Task

Introduction:

In this submission you should create a program that simulates cell life and death. This should be done using a model called Conway's Game of Life.

In short, the program will control a game board of any size, where each field in the board should contain a cell. A cell may be alive or dead. Simulation takes place over several generations using regular updates, where cells die or live depending on their environment.

The program will let the user observe the simulation generation for generation by drawing up the game board in the terminal together with additional information about which generation we are watching and how many cells currently live.

When you deliver this task, you must submit **all** python files you've used in the solution. It is important that you comment on how your solution works. In addition, **you should** comment on a separate document on what has been challenging and what has been easy through the assignment (between 5-7 lines are enough).

Game rules:

A new generation is created by all the cells in the board changing status depending on their neighboring cells. As neighboring cells considers all moving cells, both living and dead, are considered to be.

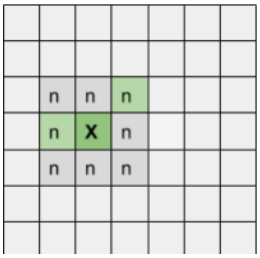


Illustration 1: A cell (X) and its neighboring cells (green cells are "living"). X has 8 neighboring cells, and 2 of them are alive.

The cell's new status is determined by the following rules:

- If the cell's current status is "alive":
  - At less than two living neighboring cells, the cell dies (*subpopulation*).
  - By two or three living neighboring cells, the cell will live on.
  - If the cell has more than three living neighboring cells, it will die (*overpopulation*)

- If the cell is "dead":
  - The cell's status becomes "live" (*reproduction*) if it has exactly three living neighbors.

Note that the update of the cell's status occurs **at the same time**! This means that we must determine the new status of all cells depending on the current status before the update actually occurs.

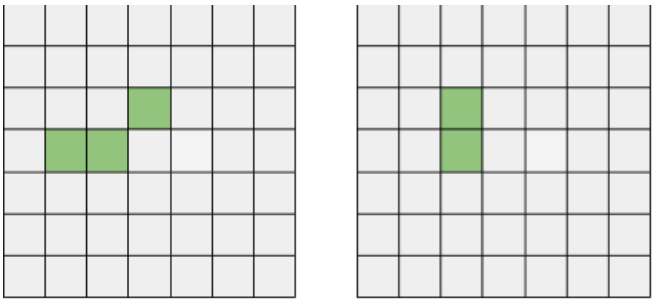
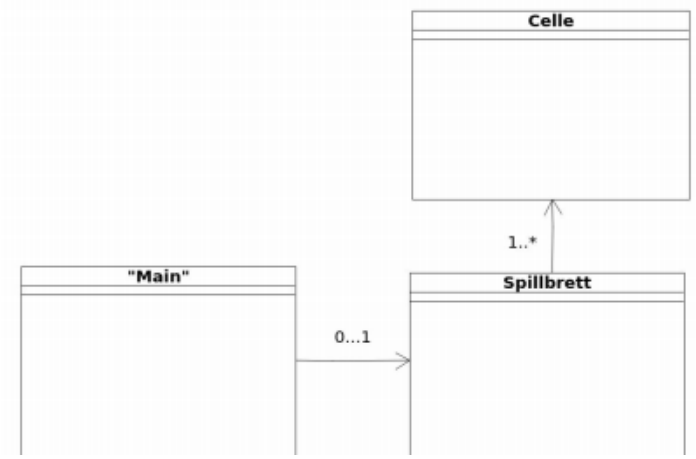


*Illustration 2: Two generations of cells in a 7 * 7-size game board*

Data Structure Drawing

Read the descriptions of the program structure under. Here you can see a UML diagram of the data structure:



Spillebrett=game board
Celle= cell

Structure (explaining the diagram above):
The program consists of three files with two classes in addition to a "main program", all of should must be delivered in each code file. The described classes must be named as described. The code sketch for the three files is attached(Appendix) to the task.

Cell
File Name: celle.py
The class describes a cell in the simulation. A cell must have a variable describing status (live / death).

1. Type a constructor for the class that creates the cell with "dead" status as the starting point.
2. Type the methods *settDoed* and *settLevende* which do not take any parameters, but which sets the status of the cell to "dead" and "living" respectively.
3. Type the method *erLevende* that returns the cell's status; True if the cell is alive and false otherwise. In addition, write a method that returns a character representation of the cell's status for use in drawing the board. If the cell is "alive", an "O" must be returned, but if it is dead, a dot will be returned.

See Appendix at the end of the document for code skeleton.

Game board
File name: spillebrett.py

This class describes a two-dimensional board containing cells. The game board will keep track of which cells will change status and update them for each generation.

1. Write the completed constructor for the class Game Board. 1. Type the completed constructor for the Game Board class. The constructor accepts dimensions on the game board and stores them in the instance variables *self_rader* and *self._kolonner.* The constructor must:

   a) Create a grid in the form of a two-dimensional (nested) list. The grid should filled with a number of cell objects equal to the number of rows multiply the number of columns.

   b) Create a variable that keeps track of the *generation number* and which should increases every time the board is updated.

2. The method *generer* goes through the grid and ensures that a random number of cells gets the status "alive." This is called a "seed" and represents the starting point, or "zero generation" for our cell simulation.

   a) Import *random* in your program. Create the method *generer,* which will allow each cell in the grid to have ⅓ chance of being alive. Random has a method randint (number1, number2) that returns a random number between these. It is counting from number1, to the number2. Example:
   random.randint (0.2)
   This method returns a random number that is between 0 to 2, i.e. 0, 1 or 2.

   b) Expand the class constructor to call the methon *generer*.

3. To show up and test the game board, type the method *tegnBrett*. This method should use a nested for-loop to print each element in the grid. Remember to test your program so far by creating a Game Board object and printing the board in a small test program. This can be done in the *main* file method main.py described further down in the task text. Tips for formatting printing:

   a) To avoid line breaks after each printout, you can finish printing with an empty string instead, like that:
   `print(arg, end="")`

   b) You may want to "clear" the terminal window between each printout. For example, you can do this by printing a dozen blank lines before printing the board.

4. To determine which cells will be "living" and "dead" in the next generation, we need to know the status of each cell's neighbor. The game board class therefore contains a method *finnNabo*. The method will receive two coordinates in the grid and return a list of all the neighbors of the cell. See the picture further up in the task for illustration. Take care of edges and corners.

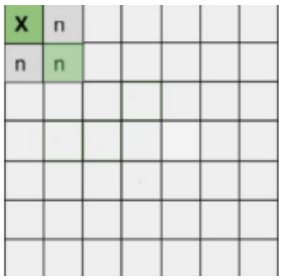


Illustration 3: The cell X has three neighbors, two dead and one living.

5. To calculate the next generation of cells, the method is required *oppdatering*. This method should do the following:

   a) Create two lists. One list should contain all dead cells that will have the status "alive", while the other should contain living cells that will be given "dead" status. Leave the lists blank for now.

   b) Then, the method will go through the grid using a nested loop. For each cell, it will check if the cell is live or dead and then calculate whether it should change status based on the number of living neighbors. Here you have to pick up all the neighbors of a cell and count the number that lives. Follow the list of rules described under "Game Rules" further up. Cells that will change status should be entered in the correct of the two lists we previously created.

   c) Only when all the cells are checked and lists are filled with cell objects happens the actual update, and objects in the two lists changes status using cell methods *settLevende or settDoed*.

   d) Finally, keep in mind to update the counter for number of generations.

   e) Expand the test program from subproblem 3 by calling the method *oppdatering* once. Does the program behave as expected? Tip: This method can be tested by filling the grid with a known pattern and seeing that it changes as expected after a generation.

6. In addition to generational numbers, we are interested in the overall cell status on the game board. You must therefore write a method f*innAntallLevende* that can calculate and return the number of living cells. You can easily do this by going through the grid and increasing a counter for every living cell you find.

Main Program
File Name: main.py

Type a main program, *main*, where the user should first be asked to enter dimensions on the game board. Then you create the board and print the "zero" generation.

Using a menu loop and input, the user can then choose to enter an empty line to move on to the next step, or enter the letter "q" to quit the program. Each time the user states that they want to continue, call the *oppdatering-method* and then reprint the board together with a line describing which generation is displayed and how many cells are currently living.

Example of printing the game board:

Tranlations of last two lines in the game board.

Generasjon:20 =
Generation: 20

Antall levende celler: 129=
Number of living cells: 129

press enter for å fortsette. Skriv inn q og trykk enter for å avslutte: =
press enter to continue. Type q and press enter to exit:

Here the word *pass* in the code is used only as a placeholder so you can test your program before you have written all the methods. It should not be included when you submit your assignment.

Appendix 1: Code skeletons

Cell

Filnavn: celle.py

```
class Celle:
# Constructor
def __init__(self):
    pass

# Change status
def settDoed(self):
    pass

def settLevende(self):
    pass

# Retrieve status
def erLevende(self):
    pass

def RetrieveStatusCharacters (self):
    pass
```

Game board

Filnavn : spillebrett.py

```
from random import randint
from celle import Celle

class game board:
def __init__(self, rader, kolonner):
pass

def tegnBoard(self):
pass

def oppdatering(self):
pass

def finnAntallLevende(self):
pass

def generer(self) :
pass

def finnNabo(self, x, y):
pass
```

Main
Filnavn: main.py

```
from Game board import game board
def main():
pass

# start main-program.
main()
```