Programming Project: Foxes and Rabbits Simulation
Phase II

By: Merete H., Andreas L.N., Christoffer M.K.
Data Science
Introduction to programming (DS830)
University of Southern Denmark
2021

Table of Contents

Introduction to Phase II of our Project: Simulation

In this part of the project, we had to implement the middle part of the program - Simulation. Simulation has to initialize the data structures to represent the simulated world according to given parameters. The simulation updates in ticks, where each tick represents the grass growing, animals moving, aging and dying etc.
Part of the program tracks data and statistics for the reporting stage of the program (phase III), like the size of the population over the execution of the simulation, age at the time of death for each animal, total deaths etc.

Collaboration

Since this phase concerned implementing only one module, and not several like in phase I, it affected the collaboration of this phase. It wasn't really possible to assign parts of the Simulation to each of us, since it had to be coded as several functions that were all dependent on each other. Because of this, the collaboration developed into more of a "peer-to-peer"-system when creating the code.

This model of collaboration worked relatively well, however, it did force us to focus even more on code readability and documentation, since we had to explain and document the changes so every group member knew what the latest version of our code did. We, therefore, kept a copy of all previous versions, so we easily could compare the latest version to the previous version.

Bottom up Design

We started by using the hand-out for phase II, that specified which tasks Simulation was supposed to handle. The various tasks Simulation should handle were deconstructed into sub-functions, which were later integrated into a bigger solution. While some of the sub-functions were used in this module, some of them turned out to be unnecessary, like *get_spec_field*, which would return a specific field from the world.

In short, our design approach resembled a bottom-up approach, where we designed our module around its functionalities. This made documenting the code easier, and it made it easy to quickly implement changes in our functions. However, it also made it easy to lose sight of the overall design goal, which sometimes resulted in redundant code.

Structure of Module

In phase II the module Simulation was added to the project. This module runs a simulation and collects statistics of the overall simulation.

Simulation module:

- **create_world**:
  A function that creates an empty world for entities. It returns a list with nested lists corresponding to a grid with the dimensions of the world (a matrix).
- **fill_world:**
  A function that fills an empty world with patches
- **get_rand_field:**
  A function that returns a random position from the provided world. It picks random values using the random module
- **populate_world:**
  A function that populates a world of patches with animals.
  _create_animals: A helper function for creating new animals
- **_nearest_coords:**
  A private function that gets neighbor fields according to specified coordinates. It gets all neighbors from the world according to north south and west east coordinates. The neighboring fields can be chosen according to rook, bishop or queen movement.
- **get_near_by_fields:**
  A function that gets nearby fields of an animal according to specified neighbors. It returns a list of fields from the given world.
- **reproduce_animal:**
  A function that tests whether or not the animal can reproduce according to the geographical and internal rules for reproduction.
- **move_animal:**
  A function that moves an animal to a random and valid empty nearby field.
- **update_entities:**
  A function that updates the world, its entities, and collects statistics.
- **_collect_stats:**
  A private function that collects and updates the statistics of the simulation.
- **run:**
  A function that runs the simulation according to the specified parameters.

World Representation

An important decision when creating the module was choosing the primary representation of our world. We could either represent the world as a flat list of patches or we could use a matrix, which would be a list containing nested lists of patches. Both representations would eventually be needed since the subclasses of *Batch* (*ColourGraphics* and *GrayscaleGraphics*) from the module *Visualiser* both take a flat list of patches as an argument. However, the class *SimulationStats* establishes a contract, where it expects that the attribute *kills_per_patch* is a matrix containing patches.

Even though both representations were eventually needed, we had to decide which representation our functions expected, in order to establish a coherent contract throughout the module.
We decided to use a matrix as our primary representation of the world. This was done since we deemed it easier to convert a list of nested lists to a flat list for the visualiser, rather than converting a flat list to a list of nested lists for the SimulationStats.

We also concluded that it seemed more intuitive and made converting West-East and North-South coordinates to indices simpler. We did this conversion by making the outer list our North-South axis with index 0 being the furthest point north. Our West-East axis is represented by the index of the inner list with 0 being the furthest point West:

3x3 world
North
↑
West← →East

Index: 0,0 Coords: 1,1	Index: 0,1 Coords: 1,2	Index: 0,2 Coords: 1,3
Index: 1,0 Coords: 2,1	Index: 1,1 Coords: 2,2	Index: 1,2 Coords: 2,3
Index: 2,0 Coords: 3,1	Index: 2,1 Coords: 3,2	Index: 2,2 Coords: 3,3

West← →East
↓
South

With this representation, we can easily convert the indices of our world to a North-South and West-East position by simply taking the indices of the outer and inner list and adding 1 to them so they are transformed to 1-based-indexing. This also works the other way around, since we can easily convert North-South and West-East to indices in our world by subtracting each coordinate by 1.

Neighbours

How an animal moves and how it reproduces is closely tied to how neighbors are defined. In our current implementation an animal can reproduce with animals of the same species in a neighboring field. For reproduction, these fields are always defined by queen-neighbors. For movement, valid neighbors are specified by the user. The user can choose between the following neighbor types:

Queen neighbors

In this definition all fields, that touches the current field are defined as valid neighbors:

3x3 world
North
↑
West←—→East

Valid	**Valid**	**Valid**
Valid	Animal	**Valid**
Valid	**Valid**	**Valid**

West←—→East
↓
South

Rook neighbors

In this definition the fields, which are connected by the diagonal, are discounted as neighbors:

3x3 world
North
↑
West ←—→East

Invalid	**Valid**	**Invalid**
Valid	Animal	**Valid**
Invalid	**Valid**	**Invalid**

West ←—→East
↓
South

Bishop neighbors

In this definition the fields, which are not connected by the diagonal, are discounted as neighbors:

3x3 world
North
↑
West ←—→East

Valid	**Invalid**	**Valid**
Invalid	Animal	**Invalid**
Valid	**Invalid**	**Valid**

West ←—→East
↓
South

Neighbors and World Shape

How neighbors are defined, when an animal is positioned on an edge of the world, is dependent on which world shape the user has chosen.

Island Shape

If the world is an island, the animal will not be permitted to move beyond the edge of the world:

Queen Movement

4x4 world

North

↑

West ⟵⟶East

Animal	**Valid**	**Invalid**	**Invalid**
Valid	**Valid**	**Invalid**	**Invalid**
Invalid	**Invalid**	**Invalid**	**Invalid**
Invalid	**Invalid**	**Invalid**	**Invalid**

West ⟵⟶East

↓

South

Rook Movement

4x4 world

North

↑

West ⟵⟶East

Animal	**Valid**	**Invalid**	**Invalid**
Valid	**Invalid**	**Invalid**	**Invalid**
Invalid	**Invalid**	**Invalid**	**Invalid**
Invalid	**Invalid**	**Invalid**	**Invalid**

West ⟵⟶East

↓

South

Bishop Movement

4x4 world

North

↑

West ⟵⟶East

Animal	**Invalid**	**Invalid**	**Invalid**
Invalid	**Valid**	**Invalid**	**Invalid**
Invalid	**Invalid**	**Invalid**	**Invalid**
Invalid	**Invalid**	**Invalid**	**Invalid**

West ⟵⟶East

↓

South

Toroid Shape

If the world is shaped like a toroid the animal can move to the opposite side of the world when encountering the edge:

Queen Movement

4x4 world

North

↑

West ⟵⟶East

Animal	**Valid**	**Invalid**	**Valid**
Valid	**Valid**	**Invalid**	**Valid**
Invalid	**Invalid**	**Invalid**	**Invalid**
Valid	**Valid**	**Invalid**	**Valid**

West ⟵⟶East

↓

South

Rook Movement

4x4 world

North

↑

West ⟵⟶East

Animal	**Valid**	**Invalid**	**Valid**
Valid	**Invalid**	**Invalid**	**Invalid**
Invalid	**Invalid**	**Invalid**	**Invalid**
Valid	**Invalid**	**Invalid**	**Invalid**

West ⟵⟶East

↓

South

Bishop Movement

4x4 world

North

↑

West ⟵⟶East

Animal	**Invalid**	**Invalid**	**Invalid**
Invalid	**Valid**	**Invalid**	**Valid**
Invalid	**Invalid**	**Invalid**	**Invalid**
Invalid	**Valid**	**Invalid**	**Valid**

West ⟵⟶East

↓

South

Movement and Reproduction

As previously mentioned, an animal can reproduce with animals from queen-neighboring fields. We originally let reproduction follow movement (ie. if animals moved according to rook neighbors, it could exclusively reproduce with animals from rook-neighboring fields). However, we decided to exclusively use queen movement for reproduction, because it gives our animals a larger chance of reproducing. Since there will be more available fields to choose a mate from and more empty fields for the newborn animal to spawn.

When reproducing we create a list of neighboring fields (queen definition) and pass it as an argument to the function *reproduce_animal*. This function validates if there are patches with animals of the same species and without predators. It also makes sure that there are empty fields where the new animal can spawn. This function returns the newborn animal or None and a boolean value indicating if the reproduction was successful.

If the function *reproduce_animal* returns false and None, the animal moves according to the user specified movement type. This is done with the function *move_animal*, which takes a list of neighboring fields and validates if there are available fields without animals of the same species. If the animal can move, it will be moved to a random field from the list of available neighboring fields.

Functionality

Initialisation

The initialisation takes place in the function *run*. The initialisation gets its parameters from the choices the user made in the configuration menus from phase I.
First, the empty world is created from the function *create_world*. The empty world then uses the *fill_world* function to place patches (from the class Patch) corresponding to a set of coordinates.

Afterwards, the world is populated by the function *populate_world*. Each nested list in the world gets "populated" when a new animal is created. This can only happen, if an animal of the same species and the same coordinates doesn't already exist. To check this, the function iterates through a list of animal coordinates checking for duplicates.

In the next part, the world is passed on to Visualiser. Since Visualiser only takes a flat list as an argument, the world is converted to a flat list. At last, the simulation is started by the class method *start*, where it loops through the steps of the simulation.

Update

The function *update_entities* updates each entity in the world, it loops through the world and starts out by updating each patch with a tick. It then loops through each animal on a given patch and updates the animal's tick and makes it feed. The animal then tries to reproduce by using the function *reproduce_animal*. If the reproduction is unsuccessful the animal will move to a random and valid neighbor field.
During the process mentioned above, the function continually updates four different lists:

- a list of all rabbits in the current simulation step

- a list of all foxes in the current simulation step
- a list of all rabbits born in the current simulation step
- a list of all foxes born in the current simulation step

The four lists are then used as arguments in the function *_collect_stats*, which updates the relevant statistics. We originally used a dictionary to keep track of these values, however, we decided to use four lists instead, since it made our code more readable.

Collect statistics

To collect the statistics of the simulation we created a function with the name *_collect_stats* that takes five parameters. The purpose of the function is to collect statistics for every animal entity and every simulation step to eventually calculate and return the total statistics when a user calls the *run*-function. These statistics can then be used by the modules, which handles reporting.

To begin with, the *_collect_stas* function initialized and returned the statistics in a dictionary which then later could be updated in our *update_entities* function. Although we found that this approach created a lot of redundancy in our code. With this approach we quickly saw that we were actually updating and changing the statistics three times and in three different places. Our solution to this was to move the updating and the initialization of the code that handles statistics, to the *update_entities* function.

This means that inside the *update_entities* function is now where *_collect_stats* is initialized and instead of running a *for-loop* for both animals in the *_collect_stats* function we can now just call *_collect_stats* in the *update_entities* function, giving *_collect_stats* the parameters of, what population or entity we want to collect statistics about. Both ways of doing it can be used as an approach, but since we want our code to have as little repetitiveness as possible while also being readable and organized, we thought that solving it this way would fulfill those wishes.

Changes and choices:

The Simulation module underwent several changes during the development, such as substituting and expanding on functions.
An earlier version of Simulation, included the functions *get_spec_field* and *get_near_by_fields*, where Index- and Type errors were handled by try/except statements. While these functions did, in some ways, work as intended, they were changed to optimize the code, to remove redundancies, and to make it more readable. In the final version of the module, the function *get_spec_field* was deleted, while the function *get_near_by_fields* was restructured to return a newly added function *_nearest_coords,* that returns a list of neighboring fields according to specified coordinates. *_nearest_coords* is a private function, meant to only be used with *get_near_by_fields*.

Since we managed to overlook the Results module, some time was spent developing our own code to collect and return the data from the simulation. Our own solution involved the use of dictionaries and lists to collect and return the data. Moreover, when the data from the simulation was returned, it seemed “off”. As such, some time was wasted trying to figure out what went wrong when collecting and returning the data. It turned out that the same animal was recounted, when it was moved to a new field, with a higher index than the current patch, which of course skewed the results. To avoid duplication tracking we created the four lists mentioned on page 8-9 in section *Update* which lets us keep track of duplicates of animals.

Phase II vs. Phase I

While the first phase of the project concerned handling user inputs to the menus and implementing the choices of the user, the Simulation module involved taking less user input. Because of that, the need for error handling was limited. This is reflected in the way our code is written. We generally assume that the input handled by our functions, abides by the public contract, therefore, our code should generally not raise errors when run. Therefore, a smaller occurrence of Try/Except statements appear. This is due to the principle of "Contract Programming", where preconditions are expected and strictly defined for the program. (See p. 6, "Testing pre-conditions" in our prior report: Phase I: Foxes_and_Rabbits).

For phase I we created a folder to hold all of the required modules for our program. This made it easier for the user to know which script was the main one, and which scripts were for internal use. For the sake of a more user-friendly repository we took the liberty to categorize our modules further. Now, for phase II the user can see which modules handle classes, which modules handle menus, and which modules handle the running of the simulation and reporting of results. Furthermore we added a "requirements" file so that the user can easily install external dependencies, needed to use our program.

Bug-testing

Similar to the bug-testing in phase I of the project, we tried some different approaches to make sure that our code would not fail during runtime, like:

- Tested that each entity worked as intended, animals feeding, reproducing and moving in the grid by changing the values of different parameters like Rabbit population, Fox population, World etc.
- Similar to phase I we chose to continue doing cross-platform testing: OS/Software. Test that the script works on different operating systems like Windows, OSX or Linux. Furthermore, test for functionality on different python IDE's. Fx. IDLE, Anaconda, VSCODE etc. (see p. 8 in Phase 1: Foxes_and_Rabbits)

Future Implementation and Configuration

Since this part of the program differed in its structure compared to part I of the program, it had to be solved in a different manner. In the present phase, the functions in the module are much more intertwined and dependent on each other, since several functions assist in other functions.
At this stage of the project, the module Simulation initializes the status of the data structure and its entities, simulates the model in steps, and tracks its data and statistics for phase III of the project. Similar to phase I of the project, the Simulation module works as intended, and the program remains efficient and intuitive. Simulation remains as persistent, functional, modular and readable as the code in phase I.