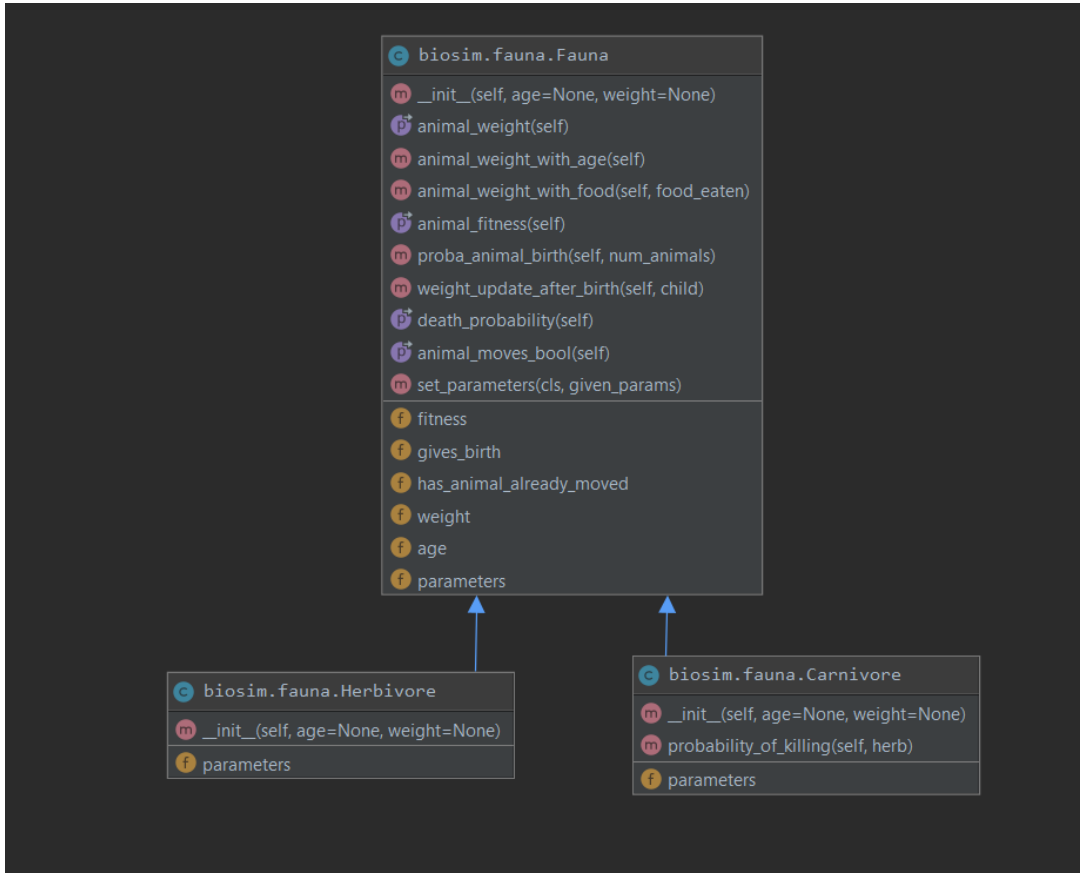
BioSim project INF200

By Ashesh Raj Gnawali and Martin Bø

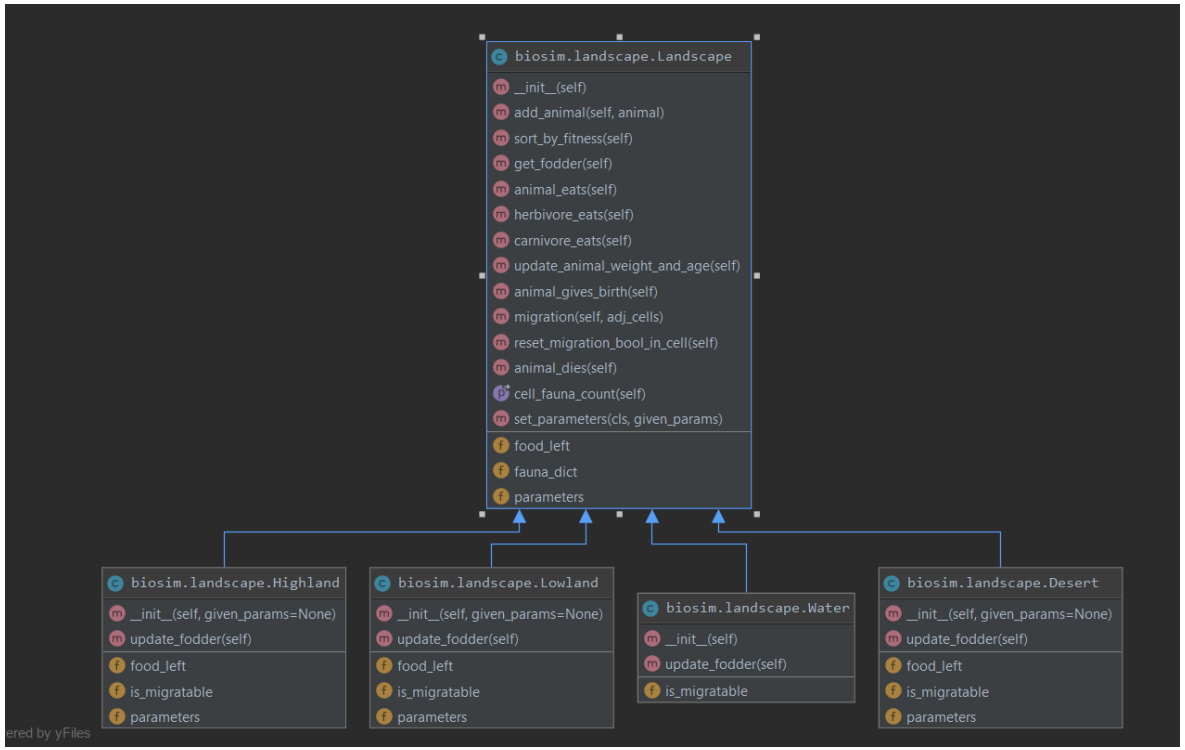
22.06.2020

Brief overview of code structure - Fauna



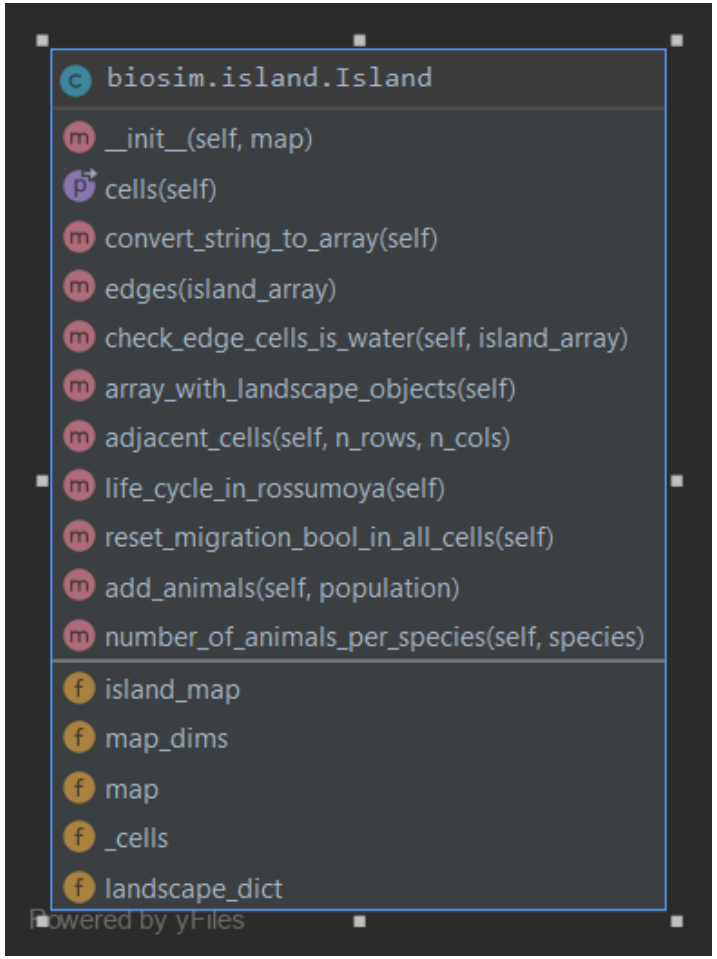
- Parent class “Fauna”
  - Child classes
    - Herbivore
    - Carnivore
  - All methods in Fauna apply for both animals, besides the probability of killing which is specific for the carnivores

Brief overview of code structure - Landscape



- Parent class “Landscape”, includes common methods for all landscape types
  - Child classes Highland, Lowland, Water and Desert
    - These have slightly different parameters and values where the desert is migratable and contains no food for herbivores. Water is also not migratable
  - Most functions in landscape are used in the lifecycle method in the island class

Brief overview of code structure - Island



- The Island only contains one class, Island.
  - Here the multiline string input is converted to a 2D array and made to include landscape objects
  - This is where the map and animals are added according to the cells specified.
  - Methods involving migration and location is in the island class, which refers to landscape types in the landscape class.

Optimization

By changing numpy.exp to math.e for the calculation of fitness, the program ran twice as fast. This test was run for 300 years where carnivores were introduced after 100 years.

Name	Call Count	Time (ms)	Own Time (ms)
animal_fitness	116755680	511928 49,9 %	511928 49,9 %
<method 'uniform' of 'numpy.random.mtrand.Ran	40795857	119213 11,6 %	119213 11,6 %
probability_of_killing	32987807	492407 48,0 %	62961 6,1 %
<built-in method builtins.input>	1	40968 4,0 %	40968 4,0 %
carnivore_eats	47100	643632 62,7 %	31622 3,1 %

Name	Call Count	Time (ms)	Own Time (ms)
animal_fitness	116755680	131759 25,0 %	131759 25,0 %
<method 'uniform' of 'numpy.random.mtrand.Ran	40795857	95675 18,1 %	95675 18,1 %
<built-in method builtins.input>	1	33613 6,4 %	33613 6,4 %
probability_of_killing	32987807	138089 26,2 %	28782 5,5 %
carnivore_eats	47100	244252 46,3 %	20901 4,0 %

Possible improvements

- The optimization could be improved further by splitting functions with calculation in two, where one part does the calculation and the other returns a bool by checking multiple conditions. This allowes the usage of packages like numba and the @jit decorator to speed up the calculations.
- @njit(parallel = True, fastmath= True)

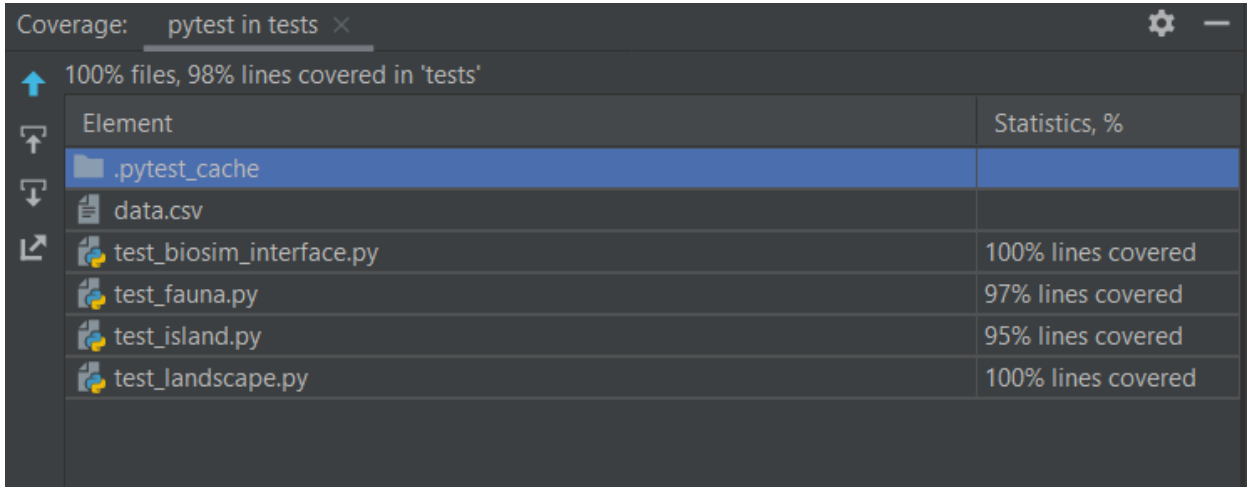
```python
@property
def animal_fitness(self):
    """
    Calculates the fitness of an animal based on age and weight \n
    """
    if self.weight > 0:
        q_pos = 1 / (1 + e**(
            self.parameters['phi_age'] * (self.age - self.parameters['a_half'])))

        q_neg = 1 / (1 + e**(
            -1 * self.parameters['phi_weight'] * (self.weight - self.parameters['w_half'])))

        return q_neg * q_pos
    else:
        return 0
```

Test coverage and reliability

- We have written a variety of tests covering all the major parts of our code. Test-driven development was our starting point, where we tried to write tests and make sure to pass them before moving on in the code. We used kanban cards in Github to list issues describing the problems and programmed parallely
- Our tests cover most of the code, and when we ran tests with coverage in pycharm we get the following results:



Simple code and detailed documentation

- The picture on the right shows an example of one of our methods. It updates the animals weight after eating. Variables names are chosen such that it is easily readable and understandable, even for unexperienced programmers.
- The doc strings are detailed enough that it explains what the function does.
- Thus making the documentation generated through Sphinx more detailed and easy to use

```python
def animal_weight_with_food(self, food_eaten):
    """
    Updates the weight of an animal based on it's feeding behavior \n
    :param food_eaten: the amount of food eaten by an animal, float
    """
    self.weight += self.parameters['beta'] * food_eaten
    return self.weight
```

Play video

Thank you for your attention