Programming Project: Foxes and Rabbits Simulation

Phase 1

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 our project: Phase 1

A simulation is a way to imitate real world systems or operations. Working with simulations, one can create or gather knowledge about a real-life situation without the situation having to physically take place. The precision of a simulation depends on the model that the simulation is based on. Information and data is what such a model is built upon and the more correct and accurate the information and data that is provided, the more true-to-reality a simulation can become. Although one must not forget that a simulation is only as it states: a simulation. A real life system or operation does not necessarily "play out" the way a simulation will, vice versa a simulation can contribute with a new unseen reality of an event where simple observations can struggle to provide complex results.

For this exam project we will be doing a program that can simulate a predator/prey model for foxes and rabbits. The main goal for the simulation is to provide data specific details and information about how a population of rabbits and a population of foxes will evolve over time. Also, we want to give the user an easy experience when using the simulation program.

Collaboration

Before writing the programme, we discussed how to deconstruct the project, and split it into several parts (modules). By splitting the project, we made it possible to work on each part independently. Since the range of experience in the team was quite big, it also made sense to divide the modules according to the experience of each team member. While doing this, we didn't define any coding standard, besides the obvious conventions. This leads to quite diverse coding styles in each module. The diverse coding style could potentially lead to some weaknesses in the code, like semantic errors from typing too quickly or weaknesses related to the design of the project, like trouble integrating previously written programs into a complete solution. To avoid this, and considering the future use of our modules, we used docstrings to document the code and made consensus about the visual representation of our individual approaches to write code. By doing so, it becomes clear what the meaning is with the various parameters, functions etc. in each module. In a collaborative context, it also provides an insight into how each team member has conceived the module, and makes it possible for other team members to understand the module. This also relates to anticipating the difficulties others will have understanding, debugging, modifying and using the code in other projects.

Before writing the code, we made a sketch on paper, where we discussed how we were going to program around a top-level design approach, structure the project, and use loops and/or recursion in the code. As previously mentioned, our experience with programming varies considerably, therefore, we chose to avoid while-loops and focus on recursion. This had certain advantages, first of all it forced us to make our code more modular. It likewise made us consider more carefully how each menu interacted with its submenu and parent-menu, which is more aligned with a top-level design approach.

Top-level Design

As previously stated, our project is designed around a top-level approach, where we used the provided flow-chart as a guide for creating the main components, the user interfaces, and the structure of the program. We, therefore, started out by creating the interface for the main menu (*configuration*) and its two primary sub-menus: *advanced_menu* and *reporting_menu*. Then we implemented additional sub-sub-menus. Finally, we implemented the functionality (such as letting the user change parameters or displaying the parameters) for each menu and submenu. In the reporting menu, we implemented mock functions.

Structure and Modules

As previously stated, we decided to split our menus and submenus in the following modules *foxes_and_rabbits*, *reporting_menu*, *advanced_menu*, and *config_menus*. Each menu and submenu is defined as a function, and uses recursion to let the user stay in the menu after they provide an input.

For the convenience of the user we have seperated the location of the modules. This means that the main module *foxes_and_rabbits* is placed in the main folder for itself, whereas the rest of the modules is placed in a second folder with the name "Modules". The modules path is appended/joined in the foxes_and_rabbits module (See code).

For this separation or structure to work, an empty python module with the name "*__init__*" has been created and placed in the "Modules" folder, to indicate that this directory contains code for a python module. The empty "*__init__*" file sets up how packages and functions will be imported into the other python files.

The overall structure of the modules and its functions looks as follows:

`foxes_and_rabbits module`:

- **`ascii_text`**:
  A function that prints a ascii-art generated text "Foxes_and_rabbits"
- **`display_parameters`**:
  A function, which displays the parameters for the simulation.
- **`quick_setup`**:
  A submenu to the configuration menu which lets the user configure selected parameters.
- **`configuration`**:
  The top level main menu.

`advanced_menu module:`

- **`advanced_menu`**:
  A submenu to the configuration menu which lets the user configure all parameters.

`config_menus module:`

- **`configure_world`**:
  A submenu to the advanced menu, which lets the user configure world parameters.
- **`select_shape`**:
  A submenu to configure_world, which lets the user select the shape of the world.
- **`configure_species`**:
  A submenu to the advanced_menu, which lets the user configure the parameters for a species.
- **`configure_execution`**:
  A submenu to the advanced_menu which lets the user configure the parameters for the execution of the simulation.
- **`select_mode`**:
  A submenu to configure_execution, which lets the user configure the execution modality.

`reporting_menu module`:

- **`reporting_menu`**:
  A submenu to the configuration menu which is displayed after the simulation is run. It currently initiates mock functions rather than reporting a result.

By separating the functions into specific modules, our code became more readable and modular. However, it also created circular dependencies between our modules, since it is always possible to go from a submenu to the parent menu. These circular dependencies could be avoided by using recursive calls inside the parent menus and using *pass* in the submenus when "go back / exit" was selected. However, we chose to keep the circular dependencies since it seems more sensible for a human reader, that the submenus were calling the parent menus instead of being passed.

Foxes_and_rabbits Module

As mentioned in the chapter "Structures and Modules", the "foxes_and_rabbits" module will for this project work as our main script. It is here that a user can set up and edit the way in which the simulation should take place. For the user to configure the simulation, a configuration menu "configuration" is defined and when initiated, the user can display and edit necessary parameters for the simulation-model to use for simulation later on. For the whole design to make sense, we need a main menu that calls our sub-menus. Furthermore, we also need this main module so that we can work under the same scope at all times, with the same parameters.

For the configuration function to work it needs a set of parameters that in this case comes from the pre-given module parameters. Creating a new instance of the class Simulation we now have all default values for the simulation-model and it is now possible to feed it to our functions. This new instance "params" is created when the main script/the module is run and for the other modules' functions to work properly, the "params" instance will be passed as parameters to their functions. To make it easier for the user to run a simulation with new parameters, we implemented an ability to reset current parameters to default values. This is done by simply overwriting the current instance of the parameters.Simulation/"params" class with itself.

Advanced_menu Module

This module consists of a single function called *advanced_menu*, which takes an instance of the class "Simulation" from the module "parameters" as a parameter. This function displays a selection menu, which lets the user choose four parameters to configure in the simulation:

- **`World`**
- **`Rabbit population`**
- **`Fox population`**
- **`Execution`**

Depending on the user choice, the function then activates submenus by calling one of the following functions from the module *config_menus*:

- **`configure_world`**
- **`configure_species`**
- **`configure_execution`**

The menus in the module *config_menus* were originally part of the module *advanced_menu*, however we chose to split the module between advanced_menu, which is the parent menu, and the submenus which was moved to the *config_menus* module. This was done, since two smaller modules; one for the parent menu (*advanced_menu*) and one for the submenus (*config_menus*) seemed more readable, modular and maintainable.

Config_menus Module

This module is a collection of menus which lets the user configure the parameters for the simulation. They all have *advanced_menu* as their parent or grandparent menu (see section "Structure and Modules" for an overview of the menu structure). All functions take an instance of the class "Simulation" from the module "parameters" as a parameter. Two of the functions *select_shape* and *select_mode* have *configure_world* and *configure_execution* respectively as parent menus. We considered moving these two functions to their own module, or moving them to their own module along with their parent menus. However, we decided against this, since we determined that it seemed more understandable and coherent to have all submenus and sub-submenus of *advanced_menu* bundled together in the same module.

Reporting_menu Module

Reporting_menu is a function that takes results from the function "run", in the Simulation module. It offers actions selection from the user, and summarizes and reports results from the simulation. It lets the user choose between 5 functions:

- **`summary`**
- **`plot population size/time`**
- **`plot lifespan`**
- **`plot energy`**
- **`plot kills distribution`**

By taking the chosen action, the corresponding mock function from module "reporting" is activated as a mock implementation. Since, this module is currently a mock module, it is the menu, with the least functionality, and we expect it to gain considerably more functionality in the next phases.

Error-handling and Input Validation

When validating whether a user input is valid, we find three types of input, which could break our script:

- Invalid types
- Parameters that does not satisfy a specific precondition
- An invalid selection in a menu

To get around these types of errors, "Try/except & if/else" statements are carefully used to prevent the user from entering invalid values that will result in a crash of the program. If a value is invalid or does not live up to a precondition from other modules (see section "Testing preconditions"), the user is sent back to the corresponding menu or section from which the error occurred and is asked to try again. By this, the script will only break if the user terminates the program from the main menu or the user does a `Keyboard Interruption`: `"Ctrl+C"`.

The Try/except statements are, in our project, exclusively used for handling type errors that occur when converting input from string to integers. In our current implementation, this exclusively happens when the user configures simulation parameters.
Input for menu choices are handled exclusively using if-else statements. If and elif statements specify which choices are allowed, while a final else statement makes sure that everything else is handled by an error message and a recursive call. By using this approach, where we whitelist some specified input and blacklist everything else, we avoid cluttering our code with try-except statements and we avoid having to deal with every single edge-case that might occur, when an user inputs data. If-else statements are likewise used to handle preconditions in the simulation parameters.

Testing Preconditions

The modules "parameters" and "simulation" follow the principle of contract programming. In this design paradigm classes and functions have strictly defined preconditions, which are expected to be met, in order for them to execute properly. Since we can't rely on the user input to follow the preconditions, we have to validate and test if the input meets the preconditions. Whenever the user have the opportunity to manually enter a simulation parameter, which happens in the menu *quick_setup* (foxes_and_rabbits module) and the menus in the module *config_menus*, we, therefore, test the following preconditions:

- **World parameters:**
    - *North-South length:* Must be a positive integer.
    - *West-East length:* Must be a positive integer.
- **Species population parameters:**
    - *Initial size:* Must be a positive integer, that is smaller than the size of the world.
    - *Metabolism:* Must be a non-negative integer.
    - *Max energy:* Must be a positive integer.
    - *Max age:* Must be a positive integer.
    - *Reproduction probability:* Must be a floating point representing probability, which is a float from 0 to 1.

However, certain values, which are not specified as violating the preconditions, seem absurd to accept as valid input. So we further validate user input for the following:

- **Species population parameters:**
    - *Reproduction minimum age:* Must be a non-negative integer.
    - *Reproduction minimum energy:* Must be a non-negative integer.
- **Execution parameters:**
    - *Max number of simulation steps:* Must be a positive integer
    - *Step delay:* Must be a non-negative integer.

By testing for both the specified preconditions and creating our own preconditions to validate, we therefore ensure that the user input always will be valid, abide by the contract and have a reasonable value, which guarantees that our script will run without bugs.

Bug-testing:

For bug-testing we tried some different approaches to make sure that our code would not fail during runtime. The following procedures were taken into account when we worked with bug-testing of our program:

- Test that each choice given by the user will not return an error or break the script.

- Test that invalid inputs will raise an error and send the user back to the corresponding menu.

- Move back and forth from different menus.

- Give true inputs and test that the correct parameters are printed when parameters are displayed.

- Running the mock-implementation of the function “run” from the pre-given “simulation” module to test that parameters are set correctly and that preconditions are met.

- 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.

Future Implementation and Configuration

There are several general considerations for the future implementation and configuration of our code. In the next phases of the project, we will have to adapt and build upon our existing code and modules. Future implementation will show whether or not our code is compatible over time, and whether or not future features/modules will integrate with our existing code, without creating bugs. Since the project is based on a modular approach, we should avoid having to modify and update most of our existing code, and adding new functions should be easy. Moreover, writing the project is a dynamic process, and while we continue to add other modules to the existing parts, in the later stages of the project, we will also be able to apply new knowledge to improve our old solutions, if/when it is necessary.

As for now, we feel our code is as efficient and intuitive as it can be for the first phase, but as the project evolves, and our knowledge expands, it is possible that parts of it are upgradeable and can be optimised or done in another manner. However, at this point of the project writing persistent, functional, and readable code is the primary goal and by using a top-down approach, we predict that upgrading the code will not break the program.