Programming Project: Foxes and Rabbits

Phase III

By: Merete H., Andreas L.N., Christoffer M.K.
Data Science
Introduction to programming (DS830)
University of Southern Denmark
2021/2022

Table of Contents

Introduction to Phase III of our Project: Analysis and Visualization of data **2**

Collaboration **2**

Structure of Modules **2**

Functionality: **5**

Entities 5

Collecting statistics 5

Summary 6

Plot_pop_size 7

Plot_energy 8

Plot_lifespan 9

Plot_kills 12

Changes and choices: **14**

Phase III vs. Phase II and Phase I 14

Bug-testing **14**

Implementation, Configuration and final thoughts **14**

Introduction to Phase III of our Project: Analysis and Visualization of data

In this part of the project we had to write the classes for modeling the entities of the simulation (*Animal, Fox, Rabbit, Patch*), and implement the functions that analyze and visualize the data of the simulation.

Collaboration

The functionality and structure of the modules we needed to implement, *reporting* and *entities*, made it possible to work much more independently. Due to the fact that the module *entities* consisted of 4 classes we could work on the same module simultaneously. This was done by letting each member of the group write a class independently and use mock-implementations during development. However, we did start out by fully implementing the superclass *Animal*, since it would be difficult to implement the subclasses *Fox* and *Rabbit* without it. When *entities* were completed, we developed the *reporting* module. This module consists of independent functions which visualize the results, therefore, it was even easier to collaborate, since each group member could work independently on each function, without even using mock-implementations.

Structure of Modules

In phase III the module *entities* and *reporting* was added to the project.

Entities contain the classes *Patch, Animal, Fox* and *Rabbit* that represent the entities in a simulation run.

Reporting defines functions for analyzing and reporting the results of a simulation run, like charts and summaries.

Entities module:

- **class 'Animal' contains the following methods:**
  - **age:**
    returns the age of the animal
  - **can_reproduce:**
    returns *True* if animal is alive, old enough, and has enough energy to reproduce, otherwise *False*
  - **energy:**
    returns the energy of the animal
  - **feed:**
    feeds itself using the resources at its current location.
  - **is_alive:**
    Returns *True* if the animal is alive, *False* otherwise
  - **move_to:**

If the animal is alive, it goes from its current patch to the given one.

- **patch(self):**
  Returns the position of the animal.

- **predators_in:**
  Returns *True* if the given patch contains a living predator of this animal

- **reproduce:**
  If the animal is alive, it tries to reproduce using the current patch. Returns an instance for the newborn or *None*.
- **same_species_in:**
  Returns *True* if the given patch contains an alive animal of the same species
- **tick:**
  Records the passage of time = one step in the simulation. If the animal is alive, it ages and consumes energy, if it's too old or uses all of the energy reserve it dies and is removed from its patch.

- **class 'Fox' (extends Animal), and specializes the following methods:**
  - **is_alive**:
    Checks if the energy reserve of this animal is not depleted and that its age is below the max age for foxes.
  - **feed:**
    Feeds this fox with a rabbit from its current patch, if the fox is alive, its energy reserve is not full, and in a patch with an alive rabbit. One rabbit adds *food_energy_per_unit* units of energy to the energy reserve of this fox up to the maximum level possible. Extra energy is ignored.
  - **reproduce:**
    Returns an instance of this class when successful and reduces the energy reserve of this animal by the minimum energy requirement for reproduction, multiplied by *reproduction_cost_rate*.
  - **predators_in:**
    Checks if there are predators in a given patch. Since foxes do not have predators it returns False.

- **class 'Rabbit' (extends Animal):**
  - **was_killed:**
    Checks if this rabbit was killed and returns a bool.
  - **kill:**
    Used to kill a rabbit and remove it from its current patch, if the rabbit is alive.

  This class furthermore specializes the following methods:
  - **is_alive:**
    Checks if the energy reserve of this animal is not depleted, if its age is below the age limit, and if this rabbit was not killed. Returns a bool.

    - **feed:**
      If the rabbit is alive it is fed with grass from its current patch. Each unit of grass increases the energy reserve of this rabbit by one. The amount of grass a rabbit can eat each turn is limited by its metabolism value multiplied by *feeding_metabolism_rate*, the amount of energy that can be added to its reserve, and the amount of grass available at its current patch.
    - **reproduce:**
      Returns an instance of this class with age zero located at the provided Patch, when successful and reduces the energy reserve of this animal by the minimum energy requirement for reproduction multiplied by *reproduction_cost_rate*.
    - **predators_in:**
      Checks if the given patch has an alive fox, returns a bool.

- **class 'Patch':**
    - **coordinates:**
      Returns the coordinates of the patch
    - **grass:**
      Returns the amount of grass in a patch.
    - **tick**:
      Records the passage of time = one step in the simulation.
    - **animals:**
      Returns a list of animals in the patch
    - **has_alive_fox:**
      Checks if there is an alive fox in the patch
    - **has_alive_rabbit:**
      Checks if there is an alive rabbit in the patch
    - **add:**
      Adds an animal to the patch
    - **remove:**
      Removes an animal from the patch

Reporting Module:

- **summary**:
  Prints a summary of the simulation results and basic statistics.
- **_line_break:**
  Prints a line break with the appropriate length according to the length of the headers
- **_format_cell:**
  Formats a cell value so its layout fits the table
- **_print_row**:
  Prints a row containing four cells that are formatted to fit the table. It is dependent on variables in the outer scope.
- **plot_pop_size**:
  Plots population sizes across time.

- **plot_lifespan**:
  Plots lifespans across each individual in a given population
- **plot_energy**:
  plots the total energy over time of each individual.
- **plot_kills**:
  Shows the distribution of kills in a grid

Functionality:

Entities

For the world to be able to have animals, food and patches to travel across, another module was to be created following the logic of object-oriented programming. Every instance of an animal, a patch of grass, time/ticks and any event conflicting the physical world we want to set up and simulate, holds various information about some behavior. This is due to the way they've been declared with rules and limits of action in the world they are to be put in.

Collecting statistics

To collect statistics from the simulation a reporting module needed to be implemented. Here, the user should be able to collect data and statistics from the predator-prey simulation with the possibility of having the results visualized using the third-party modules numpy and Matplotlib. Furthermore it is possible to smoothly exit and reenter the various reporting choices presented to the user, without generating or raising any errors that could crash the program. Finally, when the reporting menu is terminated, the user can run a new simulation and collect statistics again.

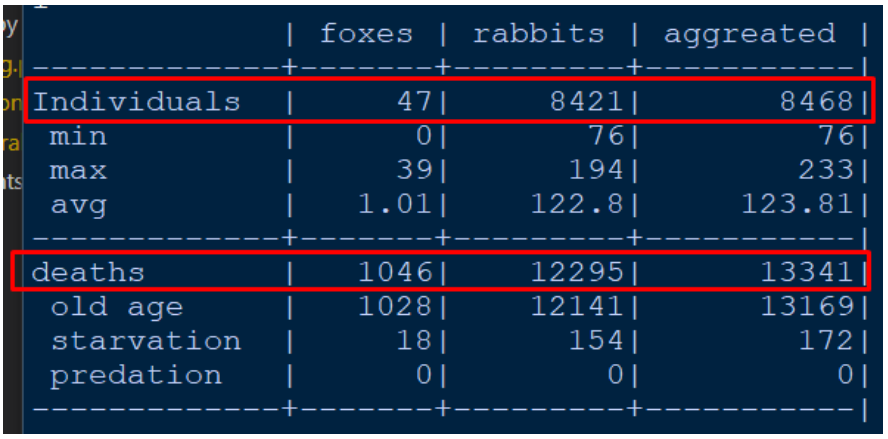
Hence we are working with creating different data points, rules of behavior and handling multiple beings of the same environment, the approach of object oriented programming has seen itself suitable even necessary for the ability to collect statistics properly. It is the combination of setting up both entities and the statistics collection as objects. If written correctly this ensures that the right data is being pulled back and forth when the program is running, minimizing the possibility of bugs and errors.

When visualizing the results it quickly became apparent that our version of *entities* wasn't implemented correctly, since it displayed vastly different results, when compared to the version provided in the previous phase.

Summary

Summary was formatted correctly, however, it was obvious that we counted dead animals multiple times compared to the provided version of the entities module:

Our version of entities

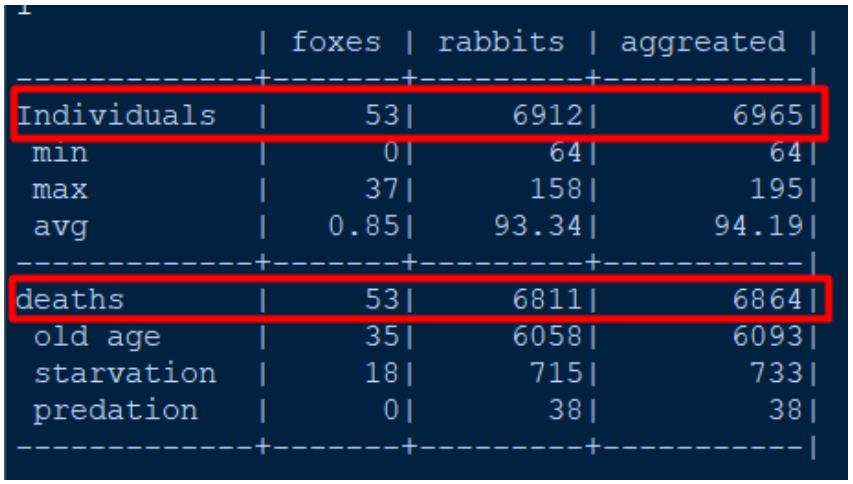


	foxes	rabbits	aggreated
Individuals	47	8421	8468
min	0	76	76
max	39	194	233
avg	1.01	122.8	123.81
deaths	1046	12295	13341
old age	1028	12141	13169
starvation	18	154	172
predation	0	0	0

Fig. 1:

Our version of entities counts more dead animals (13.341), than animals that have ever existed (8.468)

Provided version of entities



	foxes	rabbits	aggreated
Individuals	53	6912	6965
min	0	64	64
max	37	158	195
avg	0.85	93.34	94.19
deaths	53	6811	6864
old age	35	6058	6093
starvation	18	715	733
predation	0	38	38

Fig. 2:

The count seems correct with the provided version of entities. Since there are fewer dead animals (6.864) than animals, that have existed (6.965)

Even though our results are vastly different, the sensible results we get from the provided version of entities at least helped us determine that the problem was our implementation of entities, not with the other modules.

Plot_pop_size

Since it at first glance appears that we recounted dead animals, we expected that our population size would likewise return unusual results. However, to our surprise, the results of our implementation of entities and the provided version, returned similar results:

Our version of entities

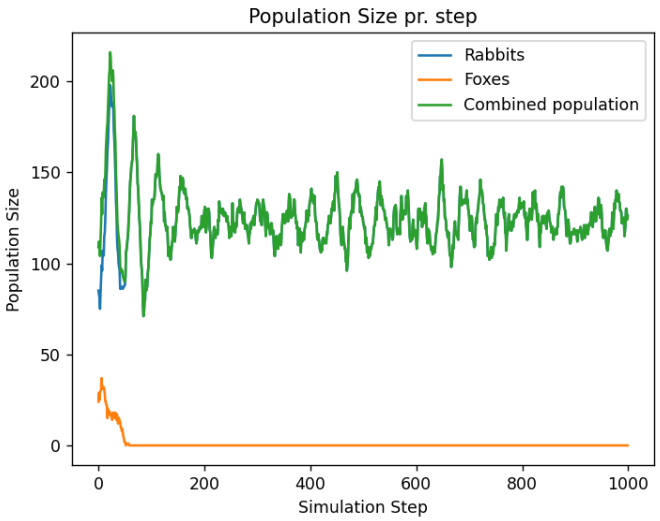


Fig. 3:
Our implementation of entities provided sensible results

Provided version of entities

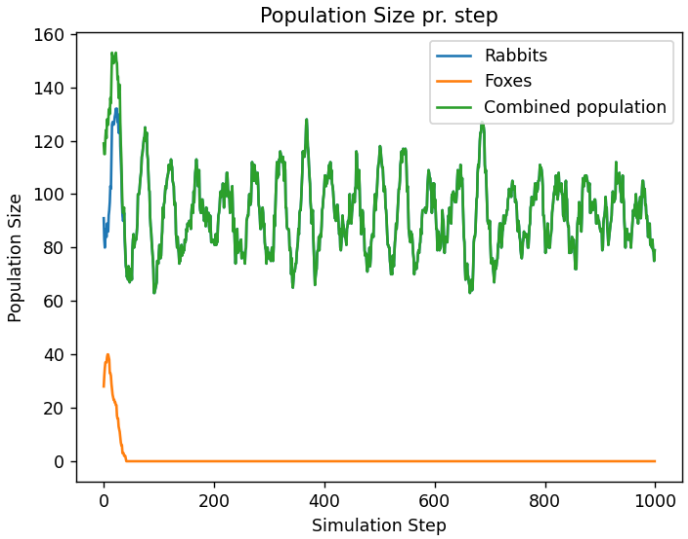


Fig. 4:
The provided version of entities provided results similar to our implementation

These results did provide further clarification about what was happening. It appears that we were counting the size of the population in each step correctly. However it seemed that dead animals weren't being removed from the patches.

Plot_energy

When plotting the energy, the underlying problem with our implementation became more apparent:

Our version of entities

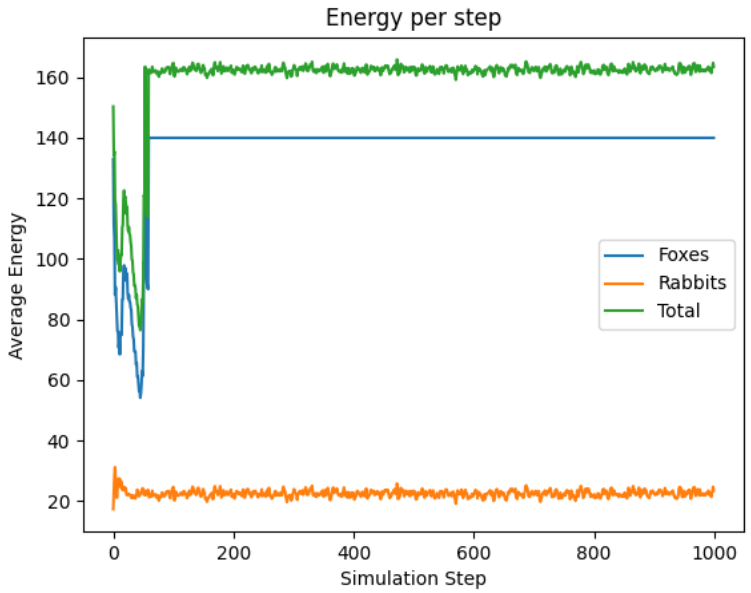


Fig. 5:
In our version of entities and in this example, the energy level flat lines at 140 for foxes.

Provided version of entities

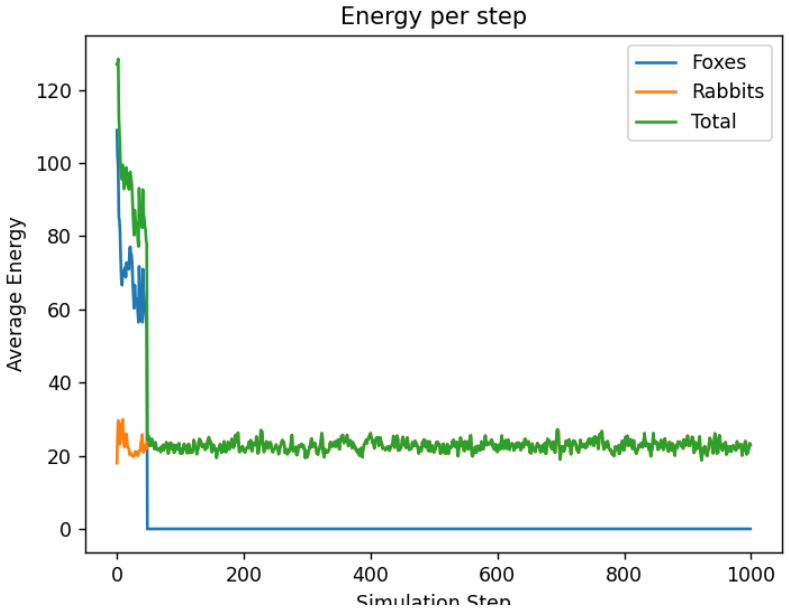


Fig. 6:

The provided version of entities looks much more sensible, with the energy level flat lining at zero, when the fox population has died out.

Even though the results of energy level did not provide clear results, it did point us towards the underlying problem. Due to the lack of fluctuations and the flat line above zero for foxes, it appeared that we keep recounting a single dead fox. This likewise suggested that the underlying problem might have been that some dead animals were not removed from their patches, or that the method *is_alive* had implementation problems.

Plot_lifespan

When plotting the lifespan of Foxes, we started out by using a line plot, however, when using this visualization our results were incomprehensible:

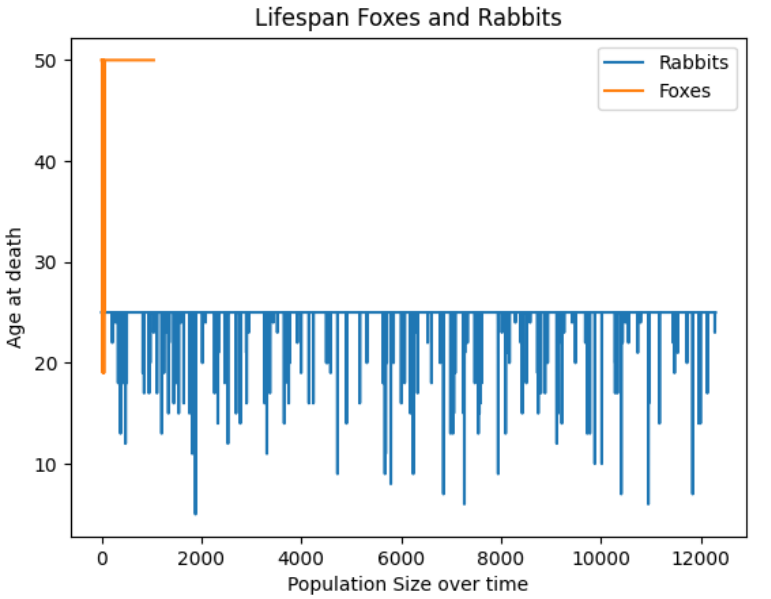


Fig. 7

Our data over age distribution when visualized with a line plot.

The reason why our visualization was off, was due to the fact that we were plotting a list of individual ages rather than the relevant data, which is how frequent each age occurs. We therefore decided to use a histogram, since matplotlibs built-in function hist automatically computes frequencies. Our first attempt at creating a histogram looked as follows:

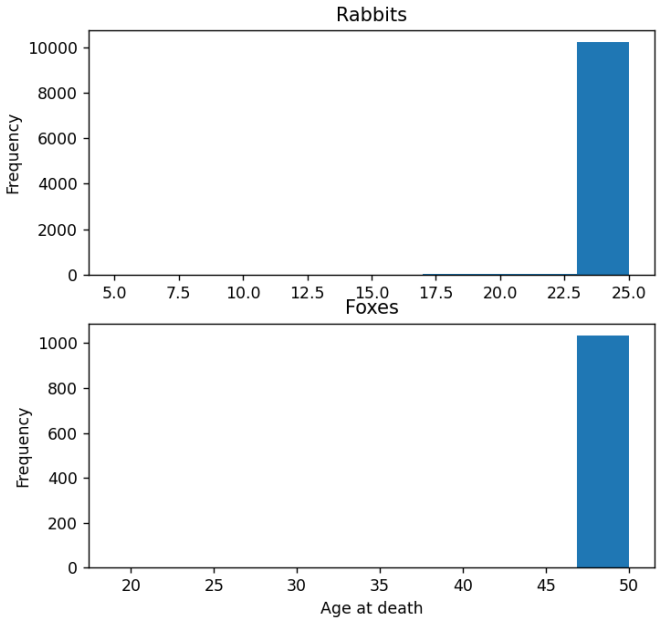


Fig. 8:

Our first histogram. Apparently, almost every animal died of old age, and there were more dead animals than animals which have ever lived.

This again showed that we recounted dead animals, since the frequencies were way too high. Moreover, it also showed us that almost all of our animals died at their max age. Lastly, the ticks on the x-axis weren't center aligned with the bars, and they sometimes displayed float values, rather than integer values. After fixing entities and tweaking the layout we ended up with the following visualization:

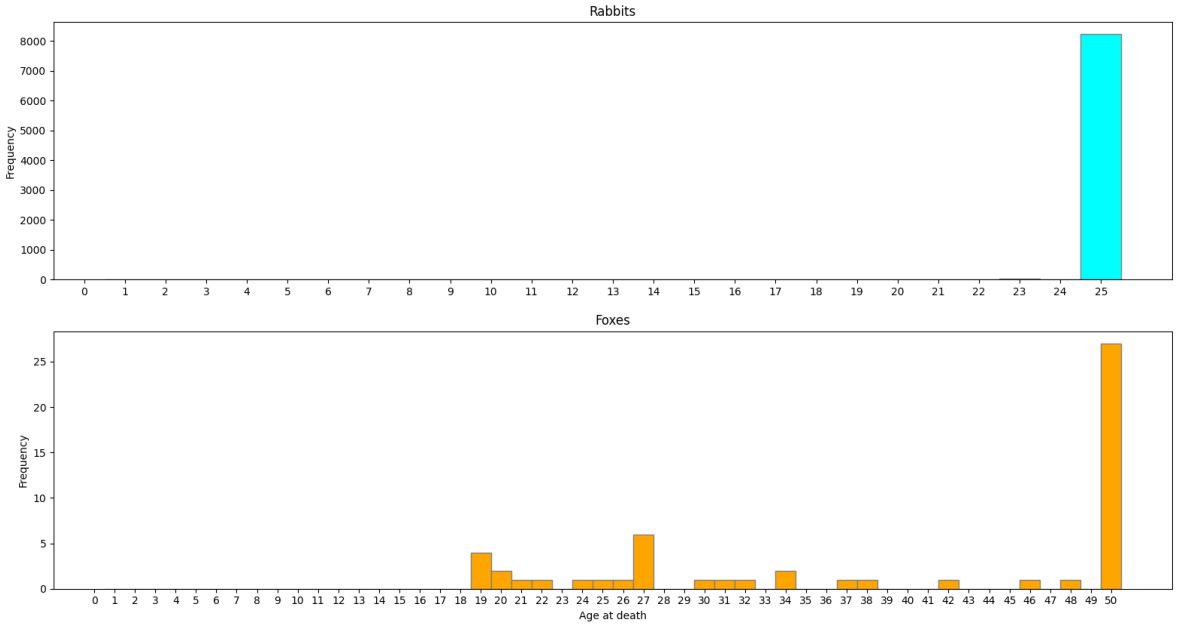


Fig. 9:
Our final histogram with the fixed version of entities

Even though it is hard to tell, there are rabbits that die below the age of 50, however, due to our default parameters, the fox population tends to die out rather quickly, which gives the rabbits ample opportunity to reproduce.

Plot_kills

As seen below, this function displays the spatial distribution of deaths on each patch.

It is clearly seen in figure 10 that the rabbits simply weren't being preyed upon

Our version of entities

Fig. 10

Our data over distribution of kills. It appears that all of the patches contain the same amount of dead animals and do not account for any fluctuations as expected.

Provided version of entities

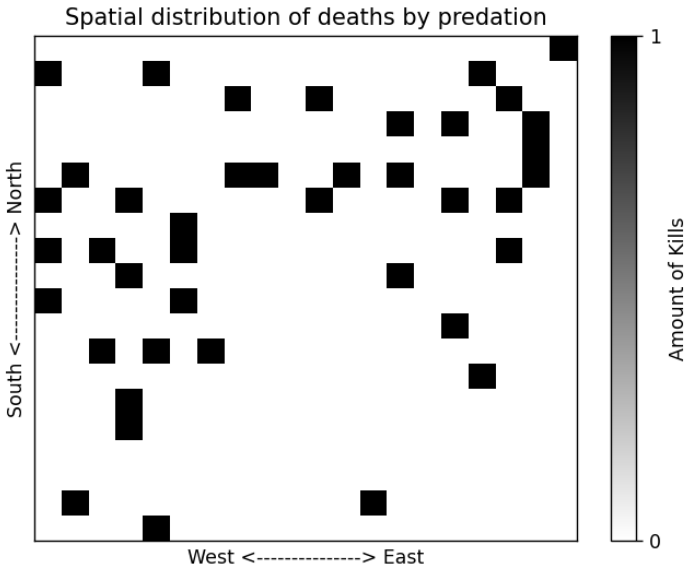


Fig. 11

The provided version of entities returns a more nuanced distribution of the animals killed.

This meant that not only did we forget to remove dead animals, we also had a problem with how the method *feed* was implemented. After fixing entities we ended up with a visualization, which made much more sense:

Our version of entities (fixed)

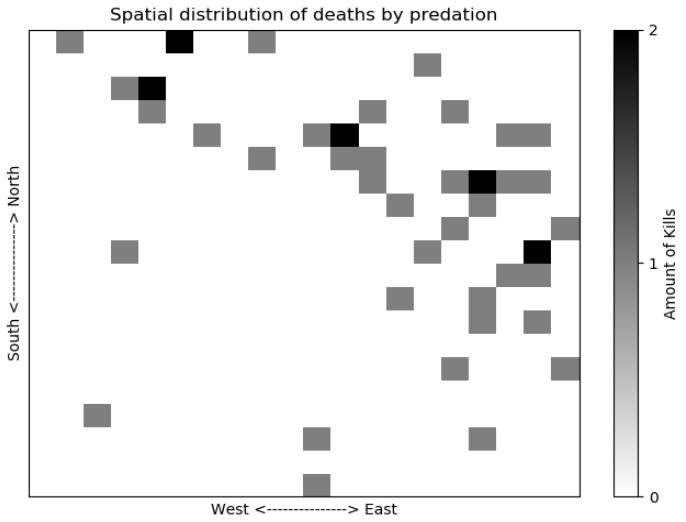


Fig. 12

Our data over distribution of kills (fixed). It provides a more realistic and nuanced distribution, with varying amounts of dead animals on each path.

By examining these errors we were able to fix entities module, so that it could provide results similar to the previous provided version of the module.

Fixing entities

The kind of bug we were dealing with, was a tricky one since it did not raise any errors. However, the results and visualizations did provide enough information for us to fix it. Since the total population seemed to be counted correctly, the problem was how dead animals were handled. Moreover, since all our animals seemed to die of old age, our animals either kept aging after they died, or their energy reserves weren't depleted, which resulted in them never dying of starvation. However, after inspecting the summary we discovered that they did indeed die of starvation, which meant that they kept aging after they died. We, therefore, concluded that the error must be that we did not always remove the animals from a patch when they died. What was happening was that when animals were dying due to reproduction (their energy reserves were depleted after reproducing), they weren't being removed from the patch. This resulted in patches, which (quite literally) were piling up with dead animals that kept being recounted. In order to catch this type of error, we, therefore, made sure to remove an animal, if it died after reproduction. In order to also catch other potential errors, we also made the method *tick* check if an animal was dead and remove it if it was.

Moreover, the foxes were not preying on rabbits, since we validated the species incorrectly. Rather than checking if the animal was an instance of a Rabbit, we checked if its species attribute (located in the class parameters) were equal to the string “Rabbit”. This resulted in us never getting a true value, since the attribute was written without capitalization. We, therefore, switched to validating the species with python’s built-in function *isinstance*, which fixed the problem.

Changes and choices:

Phase III vs. Phase II and Phase I

While phase I and II did not require us to write classes and class methods, it became the primary focus in this phase, following the logic from object oriented programming. Since the structure of the module *entities*, including its classes and functions were already given in the handout, we only had to define them. Following a bottom-up approach, each class and its methods was specified, promoting code reuse in the classes Foxes and Rabbits by inheritance from the superclass Animal, while Patch is a class on its own.

Bug-testing

Similar to phase I and II we continued to use different approaches to check our code for bugs:

- 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)
- cross-referencing our module *entities* with the provided module, to detect whether or not our program was acting as expected when given a module without bugs.

Implementation, Configuration and final thoughts

At this point the program consists of 3 layers:

- The top layer which concerns the flow of the program, user interaction and simulation setup
- The middle layer which concerns the execution of the simulation and collects data
- The bottom layer (phase III) which finalizes the simulation, implements the common data model and analyses and visualizes the collected data

For now, the program is still mostly a framework for a predator-prey model, and the simulation does not account for expected natural behavior from the animals. In the present simulation, a fox does not check a patch for a rabbit before it moves there, it moves randomly and does not show preference for choosing the rabbits that are closest to it. To improve the simulation-model, it would make sense to

add functions so the foxes choose the patches with rabbits nearest to them in a given range, and vice versa for the rabbits ie. choose patches that are further away from foxes.

The module in the present phase works as intended with the other modules, and the finished program as a whole, remains as persistent, functional, modular and readable as intended.

During the project we have utilized various elements and strategies from the course, like:

- problem solving and how a task can be broken into smaller parts
- choosing a relevant tool: working with recursion, lists, classes and methods, handling errors and using loops
- the approach of working modularly with a continuous assembling of a program
- The concept of contract programming.
- The way of writing object-oriented code.

It also required us to collaborate in different ways, both individually but also as a group and with peer-to-peer feedback and instructions.