Foxes and Rabbits

Milestone 3

Exam 2021 - Group Project

DM562 Scientific Programming
DM857 Introduction to Programming
DS830 Introduction to Programming

Functionality and Structure

This phase of the project is about refining the program from Milestone 2 by adding the last layer. This layer which consists of

- functions for the analyses and visualisation of the data collected during a simulation run (provided by module `reporting`);
- the common data model i.e., classes for storing simulation parameters (module `parameters`) and results (module `results`), and classes for modelling the entities of the simulation (module `entities`).

For this phase, you will implement module `entities` and module `reporting`. You are not required to implement the remaining two modules which are instead provided as part of the project material[1]

Common data model

Module `entities` defines the classes `Patch`, `Animal`, `Fox`, and `Rabbit` which are used to represent the entities in a simulation run.

Class `Patch` represents a patch of grass in the 2D grid of the simulation. Its public contract must contain the following.

- Class attributes `min_grass_growth = 1`, `max_grass_growth = 4`, `max_grass_amount = 30`.
- A constructor that takes two arguments, corresponding to the west-east coordinate and to the north-south coordinate of the patch, respectively.
- A method `coordinates(self) -> Tuple[int,int]` for getting the coordinates of this patch.
- A method `grass(self) -> int` for getting the amount of grass in this patch.
- A method `tick(self)` that increments the amount of grass in this patch by a random value between `min_grass_growth` and `max_grass_growth` for a maximum of `max_grass_amount`.
- Methods `has_alive_fox(self) -> bool` and `has_alive_rabbit(self) -> bool` to check if the there is a fox or a rabbit alive on this patch.
- Methods `add(self, animal: Animal)` and `remove(self, animal: Animal)` for adding and removing the given animal from the patch.

[1] This is to reduce the project workload; implementing `parameters` and `results` requires competences already covered by `entities`.

Class `Animal` represents a generic animal in the simulation and is further specialised by its subclasses `Fox` and `Rabbit`. The public contract of `Animal` must contain the following.

- A constructor that takes four arguments, the configuration parameters for the population this animal belongs to, the patch where this animal is, the current energy, and the current age of this animal.
- Methods `age(self) -> int`, `energy(self) -> int`, `patch(self) -> Patch` for reading the age, energy, and position of this animal.
- A method `is_alive(self) -> bool` for checking if this animal is alive.
- A method `can_reproduce(self) -> bool` for checking if this animal is old enough and has enough energy to reproduce.
- A method `tick(self)` to record the passage of time (one step in the simulation). If the animal is alive, it ages and consumes its energy. If the animal becomes too old or depletes its energy reserve, it dies and it is removed from its current patch.
- A method `move_to(self, patch: Patch)` to move this animal from its current patch to the one provided, if it is alive. The method assumes that the given patch is different from the current one and that it does not contain (alive) animals of the same species of this animal.
- A method `same_species_in(self, patch: Patch) -> bool` to check if the given patch contains an alive animal of the same species.
- A method `predators_in(self, patch: Patch) -> bool` to check if the given patch contains an alive predator of this animal.
- A method `feed(self)` to feed this animal using resources in its current patch, if it is alive.
- A method `reproduce(self, newborn_patch: Patch) -> Optional[Animal]` to make this animal try to reproduce using the patch provided. If a new animal is born, the method returns the instance representing the newborn, otherwise it returns `None`. The newborn is located at `newborn_patch` which is assumed to meet all necessary conditions.

The implementation of `is_alive`, `same_species_in`, `predators_in`, `feed`, and `reproduce` may depend on the specific species. You will have to decide where (`Animal`, `Fox`, `Rabbit`) and how to implement these methods.

Class `Fox` represents a fox in the simulation and extends class `Animal`. Its public contract must contain the following.

- Class attributes `reproduction_cost_rate = 0.85` and `food_energy_per_unit = 15`.
- A constructor that takes three arguments, the configuration parameters for the population this animal belongs to, the patch where this animal is, and the current age of this animal. Energy is initialised to 70% (the value is rounded to be integer).

Additionally, methods required by the public contract of the superclass `Animal` must be specialised as follows.

- Method `is_alive(self) -> bool` checks if the energy reserve of this animal is not depleted and if its age is below the age limit.

- Method `feed(self)` to feed 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. The rabbit is killed (see method `kill` of class `Rabbit`). A unit of food (one rabbit) provides `food_energy_per_unit` units of energy which are added to the energy reserve of this fox up to the maximum level possible (extra energy is simply ignored, the fox wasn't so hungry after all and didn't finish its meal).
- Method `reproduce(self, newborn_patch: Patch) -> Optional[Fox]` returns an instance of this class when successful and reduces the energy reserve of this animal by the minimum energy requirement for reproduction (`reproduction_min_energy` from `parameters.Population`) multiplied by `reproduction_cost_rate`.
- Method `predators_in(self, patch: Patch) -> bool` always returns `False` (foxes have no predators).

Class `Rabbit` represents a rabbit in the simulation and extends class `Animal`. Its public contract must contain the following.

- Class attributes `reproduction_cost_rate = 0.85` and `feeding_metabolism_rate = 2.5`.
- A constructor that takes three arguments, the configuration parameters for the population this animal belongs to, the patch where this animal is, and the current age of this animal. Energy is initialised to 25% (the value is rounded to be integer).
- A method `was_killed(self) -> bool` to check if this rabbit was killed.
- A method `kill(self)` to kill this rabbit and remove it from the current patch, if this rabbit is alive.

Additionally, methods required by the public contract of the superclass `Animal` must be specialised as follows.

- Method `is_alive(self) -> bool` checks if the energy reserve of this animal is not depleted, if its age is below the age limit (same as foxes), and if this rabbit was not killed.
- Method `feed(self)` to feed this rabbit with grass from its current patch, if the rabbit is alive. 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.
- Method `reproduce(self, newborn_patch: Patch) -> Optional[Rabbit]` returns an instance of this class when successful and reduces the energy reserve of this animal by the minimum energy requirement for reproduction (`reproduction_min_energy` from `parameters.Population`) multiplied by `reproduction_cost_rate`.
- Method `predators_in(self, patch: Patch) -> bool` checks if the given patch has an alive fox.

Analysis and reporting

Module `reporting` defines the following functions for analysing and reporting on the results of a simulation run (class `SimulationStats` is from module `results`).

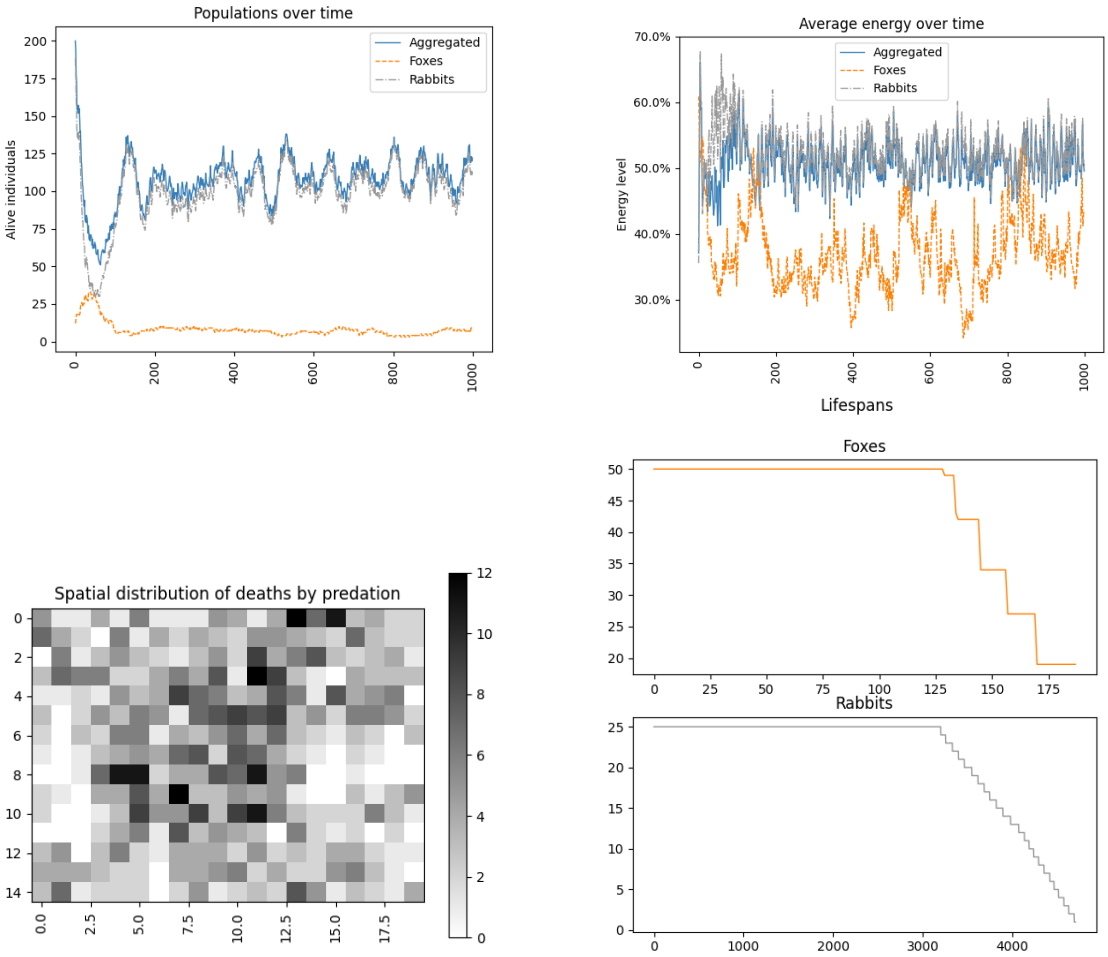

Figure 1: Examples of charts.

- Function `print_summary(results: SimulationStats)` that prints a summary of the simulation results with the following statistics for the population of foxes, rabbits, and the two populations combined.
  - The number of individuals ever lived.
  - The minimum, maximum, and average number of individuals alive at each step of the simulation.
  - The number of deaths for each cause (old age, starvation, predation) and the total.
- Function `plot_pop_size(results: SimulationStats)` that displays one or more charts that visualise how the size of populations of foxes, of rabbits, and of the two combined changes over time.
- Function `plot_kills(results: SimulationStats)` that displays one or more charts that visualise how deaths by predation are spatially distributed (kills per patch).
- Function `plot_lifespan(results: SimulationStats)` that displays one or more charts that visualise the distribution of lifespans across individuals of each population and of the two combined.
- Function `plot_energy(results: SimulationStats)` that displays one or more charts that visualise how the average energy of foxes, of rabbits, and of the two combined changes over time (it can be in absolute value or relative to the maximum energy).

You can use third party packages (Matplotlib is highly recommended) and create your own visualisations. Some examples created with Matplotlib are shown in Figure 1.

Milestone 3

Modules `entities` and `reporting`.

Hand-in

You must hand in a zip file containing:

- A PDF document named `report.pdf` containing your report.
- The python files containing your implementation of modules `entities` and `reporting` as described here.

The name of the zip file must be the name of your group e.g., `Group C18.zip` (capitalisation is immaterial). Non-compliance is ground for failure.

The report must be written in English and delivered as a single PDF file printable in black and white and long at most 15 pages excluding front and back matter e.g., an appendix (examiners are not required to consider appendices or anything above the page limit in their evaluation). The report must include the name of the group and its members (full name and SDU email where applicable).

Your code must follow the structure detailed in this document, be clearly documented, and adhere to the common coding conventions and rules of Python and this course. For this phase your code can use third party packages like matplotlib (you list and motivate their use in the report).