD to '.' or another character:
DEAD = ' '   # The character representing a dead cell.

# (!) Try changing ALIVE to '|' and DEAD to '-'.

# The cells and nextCells are dictionaries for the state of the game.
# Their keys are (x, y) tuples and their values are one of the ALIVE
# or DEAD values.
nextCells = {}
# Put random dead and alive cells into nextCells:
for x in range(WIDTH):  # Loop over every possible column.
    for y in range(HEIGHT):  # Loop over every possible row.
        # 50/50 chance for starting cells being alive or dead.
        if random.randint(0, 1) == 0:
            nextCells[(x, y)] = ALIVE  # Add a living cell.
        else:
            nextCells[(x, y)] = DEAD  # Add a dead cell.

while True:  # Main program loop.
    # Each iteration of this loop is a step of the simulation.

    print('\n' * 50)  # Separate each step with newlines.
    cells = copy.deepcopy(nextCells)

    # Print cells on the screen:
    for y in range(HEIGHT):
        for x in range(WIDTH):
            print(cells[(x, y)], end='')  # Print the # or space.
            print()  # Print a newline at the end of the row.
            print('Press Ctrl-C to quit.')

    # Calculate the next step's cells based on current step's cells:
    for x in range(WIDTH):
        for y in range(HEIGHT):
            # Get the neighboring coordinates of (x, y), even if they
        # wrap around the edge:
        left  = (x - 1) % WIDTH
        right = (x + 1) % WIDTH
        above = (y - 1) % HEIGHT
        below = (y + 1) % HEIGHT

        # Count the number of living neighbors:
        numNeighbors = 0
        if cells[(left, above)] == ALIVE:
            numNeighbors += 1  # Top-left neighbor is alive.
        if cells[(x, above)] == ALIVE:
            numNeighbors += 1  # Top neighbor is alive.
        if cells[(right, above)] == ALIVE:
            numNeighbors += 1  # Top-right neighbor is alive.
        if cells[(left, y)] == ALIVE:
            numNeighbors += 1  # Left neighbor is alive.
        if cells[(right, y)] == ALIVE:
            numNeighbors += 1  # Right neighbor is alive.
        if cells[(left, below)] == ALIVE:
            numNeighbors += 1  # Bottom-left neighbor is alive.
        if cells[(x, below)] == ALIVE:
            numNeighbors += 1  # Bottom neighbor is alive.
        if cells[(right, below)] == ALIVE:
            numNeighbors += 1  # Bottom-right neighbor is alive.

        # Set cell based on Conway's Game of Life rules:
        if cells[(x, y)] == ALIVE and (numNeighbors == 2
                                       or numNeighbors == 3):
            # Living cells with 2 or 3 neighbors stay alive:
                nextCells[(x, y)] = ALIVE
        elif cells[(x, y)] == DEAD and numNeighbors == 3:
            # Dead cells with 3 neighbors become alive:
            nextCells[(x, y)] = ALIVE
        else:
            # Everything else dies or stays dead:
            nextCells[(x, y)] = DEAD

try:
    time.sleep(1)  # Add a 1 second pause to reduce flickering.
except KeyboardInterrupt:
    print("Conway's Game of Life")
    print('By Al Sweigart al@inventwithpython.com')
    sys.exit()  # When Ctrl-C is pressed, end the program.
```

7 Conway's Game of Life with NumPy and matplotlib

- See here for the notebook (GitHub). You need to download the Python source code and run it on the terminal or in IDLE.
- This Python program sets up an NxN grid (in this case, 100x100), randomly populates it with cells that are either "on" or "off", and then updates the grid over time according to Conway's Game of Life rules.
- The program:

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

def update(frameNum, img, grid, N):
    # Copy grid to apply rules
    newGrid = grid.copy()

    # Loop through each cell in the grid
    for i in range(N):
        for j in range(N):
            # Compute the sum of the eight neighbors
            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)

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

# Grid size and animation frames
N = 100
ON = 255
OFF = 0
vals = [ON, OFF]

# Populate grid with random on/off states
grid = np.random.choice(vals, N*N, p=[0.2, 0.8]).reshape(N, N)

# Create the figure and axis objects
fig, ax = plt.subplots()

# Display the grid as an image
img = ax.imshow(grid, interpolation='nearest')

# Animate
ani = animation.FuncAnimation(fig, update, fargs=(img, grid, N, ),
                              frames=10, interval=50, save_count=50)

# Display
plt.show()
```

- Analysis:

  1. Import modules:

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

     `numpy` is used for handling arrays efficiently. In Conway's Game of Life, the world is represented as a grid of cells, which is essentially a two-dimensional array. `matplotlib` is used for plotting, and we are using its sub-module animation to create animations.

2. Define the update function:

```
def update(frameNum, img, grid, N):
    newGrid = grid.copy()

    for i in range(N):
        for j in range(N):
            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]) / 2

            if grid[i, j] == ON:
                if (total < 2) or (total > 3):
                    newGrid[i, j] = OFF
            else:
                if total == 3:
                    newGrid[i, j] = ON

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

   - This function, called `update`, is executed for each frame of the animation. It takes four arguments: `frameNum` (the current frame number), `img` (the image plot object for displaying the current state of the grid), `grid` (the current state of the grid), and `N` (the size of the grid).
   - `newGrid` is a (shallow, i.e. reference) copy of the current grid. This copy is used to store the next state of the grid.
   - The nested `for` loops iterate through each cell in the grid. For each cell, the function calculates the total number of live neighbors (dividing by 255 to normalize to 1 for live cells, since live cells are represented by 255 or white).
   - It then applies Conway's Game of Life rules to decide whether each cell should be alive or dead in the next state. The changes are stored in `newGrid`.
   - `img.set_data(newGrid)` updates the image plot object with the new grid state.
   - `grid[:]  = newGrid[:]` updates the actual grid with the new state.

3. Initialize grid and constants:

   ```
   N = 100
   ON = 255
   OFF = 0
   vals = [ON, OFF]
   grid = np.random.choice(vals, N*N, p=[0.2, 0.8]).reshape(N, N)
   ```

   – `N` represents the size of the grid (100x100 in this case).
   – `ON` and `OFF` are constants used to represent the states of the cells (255 for on/alive/white and 0 for off/dead/black).
   – `grid` is initialized as a two-dimensional array with `random` values of `ON` and `OFF`. The `np.random.choice` function is used to fill the grid with a 20% chance of a cell being alive (`ON`) and 80% chance of being dead (`OFF`).

4. Set up the plot:

   ```
   fig, ax = plt.subplots()
   img = ax.imshow(grid, interpolation='nearest')
   ```

   – `plt.subplots` creates a new figure and a set of subplots. In this case, we only have one subplot (which is the default), so it effectively just creates a new figure for the animation.
   – `ax.imshow(grid, interpolation='nearest')` displays the data in grid as an image. The parameter `interpolation='nearest'` specifies that no interpolation should be done - the value of each cell should be displayed as-is. This is stored in the variable `img`.

5. Create the animation:

   ```
   ani = animation.FuncAnimation(fig,
                                 update,
                                 fargs=(img, grid, N, ),
                                 frames=10,
                                 interval=50,
                                 save_count=50)
   ```

   – `animation.FuncAnimation` is a function from `matplotlib.animation` that creates an animation by repeatedly calling a function (`update` in this case).
   – The `fig` argument specifies the figure object on which to draw the animation.

  - `update` is the function that will be called for each frame of the animation.
  - `fargs` is a tuple of arguments that will be passed to update each time it is called.
  - `frames` specifies the number of frames in the animation (in this example, the animation will have 10 frames).
  - `interval` is the delay between frames in milliseconds.
  - `save_count` is just an optimization that tells the animation to keep the last 50 frames in memory.

6. Display the animation:

   ```
   plt.show()
   ```

   - Finally, `plt.show()` is called to display the animation. This opens a window that shows the animation of the grid evolving over time according to the rules of Conway's Game of Life.

7. Extensions: this animation only runs for 10 frames. You can experiment with different parameters - number of frames, grid size, different initial configuration of the grid.

8 When to use lists vs. arrays

In Python, lists and arrays are used to store multiple items in a single variable. However, they are used in different scenarios due to their unique characteristics and functionalities. Here are some reasons why you might use lists instead of arrays:

1. Flexibility: Lists in Python are more flexible than arrays. They can store different types of data (integers, strings, floats, other lists, etc.) in the same list, while arrays (if you're referring to arrays from the array module or NumPy arrays) typically require all elements to be of the same type.

2. Built-in Methods: Python lists come with a variety of built-in methods for manipulation, such as `append()`, `insert()`, `remove()`, `pop()`, `count()`, `sort()`, `reverse()`, etc. While arrays also have some of these methods, lists generally have more built-in functionality.

3. Ease of Use: Lists are a built-in type in Python, which makes them easy to use and understand for beginners. Arrays, on the other hand, require

importing a separate module (like the `array` module or NumPy), which can add an extra layer of complexity.

4. No Need for Vectorized Operations: If you don't need to perform mathematical operations on the entire data structure (which is where arrays shine), then a list can be a simpler and more efficient choice.

5. Memory: Lists are more memory efficient if you're storing non-numeric data types.

However, if you're doing numerical computations, especially on large datasets, arrays (particularly NumPy arrays) are often a better choice due to their efficiency and the powerful operations they support.

Python also has: **dictionaries** (mutable unordered collection of elements whose items are named rather than indexed), **sets** (mutable, unordered collections of unique elements) and **tuples** (immutable ordered collection of elements) - and beyond that, there are Object-oriented structures like classes (with attributes, like `np.shape`, and methods, like `np.array`) which are used to implement the other data structures.

9 References

- Ilievski, V. (2021). Simple but Stunning: Animated Cellular Automata in Python. URL: isquared.digital

- Johnson, N. and Greene, D. (2022). Conway's Game of Life - Mathematics and Construction. URL: conwaylife.com/book/

- OpenAI (2023). ChatGPT May 24 Version. URL: chat.openai.com.

- Sweigart, A. (2019). Automate the Boring Stuff with Python. NoStarch. URL: automatetheboringstuff.com

- Sweigart, A. (2021). Calm Programming - Conway's Game of life. URL: youtu.be/Vn8Mug5w7sw.