Building Conway's Game of Life in Python Project

Case Description

In this Building Conway's Game of Life in Python project, you get to implement a simulation of the Game of Life, invented by mathematician John Horton Conway. It’s an excellent opportunity to apply your programming skills in Python while exploring fascinating concepts in cellular automata, emergence, and complexity theory.

Conway’s Game of Life is a zero-player game requiring only an initial state and no further input. In its original setting, the game takes place on an infinite grid of square cells, each in one of two possible states: live or dead. Every cell interacts with its eight neighbors (horizontally, vertically, or diagonally adjacent cells). Starting at the initial state, the game evolves according to the following rules:

- Any live cell with two or three live neighbors survives. Otherwise, a cell dies due to loneliness (with no or only one neighbor) or overpopulation (with four or more neighbors).
- Any dead cell with (exactly) three live neighbors becomes a live cell. A dead cell with any other number of neighbors remains dead.

In this project, we restrict the infinite grid to one with finite pre-defined dimensions.

Despite its simplicity, the Game of Life has the power to create complex behaviors, which have implications for all fields—from physics and mathematics to philosophy and art.

Project requirements

For this Conway's Game of Life project, you’ll work with Python 3 or newer and an IDE of your choice (Jupyter Notebook, Spyder, PyCharm, Visual Studio, etc.) Still, note that the file attached to this project is a Jupyter Notebook.

Project files

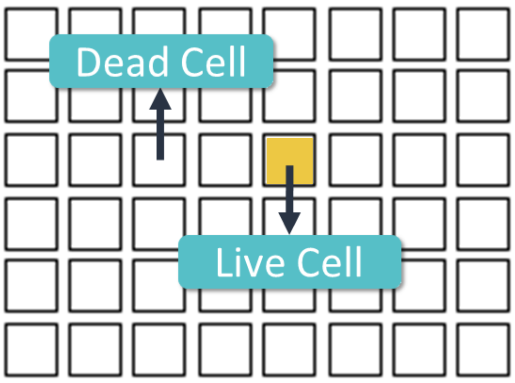
The Game of Life.ipynb file contains a skeleton of the code.

File

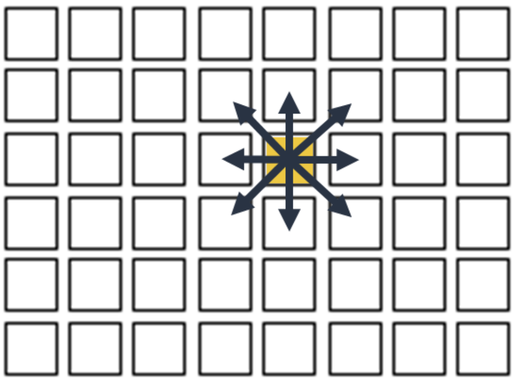
GUIDE

Create the Class and Its _init_ Method

Conway's Game of Life is a zero-player game requiring only an initial state and no further input. It occurs on a grid of square cells, each in one of two possible states: live or dead.

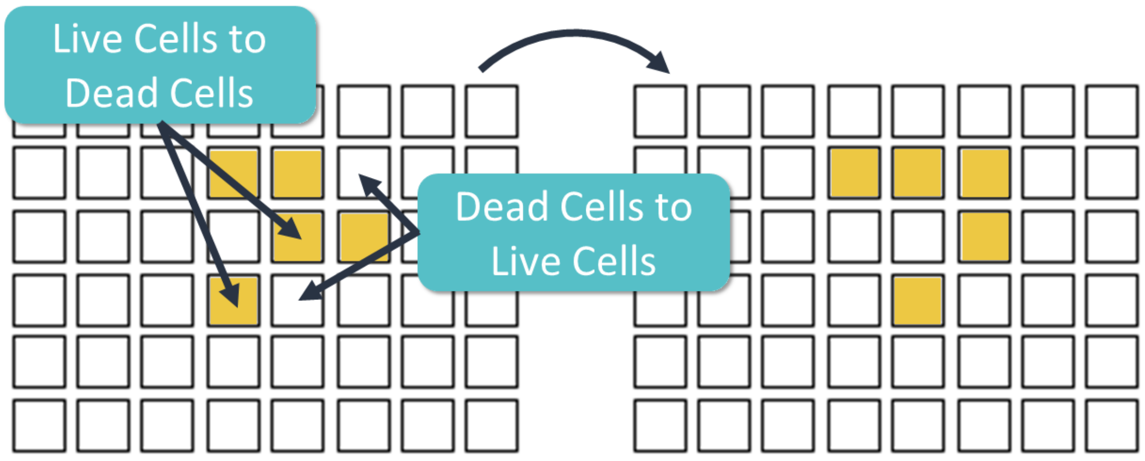


Every cell interacts with its eight neighbors (horizontally, vertically, or diagonally adjacent cells).



Starting at the initial state, the neighbors interact according to the following rules:

- Any live cell with two or three live neighbors survives. Otherwise, a cell dies due to loneliness (with no or only one neighbor) or overpopulation (with four or more neighbors).
- Any dead cell with (exactly) three live neighbors becomes a live cell. A dead cell with any other number of neighbors remains dead.



In this project, we will implement Conway's Game of Life in Python.

The skeleton of your project includes the following:

```
class GameOfLife(object):


    def __init__(self, x_dim, y_dim):
        # Initialize a 2D list with dimensions x_dim by y_dim filled
with zeros.
        pass


    def get_grid(self):
        # Implement a getter method for your grid.
        pass


    def print_grid(self):
        # Implement a method to print out your grid in a human-
readable format.
        pass


    def populate_grid(self, coord):
        # Given a list of 2D coordinates (represented as
tuples/lists with 2 elements each),
        # set the corresponding elements in your grid to 1.
        pass
    def make_step(self):
        # Implement the logic to update the game state according to
the rules of Conway's Game of Life.
        pass


    def make_n_steps(self, n):
        # Implement a method that applies the make_step method n
times.
        pass


    def draw_grid(self):
        # Draw the current state of the grid.
        pass
```

You might've noticed the `pass` keyword. `pass` is a unique statement in Python that does nothing. It can be used as a placeholder for future code. Nothing happens when the `pass` statement is executed, but you avoid getting an error when empty code is not allowed. In the context of this project, it's used in the class skeleton to indicate where you need to add your method implementations.

Your first task is to define the initial state of your `GameOfLife` class. You must use the provided skeleton as a guide to implement the `__init__` method (the class constructor).

In this method, you'll set up the initial game state–an empty grid with dimensions `x_dim` by `y_dim`. For this project, we'll represent this grid as a 2D list, with each element being either a 0 or a 1. A 0 illustrates a dead cell, and a 1 represents a live cell. All cells should be dead at the start, so your grid should be filled with zeros.

Good luck with your first step into bringing the Game of Life to... life.

1. **Define the _init_ method:** This method should take three arguments: `self`, `x_dim`, and `y_dim`.

```
def __init__(self, x_dim, y_dim):
```

2. **Initialize life_grid:** Within the `__init__` method, initialize an attribute; let's call it `life_grid`, as a 2D list with dimensions `x_dim` by `y_dim`. You should fill this grid with zeros, indicating that all cells are initially dead.
3. **Test the method:** Remember to test your constructor to ensure it works as expected. You can do this by creating an instance of your `GameOfLife` class with specific dimensions and then checking that `life_grid` has the correct measurements and is filled with zeros.

Create a Method to Return the Grid

Now that you have successfully initialized your grid in the constructor, your next task is implementing a method named `get_grid()` to return this grid. This method is simple yet crucial for accessing the game's current state outside the class.

1. **Define the get_grid method:** This method should take only one argument: `self`–standard in defining class methods.
2. **Return life_grid:** In this method, return the `life_grid` attribute you defined in the `__init__` method. This will give you access to the current state of the game grid when this method is called on an instance of the `GameOfLife` class.
3. **Test the method:** Ensure it works properly by calling it through the dot-notation. The grid should appear on the screen.

Create a Method to Print the Grid

Your game is starting to take shape. Now that you can create and retrieve your grid, it would be helpful to visualize it. This is where the `print_grid()` method comes into play. This method will print your current game grid in a user-friendly way, making it easier to track the progress of the game.

```
0 | 0 | 0 |
- - - - - -
0 | 0 | 0 |
- - - - - -
0 | 0 | 0 |
- - - - - -
```

1. **Define the print_grid method:** This method should only take `self` as an argument like before.
2. **Loop over the grid:** You'll need to iterate over each row and each cell within that row of your `life_grid` attribute. A nested loop (a loop within a loop) will be necessary. The outer loop should iterate over the grid rows, while the inner loop should iterate over each cell within a given row.

```
for i in range(# num_rows #):
    for j in range(# num_columns #):
```

3. **Print each cell:** Print the current cell's value within the inner loop. Additionally, you may want to print a vertical bar (|) or some other character after each cell to make the output easier to read. To prevent Python from automatically starting a new line after each cell, use the `end=' '` parameter in your `print()` statement.

```
0 | 0 | 0 |
0 | 0 | 0 |
0 | 0 | 0 |
```

4. **Print row separators:** After each row has been printed—i.e., at the end of each iteration of the outer loop—print a separator line to help distinguish between different grid rows. This could be a line of dashes (-), equals signs (=), or any other character you think would be helpful.

```
0 | 0 | 0 |
- - - - - -
0 | 0 | 0 |
- - - - - -
0 | 0 | 0 |
- - - - - -
```

5. **Test the method:** After implementing this method, test it by calling `print_grid()` on an instance of your `GameOfLife` class. It should correctly print the current state of the game grid, with each row separated by a line of your chosen character.

Create a Method to Populate the Grid

So far, we've created the constructor of our project and implemented a getter that returns the grid object and a print method that prints the grid on the screen.

The next task is to implement the `populate_grid()` method, which allows you to initialize the game state by specifying which cells in the grid should start as 'alive' (represented by a 1). This will help set the stage for the Game of Life.

Note that this is the only instance of the game where we can intervene and change the outcome by choosing which cells on the grid start as alive and which ones as dead. After executing this step, the initial state evolves according to the game's rules, and no further user input is required.

1. **Define the populate_grid method:** This method should take two arguments: `self` and `coord`. The `coord` parameter is a list of tuples, where each tuple represents the coordinates [in the format (row, column)] of a cell that should be marked as 'alive.'
2. **Assign values to the grid:** Loop over each tuple in the coord list. For each tuple, assign the value 1 to the cell in `self.life_grid` corresponding to the provided coordinates.
3. **Return the modified grid:** After all the coordinates in `coord` have been processed, return `life_grid`.
4. **Test the method:** Now that you've implemented the `populate_grid()` method, it's time to ensure it works as expected. Create a new `GameOfLife` object with a specified grid size, and then call `populate_grid()` with a list of coordinates. After you've called the method, use the `get_grid()` or `print_grid()` methods to inspect the state of the grid. Confirm that the cells at your specified coordinates are marked as 'alive,' while the rest of the grid remains 'dead.'

Create a Method to Make a Step in the Game of Life

Now that the grid is populated with live cells, we must create a method that advances in the Game of Life. We'll start with a technique that moves only a single step; call it `make_step()`. It should iterate over each cell in the grid, evaluate its neighbors, and update the cell's state based on the Game of Life rules.

1. **Define the make_step method:** This method should only take `self` as an argument.
2. **Initialize a sum_grid variable:** The `sum_grid` variable will store the sum of the neighboring cells for each cell in the `life_grid`. This can be a 2D list with the exact dimensions as `life_grid`, initialized with zeros. Each cell in `sum_grid` will represent the sum of the neighboring cells.
3. **Iterate over each cell:** You'll need two nested `for` loops to iterate over each cell in `life_grid`. For each cell, you should also iterate over its neighbors. Remember that a cell's neighbors include the cells horizontally, vertically, and diagonally adjacent.

```
# Loop through each cell
for i in range(# num_rows #):
    for j in range(# num_columns #):

        # Loop through its neighbors
        for a in # neighboring rows #:
            for b in # neighboring columns #:
```

4. **Calculate the sum of neighbors:** For each cell, calculate the sum of the states of its neighbors and store this value in the corresponding cell in `sum_grid`. Ensure you properly handle the cells that have less than eight adjacent cells (the ones on the rim of the grid.)

```
# Loop through each cell
for i in range(# num_rows #):
    for j in range(# num_columns #):

        # Loop through its neighbors
        for a in # neighbors' rows #:
            for b in # neighbors' columns #:

                # Calculate the sum

        # Store the sum in the corresponding cell in sum_grid
```

5. **Update the cells:** After `sum_grid` has been filled, iterate over the cells in `life_grid` again and update their state based on the following rules of the Game of Life:
    - A live cell with fewer than two live neighbors dies.
    - A live cell with two or three live neighbors lives on to the next generation.
    - A live cell with more than three live neighbors dies.

    - A dead cell with exactly three live neighbors becomes a live cell.
6. **Return the updated grid:** After all the cells have been updated, return `life_grid`.
7. **Test the method:** Now that you've implemented the `make_step()` method, it's time to test it. After populating the `life_grid` with some initial configuration, call `make_step()`, and then use `get_grid()` or `print_grid()` methods to inspect the state of the grid. Confirm that the form of the grid updates correctly according to the Game of Life rules. Try this with different initial configurations and ensure the method behaves as expected.

A good and quick way to double-check your result is by visiting the website https://playgameoflife.com/ and setting the same initial conditions in both your program and the one on the website.

Create a Method to Make n Steps in the Game of Life

You've done most of the work by implementing the make_step method. Your next task is creating a new one, `make_n_steps`, designed to simulate the Game of Life for n steps. In other words, it repeatedly applies the rules of the Game of Life (i.e., the `make_step` method) n times to evolve the game. Realizing this new method would allow us to do something exciting: obtain a snapshot of the game at a specific step.

1. **Define the make_n_steps method:** This method should take two arguments (`self` and `n`), where `n` is the number of steps the game should be advanced.
2. **Loop n times:** You'll need to call your `make_step()` method `n` times. Use a simple `for` loop to achieve this.
3. **Return the updated grid:** After all the steps have been made, return `self.life_grid`.
4. **Test the method:** To test the `make_n_steps()` method, you should first populate `life_grid` with an initial configuration of live cells. Call `make_n_steps()` with some number `n`, and then use `get_grid()` or `print_grid()` methods to inspect the state of the grid. You should see that the game has evolved `n` steps from the initial state according to the rules of the Game of Life. Try this with different initial configurations and different numbers of steps to make sure the method is working as expected.

Create a Method to Draw the Grid

Now that we have implemented the main mechanics of the Game of Life, it would be great to have a visual representation of our game board. For this task, you need to implement the `draw_grid()` method. The method aims to provide a visual plot of the current state of the game board using the `matplotlib` library. This would serve as an improved version of the `print_grid()` method where we simply printed the grid.

In the method, you'll create a scatter plot where the x and y coordinates are the positions of the cells in the grid, and the color of each point is determined by whether the cell is alive or dead.

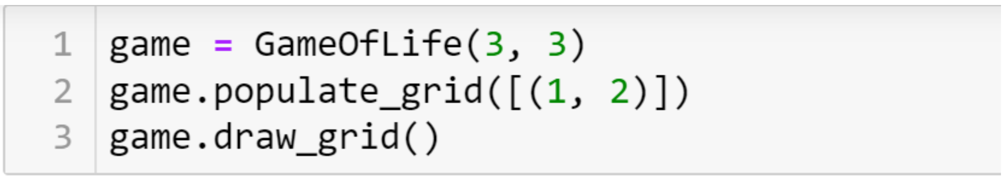
Remember that the grid's visualized version should look the same as the printed one. In other words, the positions of the live and dead cells in the scatter plot should match their positions when printed on the console.

Enjoy creating the game board!

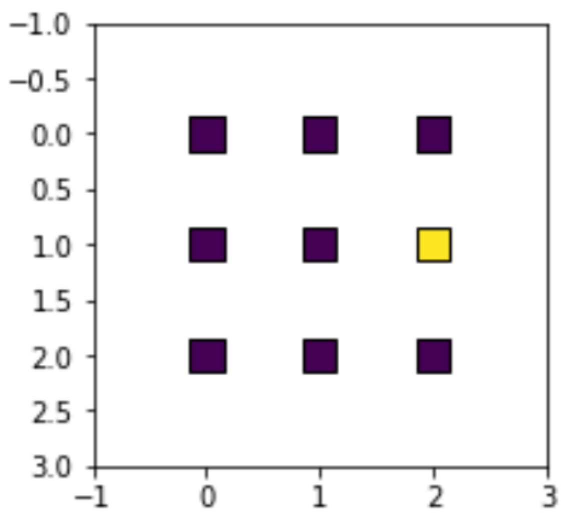
1. **Initialize the arrays for x and y coordinates:** In the beginning, create two empty lists (`x` and `y`) to hold the coordinates of the cells in the grid.
2. **Fill the coordinate arrays:** Run nested loops over the dimensions of the grid. For each cell, append the row and column indices to the `x` and `y` lists.

```
for i in range(# num_columns #):
    for j in range(# num_rows #):
        x.append(j)
        y.append(i)
```

3. **Set up the plot:** Initialize a figure (`fig`) and an axis (`ax`) object using `plt.subplots()`. You can specify the size of the figure.
4. **Draw the scatter plot:** Use `plt.scatter()` to create a scatter plot with `x` and `y` as the coordinates of the points. Use the values in the `life_grid` object to color the cells based on whether they're alive or dead. You can do this by setting `life_grid` as a value to the `c` argument of the `scatter()` method. Additionally, you can customize the scattered points' size, edge colors, and markers.
5. **Set the limits of the plot and invert the axes:** Set the limits of the x and y axes of the plot and invert any or both of them if necessary. To realize this, research the functions `xlim`, `ylim`, `invert_xaxis`, and `invert_yaxis`.
6. **Test the method:** After implementing the method, test it by creating a `GameOfLife` instance, populating the grid, making some steps, and drawing the grid. Compare the illustrated version with the printed version to ensure the positions of the live cells match in both versions.



```
game = GameOfLife(3, 3)
game.populate_grid([(1, 2)])
game.draw_grid()
```



Document the Code

Your progress is commendable. Great job!

Documenting your code is the final required task for completing the project. Proper documentation is essential for making your code understandable to others and reminding yourself what your code does when you revisit it in the future. Python provides a built-in way of writing documentation using comments known as docstrings.

Docstrings are a type of comment used at the beginning of functions, methods, classes, and modules to describe what they do. They are written between triple quotes, allowing for multiple lines of text.

Your task is to add docstrings to each method in your `GameOfLife` class.

To give you a starting point, note how to document the `populate_grid` method:

```
def populate_grid(self, coord):
    '''
    Populates the game grid with live cells at the specified coordinates.

    Parameters:
    coord: A list of tuples. Each tuple represents the (x, y) coordinates of a live cell.

    Returns:
    The updated life_grid with the new live cells.
    '''
    # Given a list of 2D coordinates (represented as tuples/lists with 2 elements each),
    # set the corresponding elements in your grid to 1.
```

In this docstring, the first line briefly describes what the method does. The Parameters section lists the input parameters and their types and explains what they represent. The Returns section describes what the function returns.

Add similar docstrings to the rest of the methods in your `GameOfLife` class. Remember, good documentation makes your code more accessible and maintainable for others and your future self. Explain clearly what the method or class does without explaining how it works. The reader can look at the code for that information.

Happy documenting!

1. **Understand the purpose of each method:** Start by reviewing each method in the `GameOfLife` class to understand what it does. You must be clear on the meaning of the methods to write accurate and helpful docstrings.
2. **Write the docstring for the populate_grid() method:** Let's begin with the `populate_grid()` method. This function takes a list of coordinates and places live cells (represented by 1s) at these locations in the `life_grid`. The first line of your docstring should concisely explain this. Then, describe the parameters and the return value.
3. **Write docstrings for the remaining methods:** Follow the same process to write docstrings for the other methods in your class. Remember to describe what the method does, its parameters, and what it returns.
4. **Review your docstrings:** Once you've written all your docstrings, go back through them and make sure they accurately describe what each method does. It may also be helpful to peer review your docstrings to make sure they make sense to someone who didn't write the code.
5. **Test your code:** After adding the docstrings, test your code to ensure everything works correctly. To do this, you can call the `help()` function on your class or any of its methods to display the docstrings. This also allows you to see how your docstrings will look to others who might use your code.

```
game = GameOfLife(5, 5)
help(game.populate_grid)
```

Quiz