Coding Example: Game of life (Python approach)

This lesson discusses the case study Game of life and explains its solution using Python implementation.

We'll cover the following

Problem Description

The Game of Life is a cellular automaton devised by the British mathematician John Horton Conway in 1970. It is the best-known example of a cellular automaton. The “game” is actually a zero-player game, meaning that its evolution is determined by its initial state, needing no input from human players. One interacts with the Game of Life by creating an initial configuration and observing how it evolves.

The universe of the Game of Life is an infinite two-dimensional orthogonal grid of square cells, each of which is in one of two possible states, live or dead. Every cell interacts with its eight neighbors, which are the cells that are directly horizontally, vertically, or diagonally adjacent.

Rules

At each step in time, the following transitions occur:

1. Any live cell with fewer than two live neighbors dies, as if by underpopulation.
2. Any live cell with more than three live neighbors dies, as if by overcrowding.

3. Any live cell with two or three live neighbors lives, unchanged, to the next generation.
4. Any dead cell with exactly three live neighbors becomes a live cell.

The initial pattern constitutes the 'seed' of the system. The first generation is created by applying the above rules simultaneously to every cell in the seed – births and deaths happen simultaneously, and the discrete moment at which this happens is sometimes called a tick. (In other words, each generation is a pure function of the one before.) The rules continue to be applied repeatedly to create further generations.

(Excerpt from the Wikipedia entry on the Game of Life)

Python implementation

In pure Python, we can code the Game of Life using a list of lists representing the board where cells are supposed to evolve. Such a board will be equipped with a border of 0 that allows accelerating things a bit by avoiding having specific tests for borders when counting the number of neighbors. We could have used the more efficient python array interface but it is more convenient to use the familiar list object.

```
# generate a two-dimensional grid Z
# Each index value indicates one of two possible states
# 0 means active, 1 means dead
   Z = [[0,0,0,0,0,0],
        [0,0,0,1,0,0],
        [0,1,0,1,0,0],
        [0,0,1,1,0,0],
        [0,0,0,0,0,0],
        [0,0,0,0,0,0]]
```

Taking the border into account, counting neighbors then is straightforward:

```
def compute_neighbours(Z): #compute neighbors of Z for each cell
  shape = len(Z), len(Z[0])
  N  = [[0,]*(shape[0]) for i in range(shape[1])]
  for x in range(1,shape[0]-1):
    for y in range(1,shape[1]-1):
      # All possible scenarios of neighbors
      N[x][y] = Z[x-1][y-1]+Z[x][y-1]+Z[x+1][y-1] \
              + Z[x-1][y]+Z[x+1][y] \
```

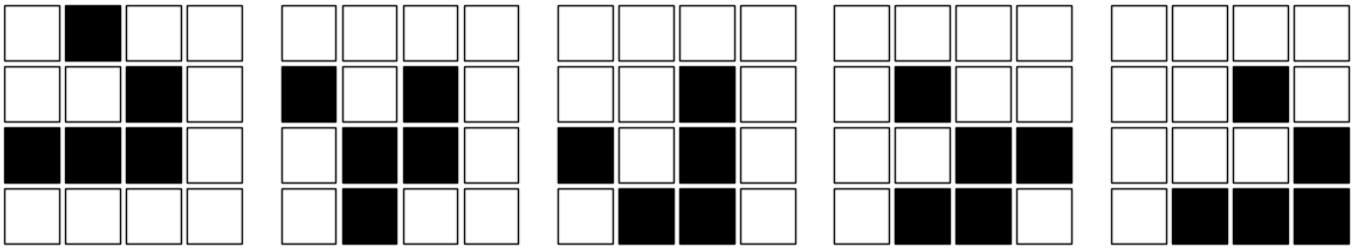
```
                + Z[x-1][y+1]+Z[x][y+1]+Z[x+1][y+1]
    return N
```

To iterate one step in time, we then simply count the number of neighbors for each internal cell and we update the whole board according to the four aforementioned rules:

```
def iterate(Z):
  N = compute_neighbours(Z) #call the compute_neighbors(Z) function
  for x in range(1,shape[0]-1): #iterate and update the grid according the forementioned rule
    for y in range(1,shape[1]-1):
      if Z[x][y] == 1 and (N[x][y] < 2 or N[x][y] > 3):
        Z[x][y] = 0
      elif Z[x][y] == 0 and N[x][y] == 3:
        Z[x][y] = 1
  return Z
```

The figure below shows four iterations on a 4x4 area where the initial state is a glider, a structure discovered by Richard K. Guy in 1970.

The glider pattern is known to replicate itself one step diagonally in 4 iterations.



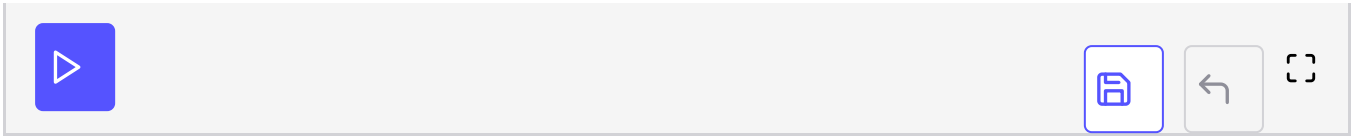
Complete Solution

Given below is the complete solution to this problem.

```
# -----------------------------------------------------------------------------
# From Numpy to Python
# Copyright (2017) Nicolas P. Rougier - BSD license
# More information at https://github.com/rougier/numpy-book
# -----------------------------------------------------------------------------

#compute the neighbors of Z for each internal cell
def compute_neighbours(Z):
    shape = len(Z), len(Z[0])
    N = [[0, ]*(shape[0]) for i in range(shape[1])]
    for x in range(1, shape[0]-1):
        for y in range(1, shape[1]-1):
            N[x][y] = Z[x-1][y-1]+Z[x][y-1]+Z[x+1][y-1] \
                    + Z[x-1][y]           +Z[x+1][y]   \
                    + Z[x-1][y+1]+Z[x][y+1]+Z[x+1][y+1]
    return N
```

```
#iterate through the grid and update the grid according to forementioned rules
def iterate(Z):
    shape = len(Z), len(Z[0])

    N = compute_neighbours(Z) #call the compute_neighbor(Z) function
    for x in range(1, shape[0]-1):
        for y in range(1, shape[1]-1):
            if Z[x][y] == 1 and (N[x][y] < 2 or N[x][y] > 3):
                Z[x][y] = 0
            elif Z[x][y] == 0 and N[x][y] == 3:
                Z[x][y] = 1
    return Z

if __name__ == '__main__':
    import matplotlib.pyplot as plt
    from matplotlib.patches import Rectangle

    #create a two dimensional grid having active and inactive states
    Z = [[0, 0, 0, 0, 0, 0],
         [0, 0, 0, 1, 0, 0],
         [0, 1, 0, 1, 0, 0],
         [0, 0, 1, 1, 0, 0],
         [0, 0, 0, 0, 0, 0],
         [0, 0, 0, 0, 0, 0]]

    # specify the size of the plot
    figure = plt.figure(figsize=(12, 3))

    # specify the label name of the plotted figures
    labels = ("Initial state",
              "iteration 1", "iteration 2",
              "iteration 3", "iteration 4")
    # 4 iterations
    for i in range(5):
        ax = plt.subplot(1, 5, i+1, aspect=1, frameon=False)

        for x in range(1, 5):  # make a grid of size 4*4
            for y in range(1, 5):
                if Z[x][y] == 1:
                    facecolor = 'black'
                else:
                    facecolor = 'white'
                rect = Rectangle((x, 5-y), width=0.9, height=0.9,
                                 linewidth=1.0, edgecolor='black',
                                 facecolor=facecolor)
                ax.add_patch(rect)
        ax.set_xlim(.9, 5.1)
        ax.set_ylim(.9, 5.1)
        ax.set_xticks([])
        ax.set_yticks([])
        ax.set_xlabel(labels[i])

        for tick in ax.xaxis.get_major_ticks():
            tick.tick1On = tick.tick2On = False
        for tick in ax.yaxis.get_major_ticks():
            tick.tick1On = tick.tick2On = False

        iterate(Z) #call the iterate method

    plt.tight_layout() #automatically adjusts subplot params so that the subplot(s) fits in t
    plt.savefig("output/glideroutput.png") #save the image
```

```
    plt.show() #plot the image
```



In the next lesson, we will try to come up with another solution using the NumPy approach!