**Game of Life with Python**

**Designed by British mathematician John H. Conway, Game of Life is a solitaire game that simulates the rise, fall, and cycle of a community of living organisms. In this section of free Python projects, we introduce you to the implementation of Game of Life with Python .**

**This game was one of the first examples of problems in modern mathematics called cellular automata .**

**Game of Life**

**The game uses a rectangular grid of cells of infinite size, where each empty cell is occupied by a living creature. Occupied cells are said to be alive, while empty cells are said to be dead. The game is played in a certain period, and each turn, a new "generation" is created based on the arrangement of organisms in the current configuration .**

**The state of a cell in the next generation is determined by applying the following four basic rules to each cell of the current configuration :**

* **If a cell is alive and has two or three living neighbors, the cell survives in the next generation .**
* **A living cell that has no living neighbors, or only one living neighbor, dies in isolation in the next generation .**
* **A living cell that has four or more living neighbors will die in the next generation due to overpopulation .**
* **A dead cell with exactly three living neighbors gives birth and survives the next generation .**

**Implementation of Game of Life in Python**

**Game of Life begins with a user-supplied initial setup. Successive generations are created by applying a set of rules simultaneously to each cell in the network. As populations of organisms change, increase, or eventually disappear, interesting patterns can emerge. Now let's see how we can implement Game of Life as a free Python project :**

**class game\_of\_life :**

**def gameOfLife ( self , board : List [ List [ int ]]) -> None :**

**"""**

**Do not return anything, modify the board in-place instead.**

**"""**

**# Neighbors array to find 8 neighboring cells for a given cell**

**neighbors = [( 1 , 0 ), ( 1 , - 1 ), ( 0 , -1 ) , ( -1 , -1 ) , \_ ( -1 , 0 ) , ( -1 , 1 ) , ( 0 , 1 ), ( 1 , 1 )]**

**rows = len ( board )**

**cols = len ( board [ 0 ])**

**# Create a copy of the original board**

**copy\_board = [[ board [ row ][ col ] for col in range ( cols )] for row in range ( rows )]**

**# Iterate through the board cell by cell.**

**for row in range ( rows ):**

**for col in range ( cols ):**

**# For each cell count the number of living neighbors.**

**live\_neighbors = 0**

**for neighbor in neighbors :**

**r = ( row + neighbor [ 0 ])**

**c = ( col + neighbor [ 1 ])**

**# Check the validity of the neighboring cell and if it was originally a live cell.**

**# The evaluation is done against the copy, since that is never updated.**

**if ( r < rows and r >= 0 ) and ( c < cols and c >= 0 ) and ( copy\_board [ r ][ c ] == 1 ):**

**live\_neighbors += 1**

**# Rule 1 or Rule 3**

**if copy\_board [ row ][ col ] == 1 and ( live\_neighbors < 2 or live\_neighbors > 3 ):**

**board [ row ][ col ] = 0**

**# Rule 4**

**if copy\_board [ row ][ col ] == 0 and live\_neighbors == 3 :**

**board [ row ][ col ] = 1**

Amirshayan Jalili

[Shayan138190@gmail.com](mailto:Shayan138190@gmail.com)

<https://github.com/Amirshayan2002>