OXFORD INTERDISCIPLINARY BIOSCIENCE
Doctoral Training Partnership

PyGoL: A Python Software Framework for Cellular Automata Modelling

Inês V. Barreiros[1]

[1]DPhil in Interdisciplinary Bioscience, Mathematical, Physical and Life Sciences Division, University of Oxford, United Kingdom.
Email address: ines.barreiros@dtp.ox.ac.uk

I have developed PyGoL, a python framework for a Game of Life-like cellular automaton, for modelling organisms' populations in an environment with renewable but limited resources available. This framework is an easy-to-use open-source tool, user friendly and computationally efficient, without requiring installation of special python modules. It is a very versatile framework that allows the user to define a number of variables for specific population studies and can be easily adapted for other applications. The framework returns the output to the user in several formats, allowing for both fast interpretation and further analysis of results. Here I document the development and workflow of PyGoL, and give insights into the further developments and applications for this framework.

Keywords: cellular automaton, game of life, python, population equilibrium.

1. Introduction

Cellular automata, plural for cellular automaton, are spatially and temporally defined discrete mathematical models extensively used as a tool to study evolution of systems[1]. The concept was firstly introduced by the mathematician Jonh von Neumann in the 1950s, originally to study machine self-replication[2].

On a typical cellular automaton, a grid-like "environment space" is defined. Within it, fixed rules are set for the initial generation at *time (t) = 0*. When a "life cycle" occurs, *t +=1*, the new generation and state of the system is created in accordance with the defined rules and given variables, updating the system with the new state of variables[1].

Stephen Wolfram significantly expanded cellular automata by studying the potential of its application to a number of fields[3]. Nowadays, cellular automata are used in a range of different disciplines, including mathematics, physics, theoretical biology, and computer sciences. By defining just a set of simple rules, in the field zoology for example, they can be used to study behaviour or predict the evolution of real systems when trying to answer a range of research questions, as predicting the patterns on sea shells or the geometrical growth of biological organisms[4].

In the 70s, Jonh Horton Conway presented a simplified version of cellular automata modelling as "the Game of Life", a game in which cells placed on a grid evolve forming different patterns according the initial conditions defined by the player. Because of its simplicity and versatility, the Game of Life had its applications expanded to a range of different research fields[5].

Although a number of cellular automata mathematical models has been described (a number of examples here: www.uncomp.uwe.ac.uk/genaro/Cellular_Automata_Repository/Software.html), by performing an online literature search, I was not able to find a framework that would enable me to run a cellular automaton in which conditions could be changed without having to install several software packages or modules. This is a problem as, often, researchers use computers owned by the university to run their simulations and analyses and do not have administrator privileges, or when there are incompatibilities issues when trying to install the required software. Here, I describe the development and workflow of PyGoL, a cellular automaton, game of life-like, framework that overcomes this problem by using built-in python libraries only. Given a set of initial variables, PyGoL allows for the study of self-replicating organisms evolution across generations in the presence of renewable but limited resources.

2. Methods

In this section I briefly described the main functions of PyGoL. A general overview of this framework workflow is presented in Figure 1. After importing the necessary built-in functions, PyGoL introduces itself to the user and explains the rules of the game:

```
The rules of this Game of Life are:
    (1) Animals need to eat at least a minimum amount of food to survive;
    (2) An animal grazes in every place one unit away from the position in which
it was created;
    (3) An animal eating food reduces the amount of food in those field units;
    (4) Food regrows at a constant rate;
    (5) If there is space, an animal will breed;
    (6) Animals can breed up, down, left or right.
```

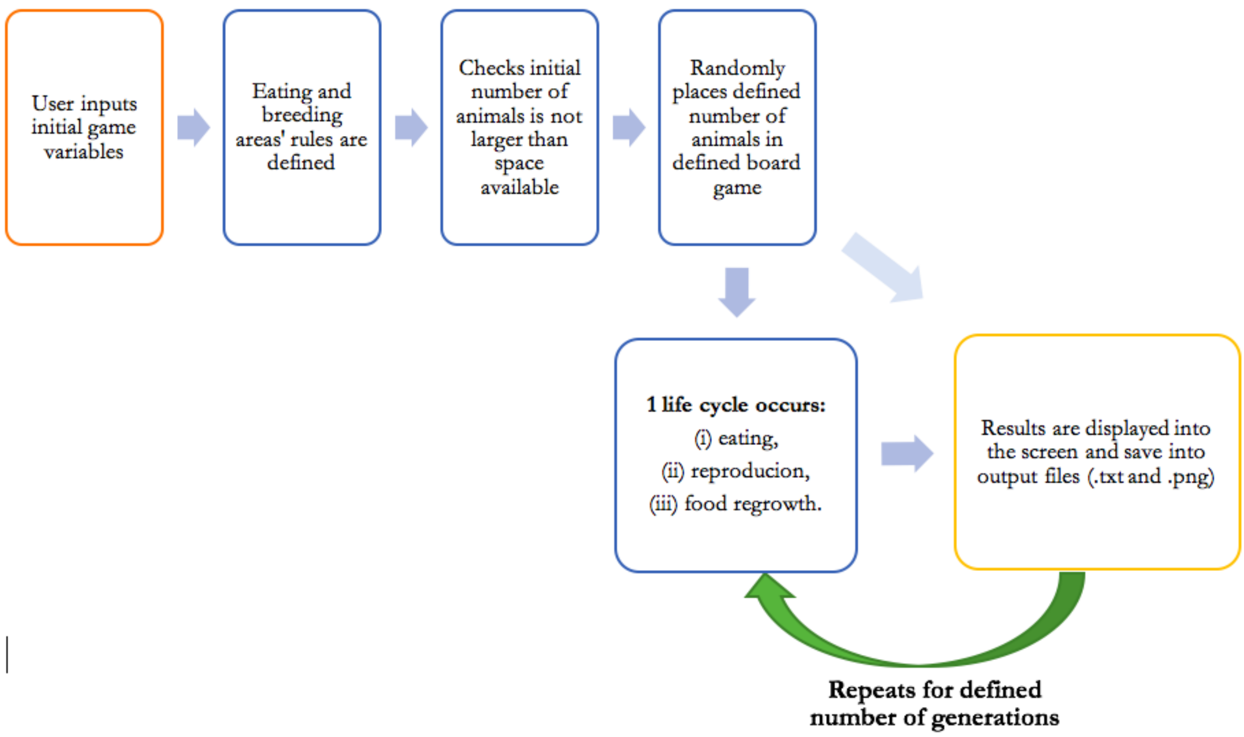


Figure 1 | General overview of PyGoL framework workflow.
At the beginning of the game, the user gives input defining the game variables: initial/maximum amount of food, initial number of animals, food regrowth rate, size of the board and number of generations. Then the rules for the eating and breading areas are defined in the program. After checking that the initial number of animals at generation 0 is no larger than the space available, it randomly places the initial number of animals in a board game of dimensions defined by the user and with maximum food available. A cycle life occurs: animals eat and reproduce, and food regrowth. This last step is repeated for the number of generations defined by the user, with each new state of the board game for food and animal locations being displayed into .txt and .png output files in a new folder with the name of the simulation.

2.1. Input from the user

Making use of the `raw_input` function, the program asks the user to name the simulation and give input values for a set of variables. The user is prompted to define: initial/maximum amount of food, initial number of animals, food regrowth rate, size of the board game (width and height) and number of generations. An error message is displayed into the screen when an input different from an integer is given. A summary of the input variables is stored in the output TXT file.

2.2. Grazing and breeding areas

Taking into account the game grazing and breeding rules, two functions, `food_neighbours` and `breeding_neighbours`, define where an animal will eat and bread considering its location in the board game. Animals cannot eat or bread outside the board game, therefore these functions define what happens also when animals are in corners or sides of the board.

2.3. Initialising the game

After checking the initial number of animals is not higher than the dimensions of the board game, the function initialise_animals places them randomly into a board with food on its maximum value. In the text format, animals are represented as 1 and absence of animals as 0.

2.4. Eating

Function `eating` will use the rules defined by `food_neighbours` to reduce the amount of food where the animals is eating (1 unit per cycle) and update the board food distribution. This function will also check if each animal has enough food around him to survive the cycle. If the animal does not eat, it will die (1 becomes 0).

2.5. Breeding

After eating, the function `breeding` will place new animals in the board game according the rules defined by `breeding_neighbours`. As two animals cannot be at the same time in a space unit of the board game, if two animals try to breed in the same location, only one of them will succeed.

2.6. Food Growth

Food growths at a rate defined by the user at the beginning of the game. The function `food_growth` will make this happen by adding more food to the board. However, food values will never exceed the starting/maximum value.

2.7. Generations (life cycles)

For each generation an animal will eat, breed and food will regrowth. Function `generation` will call the eating, breeding and food growth functions returning updated states of the board game for food and animals. The output of the game at each stage will be created with a `for` loop that will run through the number of generations defined by the user.

2.8. Displaying output

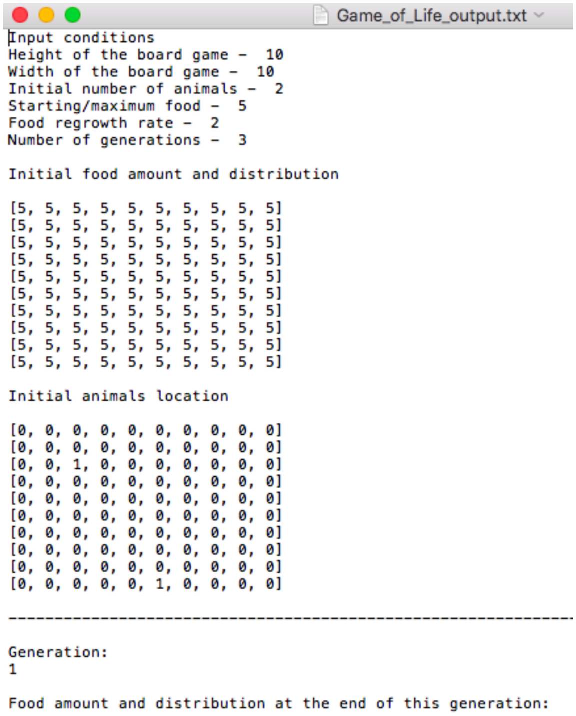
A number of functions will allow the user to observe the evolution of the game in different formats: in text and in image format:

- disp_board – for easier visualisation, this function will display the output into different text lines, rather than in a long string of numbers;
- board_image – an image of the board game combining food amounts and animals' location will be displayed for each generation.

A `for` loop will call these functions for each generation of the game and display the output in the different formats.

3. Results

The program will tell the user when it finishes running and where he can find the results. The results are saved into a text (TXT) format (Figure 2), in a file named 'Game_of_life_output.txt' that will be stored in a new folder with the name of the simulation created in the working directory. This text file contains information about the initial sate of the game and how it develops every further generation. Additionally, at the end of the file it is indicated how long the program took to run. This type of output allows the user to perform statistical studies on the results obtained at a later stage.


Game_of_Life_output.txt

```
Input conditions
Height of the board game -  10
Width of the board game -  10
Initial number of animals -  2
Starting/maximum food -  5
Food regrowth rate -  2
Number of generations -  3

Initial food amount and distribution

[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]
[5, 5, 5, 5, 5, 5, 5, 5, 5, 5]

Initial animals location

[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 1, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 0, 0, 0, 0, 0]
[0, 0, 0, 0, 0, 1, 0, 0, 0, 0]

-------------------------------------------------------------

Generation:
1

Food amount and distribution at the end of this generation:
```

Figure 2 | Example TXT format PyGoL results (partial screenshot).
PyGoL outputs results into a TXT file, storing the input variables and state of the food and animals' boards at each generation.

For a more intuitive visualisation of the results, these are also saved in individual image (PNG) files which represent the state of the system at each generation (Figure 3). Individuals are represented with black dots and amount of food in a range of colour from green (maximum) to red (0).

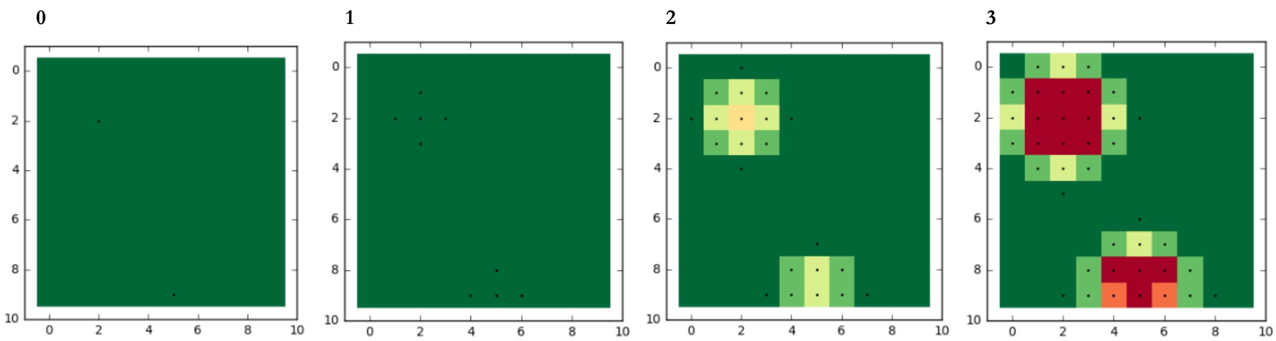


Figure 3 | Example results in PNG format of a population over 3 generations.
Representation of the evolution of a population with 2 initial animals (**black dots**) randomly placed in a 10 by 10 board game. Food has initial/maximum value of 5 (**green**) and regrowth rate of 2 per generation. Over the course of 3 further generations, it is possible to observe the reproduction of the animals (appearance of new black dots) and the food consumption, with the amount of food decreasing in the more populated areas, in this example going from **green** (maximum, 5), to **light green**, an **even lighter green**, **yellow**, **orange** and **red** (0).

As previously mentioned, all the output files will be saved in a new folder in the working directory, with the name of the simulation given by the user at the beginning of the game.

4. Implementation

I choose the python programming language[6] for the development of PyGoL. Python is a relatively simple language and therefore will facilitate the adjustment of the source code to accommodate new variables when trying to answer specific research questions, even by researchers with little programming knowledge.

5. Software availability

PyGoL is available as an open-source code repository at www.github.com/inesbarreiros/PyGoL and can be found in the Appendix of this report. PyGoL will run in Python 2.7 and can run in other versions with minor adaptations. When converting from Python 2.7 to Python 3 this can be done automatically with the python library `lib2to3` (more information at www.docs.python.org/3.0/library/2to3.html). Additionally, PyGoL framework can be converted to the user's language of choice by using conversion software as Simplifier Wraper and Interface Generator (available at www.swig.org).
To run PyGoL the user will just need to type `python PyGoL_v1.0.py` on a computer terminal of a machine with an appropriate Python version, when inside the directory where the source code is saved.

6. Summary

Cellular automata are discrete mathematical models extensively used in different research fields[5]. This report presents the development of PyGoL, a python framework for a cellular automaton, which allows researchers to study the dynamics of populations and their evolution across generations. This game of life-like framework allows the user to define a set of variables at the beginning of the game, which include initial/maximum food available, food regrowth rate, initial number of animals, dimensions of the board and number of generations. Additionally, the source code can be easily adapted to include sets of different variables or change the game rules, in order to study particular research questions.

In just a few of seconds or minutes (output of 5 generations, 10 by 10 board, run in 1.9 seconds; output of 20 generations, 100 by 100 board, in 43 seconds), PyGoL outputs the results into image and text files, which allows the user to both take quick, intuitive conclusions about the system evolution and to perform statistically analysis at a later stage. Therefore, this framework has great potential for research use.

7. Further Development

In the future, PyGoL can be expanded by adding new features and different output formats. A number of input variables from the user could be added to the program, thus making results more accurate when trying to simulate real life environments. For example, animals could die after a certain age (number of generations) or probabilistically by natural or external causes. Additionally, a different breeding method in which reproduction would only occur in the presence of at least 2 individual of different sex could be implemented. This would require a proximity between individuals of opposite sexes. To be more realistic, the offspring number could vary from a range of possibilities and animals would only be able to reproduce during a certain 'fertile period'. In other words, animals would have to reach a certain age to start reproduce and could not reproduce when too old. It is also possible to add functions that could answer of the probability of certain events occurring at a certain location and generation, given initial conditions.

In terms of output files, with the installation of specific libraries, the program could allow for a range of different formats (e.g. GIF or video format), which would give a different insight of the results to the user.

References

1. Wolfram, S. Statistical mechanics of cellular automata. *Rev. Mod. Phys.* **55,** 601–644 (1983).
2. Edmundson, H. P. Theory of self-reproducing automata. *Information Storage and Retrieval* **5,** 151 (1969).
3. Wolfram, S. A new kinf of Science. *Wolfram Media* (2002).
4. Conway, J. The game of life. *Sci. Am.* **303,** 43–44 (1970).
5. Ilachinski, A. *Cellular Automata: A Discrete Universe*. *Cellular Automata: A Discrete Universe* (2001).
6. van Rossum, G., Python reference Manual. *Technical Report CWI (Centre for Mathematics and Computer Science) Amsterdam, The Netherlands, The Netherlands* (1995).

Appendix

```python
#! /usr/bin/env python

#Program written on November 2016, Ines Barreiros - Interdisciplinary Bioscience DTP

#This is a  Game of Life-like Cellular Automaton, it provides a way to model populations by modification of input variables.
#You can interact with this Cellular Automaton by creating an initial configuration and observing how it evolves. You will also be able to define a number of initial variables.

import matplotlib.pyplot as plt
import random, copy, os, timeit #Import all the necessary modules.

print 'This program is a a Cellular Auomaton aka Game of Life, it will allow you to observe the evolution of state of the animal population over time and according the parameters inital set. \n The rules of this Game of Life are: \n    (1) Animals need to eat at least a minimum amount of food to survive; \n    (2) An animal grazes in every place one unit away from the position in which it was created; \n    (3) An animal eating food reduces the amount of food in those field units; \n    (4) Food regrows at a constant rate; \n (5) If there is space, an animal will breed;\n    (6) Animals can breed up, down, left or right. \n '

print 'In this game you will be able to define a number of variables: height and width - dimensions of the board/field of the game; number_of_animals - inital number of animals in the field;  start food - inital food available, which is also the maximum food; regrowth rate - food regrowth rate per cycle of the game; numer of loops - number of generations you want to display.\n '


sim_name = str(raw_input ('Please provide a name for your simulation: \n '))

script_dir = os.getcwd()
results_dir = os.path.join(script_dir, sim_name)

if not os.path.isdir(results_dir):
    os.makedirs(results_dir)

output_file = open(os.path.join(results_dir,'Game_of_Life_output.txt'), 'w') #Create file in which the output of the game will be written.


#Ask for input from the user for variables:
height = int(raw_input ('Please define the size of the board/field of the game. What will be the height? \n '))
if type(height) != int:
    print 'That is not a number! Please enter a number for the height of the board game.'

width = int(raw_input ('Please define the size of the board/field of the game. What will be the width? \n '))
if type(width) != int:
    print 'That is not a number! Please enter a number for the width of the board game.'

number_of_animals = int(raw_input ('How many animals there will in the board at the beggining of the game? \n '))
if type(number_of_animals) != int:
    print 'That is not a number! Please enter a number for the initial number of animals in the board game.'

start_food = int(raw_input ('What will be the maximum amount of food available per space? This will also be the amount of food initially available. \n '))
if type(start_food) != int:
    print 'That is not a number! Please enter a number for the maximum amount of food available per space.'

```

```
regrowth_rate = int(raw_input ('What will be the food regrowth rate per cycle of the game? \n '))
if type(regrowth_rate) != int:
    print 'That is not a number! Please enter a number for the regrowth rate.'

num_loops = int(raw_input ('How many further generations would you like to display? \n '))

if type(num_loops) != int:
    print 'That is not a number! Please enter a number for the regrowth rate.'

#Print summary of initial conditions to output file
print >> output_file,'Input conditions \n', 'Height of the board game - ', height, '\n', 'Width of the board game - ', width, '\n', 'Initial number of animals - ', number_of_animals, '\n', 'Starting/maximum food - ', start_food, '\n', 'Food regrowth rate - ', regrowth_rate, '\n', 'Number of generations - ', num_loops, '\n'

def food_neighbours(col,row,height,width): #An animal grazes in every place one unit away from the position in which it was created. This function defines the coordinates for eating taking into account the position of the animal, also considering where animals are in corners or sides of the board.
    list_food_neighbours = [[col,row]]
    if (col == 0) & (row == 0): #Corner of the board
        list_food_neighbours.append([col+1,row])
        list_food_neighbours.append([col+1,row+1])
        list_food_neighbours.append([col,row+1])
    elif (col == 0) & (row == (height-1)): #Another corner of the board
        list_food_neighbours.append([col+1,row])
        list_food_neighbours.append([col+1,row-1])
        list_food_neighbours.append([col,row-1])
    elif (col == (width-1)) & (row == 0): #Another corner of the board
        list_food_neighbours.append([col-1,row])
        list_food_neighbours.append([col-1,row+1])
        list_food_neighbours.append([col,row+1])
    elif (col == (width-1)) & (row == (height-1)): #Another corner of the board
        list_food_neighbours.append([col-1,row])
        list_food_neighbours.append([col-1,row-1])
        list_food_neighbours.append([col,row-1])
    elif (col == 0): #Left column of the board
        list_food_neighbours.append([col,row+1])
        list_food_neighbours.append([col,row-1])
        list_food_neighbours.append([col+1,row-1])
        list_food_neighbours.append([col+1,row+1])
        list_food_neighbours.append([col+1,row])
    elif (col) == ((width-1)):  #Right column of the board
        list_food_neighbours.append([col,row+1])
        list_food_neighbours.append([col,row-1])
        list_food_neighbours.append([col-1,row-1])
        list_food_neighbours.append([col-1,row+1])
        list_food_neighbours.append([col-1,row])
    elif (row == 0): #Top column of the board
        list_food_neighbours.append([col+1,row])
        list_food_neighbours.append([col-1,row])
        list_food_neighbours.append([col-1,row+1])
        list_food_neighbours.append([col+1,row+1])
        list_food_neighbours.append([col,row+1])
    elif (row == (height-1)): #Bottom row of the board
        list_food_neighbours.append([col+1,row])
        list_food_neighbours.append([col-1,row])
        list_food_neighbours.append([col-1,row-1])
        list_food_neighbours.append([col+1,row-1])
        list_food_neighbours.append([col,row-1])
    else: #Any other place on the board
        list_food_neighbours.append([col,row+1])
        list_food_neighbours.append([col,row-1])
        list_food_neighbours.append([col-1,row])
        list_food_neighbours.append([col-1,row-1])
```

```
            list_food_neighbours.append([col-1,row+1])
            list_food_neighbours.append([col+1,row])
            list_food_neighbours.append([col+1,row-1])
            list_food_neighbours.append([col+1,row+1])

        return list_food_neighbours

    def breeding_neighbours(col,row,height,width): # If there is space, an animal will breed and animals can breed up, down, left or right. This function defines the coordinates for breeding taking into account the position of the animal, also considering where animals are in corners or sides of the board.
        list_breeding_neighbours = []
        if (col == 0) & (row == 0):  #Corner of the board
            list_breeding_neighbours.append([col+1,row])
            list_breeding_neighbours.append([col,row+1])
        elif (col == 0) & (row == (height-1)): #Another corner of the board
            list_breeding_neighbours.append([col+1,row])
            list_breeding_neighbours.append([col,row-1])
        elif (col == (width-1)) & (row == 0): #Another corner of the board
            list_breeding_neighbours.append([col-1,row])
            list_breeding_neighbours.append([col,row+1])
        elif (col == (width-1)) & (row == (height-1)): #Another corner of the board
            list_breeding_neighbours.append([col-1,row])
            list_breeding_neighbours.append([col,row-1])
        elif (col == 0): #Left column of the board
            list_breeding_neighbours.append([col,row+1])
            list_breeding_neighbours.append([col,row-1])
            list_breeding_neighbours.append([col+1,row])
        elif (col == (width-1)): #Right column of the board
            list_breeding_neighbours.append([col,row+1])
            list_breeding_neighbours.append([col,row-1])
            list_breeding_neighbours.append([col-1,row])
        elif (row == 0): #Top column of the board
            list_breeding_neighbours.append([col+1,row])
            list_breeding_neighbours.append([col-1,row])
            list_breeding_neighbours.append([col,row+1])
        elif (row == (height-1)): #Bottom row of the board
            list_breeding_neighbours.append([col+1,row])
            list_breeding_neighbours.append([col-1,row])
            list_breeding_neighbours.append([col,row-1])
        else: #Any other place on the board
            list_breeding_neighbours.append([col,row+1])
            list_breeding_neighbours.append([col,row-1])
            list_breeding_neighbours.append([col-1,row])
            list_breeding_neighbours.append([col+1,row])

        return list_breeding_neighbours

    def initialise_animals(height,width,number_of_animals): #Confirms the initial number of animals is not larger than space available & randomly places the initial number of animals in the game board.

        if number_of_animals > int(width*height):
            print'Error: number of starting animals is larger than space available.'
            return


        temp_board_animals = [[0 for col in range(width)] for row in range(height)]

        count = 1

        while count <= number_of_animals:
            col = random.randrange(width)
            row = random.randrange(height)

            if temp_board_animals[row][col] == 0:
```

```
                temp_board_animals[row][col] = 1
                count += 1

        return temp_board_animals #Temporary position of animals in the board for initial cycle is defined.


    def eating(board_animals,board_food,master_list_food_neighbours): #Makes animals eat on the adequate location (one unit away from their location, in any direction).
        temp_board_animals = copy.deepcopy(board_animals)
        temp_board_food = copy.deepcopy(board_food)

        for col in range(width):
            for row in range(height):

                if board_animals[row][col] == 1:
                    eating_areas = master_list_food_neighbours[row][col]

                    eaten_check = 0 #Before eating, the amount of eating for the current cycle is 0.

                    for item in eating_areas: #Where each animals ie eating whithin the eating areas the animal can eat.
                        if board_food[item[1]][item[0]] > 0: #item 1 is row, item 0 is column; here the function will go through all the items of the list of neighbours that we previously created
                            temp_board_food[item[1]][item[0]] -=1
                            eaten_check = 1 #Means the animal did ate.

                    if eaten_check == 0: #If the animal still did not eat after the eating phase (because there was no food available), then the animal will die.
                        temp_board_animals[row][col]= 0

        return temp_board_animals,temp_board_food #Temporary position of animals in the board and board of food with new amounts.

    def breeding(board_animals,master_list_breeding_neighbours): #Breeding will happen in the appropriate locations. If 2 animals breeding areas overlap only one of them will bread on that area, therefore number of animals per space unit will not exceed 1 animal.
        temp_board_animals = copy.deepcopy(board_animals)

        for col in range(width):
            for row in range(height):

                if board_animals[row][col] == 1:
                    breeding_areas = master_list_breeding_neighbours[row][col]

                    for breeding_location in breeding_areas:
                        if board_animals[breeding_location[1]][breeding_location[0]] == 0:
                            temp_board_animals[breeding_location[1]][breeding_location[0]]  = 1
        return temp_board_animals #New temporary position of animals after the breeding.


    def food_growth(board_food,start_food,regrowth_rate): #With time, food will regrowth at the define rate...
        temp_board_food = copy.deepcopy(board_food)

        for col in range(width):
            for row in range(height):
                temp_board_food[row][col] += regrowth_rate

                if temp_board_food[row][col] > start_food:
```

```
                temp_board_food[row][col] = start_food #...however food amount will never be higher than the initially defined maximum.
            if temp_board_food[row][col] < 0:
                temp_board_food[row][col] = 0
    return temp_board_food #New amount of food in the board, considering the food eaten at the beggining of the cycle and the amount of food that regrowth at the end of the cycle.

def generation(board_animals,board_food): #For each generation, the functions above will be called - the animal will eat (and die if it doesn't eat), breed, and food will regrowth.
    board_animals,board_food = eating(board_animals,board_food,master_list_food_neighbours)
    board_food = food_growth(board_food,start_food,regrowth_rate)
    board_animals = breeding(board_animals,master_list_breeding_neighbours)

    return board_animals,board_food #Location of animals and food at the end of generation cycle.

def disp_board(board): #Function that will be used to display output in different lines of text.
    str_boad = ''
    for line in board:
        str_boad += '\n'
        str_boad += str(line)
    return str_boad


def board_image(board_food,board_animals):  #Function that makes an image of the state of the board at each generation.
    fig = plt.figure()
    plt.imshow(board_food,cmap=plt.get_cmap('RdYlGn'), clim=(0, start_food), interpolation='nearest') # a colormap from red (0) to green (maximum food) is used to display the amount of food.
    for col in range(width):
        for row in range(height):
            if board_animals[row][col]:
                plt.scatter(x=[col], y=[row], c='k', s=1) #a scatter plot of black dots is used to display the location of living animals
    return fig


start = timeit.default_timer() #Initiates stopwatch for time the program takes to run.

#Calling all the functions necessary to run the generations with all steps and displaying output:
board_food = [[start_food for col in range(width)] for row in range(height)]
board_animals = initialise_animals(height,width,number_of_animals)

master_list_food_neighbours = [[food_neighbours(col,row,height,width) for col in range(width)] for row in range(height)]
master_list_breeding_neighbours = [[breeding_neighbours(col,row,height,width) for col in range(width)] for row in range(height)]

print '\nSimulation running...'

print >> output_file, 'Initial food amount and distribution \n', disp_board(board_food), '\n \n', 'Initial animals location', '\n', disp_board(board_animals)

fig = board_image(board_food,board_animals)
plt.savefig(os.path.join(results_dir,('generation_0_initial_state.png')))

#Will loop through the life process during the number of generations the user wants to display
for generation_number in range(1,num_loops):
```

```

            board_animals,board_food = generation(board_animals,board_food)
            print >> output_file, '\n', '--------------------------------------------------------------------------------'
            print >> output_file, '\n','Generation:'
            print >> output_file, generation_number
            print >> output_file, '\n','Food amount and distribution at the end of this generation:'
            print >> output_file, disp_board(board_food)
            print >> output_file, '\n','Animals location at the end of this generation:'
            print >> output_file, disp_board(board_animals)
            fig = board_image(board_food,board_animals)
            plt.savefig(os.path.join(results_dir,('generation_' + str(generation_number) + '.png'))) #saving of the state of each generation as a PNG file
            plt.close(fig) #Close figures to avoid unnecessary memory usage.


        generation_number = num_loops
        board_animals,board_food = generation(board_animals,board_food)

        print >> output_file, '\n', '--------------------------------------------------------------------------------'
        print >> output_file, '\n','Final generation:'
        print >> output_file, generation_number
        print >> output_file, '\n','Food amount and distribution at final generation:'
        print >> output_file, disp_board(board_food)
        print >> output_file, '\n','Animals location at the final generation:'
        print >> output_file, disp_board(board_animals)
        fig = board_image(board_food,board_animals)
        plt.savefig(os.path.join(results_dir,('generation_' + str(generation_number) + '_final_state.png')))

        print '\nSimulation complete.'
        print '\nYou can now find the evolution your population over', generation_number, ' generations in a folder with the name of your simulation in TXT and PNG formats.'

        stop = timeit.default_timer() #Stops stopwatch for time the program takes to run.

        print  >> output_file, '\n', '\n________________________________________________________________________________\n','Time the Game of Life took to run: ', stop - start, ' seconds' #Prints how long the program took to run.

        #End of the program, close all the files opened.
        output_file.close()
```