# Homework 3 - Modeling an Ecosystem

Due Date: Tuesday December 1, 3:30 p.m.

## Assignment Overview

In this assignment, you will develop a simple model of an ecosystem and simulate its behavior over time. You will do so using essential aspects of object-oriented programming, including inheritance, encapsulation, and polymorphism.

Your model will be a "world" consisting of a 2-dimensional grid of "cells" populated with grass, wildebeest, and lions. The simulation will proceed day-by-day; each day the grass will grow, wildebeest will eat grass, and lions will (try to) eat wildebeest. Wildebeest and lions both may reproduce. Animals may move from one cell to another if they aren't getting enough food. Plants and animals will die if they get too old, if they don't get enough energy to sustain themselves, or don't get enough water. Through the behaviors you implement, you will see how the state of the world evolves over time.

Over the course of the C-, B-, and A-levels of this assignment, you will define a set of classes and progressively more complex data and methods to implement more and more aspects of the simulation.

## General notes

- All instance variables should be private to their classes (unless otherwise specified). Follow the Python conventions to name private instance variables. This implies that if an instance variable should be accessible from outside its class, you must define get and set methods. For example, an **Organism** will have an **age**. If any methods from classes other than **Organism** need to access the age of an **Organism**, you must define a **get_age** method, and if any methods from classes other than **Organism** need to modify (set) the age of an **Organism**, you must define a **set_age** method.
- All instance variables for a class should be set by the constructor (the __init__ method) of the class.
- You ***must*** implement all the classes, instance variables, and methods we define below ***exactly as described***. However, you are free to define any additional methods or functions that you think make your solution simpler and clearer.
- You will want to print the contents of your World to see what's going on and whether your code is working right. We provide a method called print_world that will do this, in the file print_world.py. This also required us to define the __str__ method for the Cell class. It *also* requires a method get_org_count which will tell you how many organisms of a particular type there are in a cell ("P" → Plants; "W" → Wildebeest; "L" → Lion). We're

going to leave it up to you to write this method; and you'll need to do so if you want to use the print_world method. You are free to use and modify these methods any way you want; we won't look at them when we grade your assignment.

# General Requirements

You are required to turn in two files: simulation.py, containing the required functions, and README.txt, which must specify which level you are attempting/claiming to have completed. As always, if you do the homework with a partner, both of you must include your names in the README.txt file.

# Grade Level Descriptions

## C-Level

**Overview**. At the C-level, your job is mostly to define the classes used in the simulation, along with some very basic behavior, essentially that organisms age and lose energy just from living. You will define 8 classes and constructors and methods for some of the classes. For each of the classes, we provide detailed specifications of the instance variables, constructor (if any), and methods (if any).

### *Classes*

#### World

*Instance Variables*

- **grid**
    - A 2-dimensional array, i.e., a list of lists. Each element in the grid is an instance of the Cell class.

*Constructor*

- In addition to **self**, the constructor should take a **grid_size** parameter, which specifies the size of the (square) grid. The default value of the **grid_size** parameter should be 10.

*Methods*

- **get_grid(self):** return the grid.
- **set_grid_at_location(self, x, y, cell):** store **cell** at index [**x**][**y**] of the grid.
- **simulate(self):** simulate the world for one "day". At the C-level, simply call the **update()** method for each Cell in the grid.

- **populate(self):** initialize the world by creating organisms and storing them in the cells of the world. Each cell should be populated by a random number of organisms as follows:
    - **Grass**: Between 300 and 500;
    - **Wildebeest**: Between 60 and 100;
    - **Lions**: Between 12 and 20.
- **run(self, days)**: simulate the world for the specified number of days.

## Cell

*Instance variables*

- **organisms**: a list of the Organisms in the cell.
- **plant_capacity**: an integer that represents the maximum number of **Plants** that the cell can hold.

*Constructor*

- Initialize organisms to an empty list; accept the plant capacity as a parameter, with 1000 as the default value.

*Methods*

- **add_organism(self, organism):** add **organism** to the end of the list of the cell's organisms.
- **delete_organism(self, organism):** remove the organism from the list of the cell's organisms.
- **update(self, world[1]):** simulate a day in the life for this cell. First, check whether the cell has more plants than its **plant_capacity**; if so, remove plants until the **plant_capacity** is reached; delete from the beginning of the list, i.e., the first-added (oldest) plants. Then call the the **update_organisms** method to let the organisms in the cell live for a day.
- **update_organisms(self, world):** for each organism in the cell, first check if the organism is alive by calling the organism's **is_alive()** method. If the organism is alive, simulate the organism living one day by calling the organism's **live_one_day()** method. If the organism is not alive, delete the organism from the cell's list of organisms.

## Organism

*Instance Variables*

- **energy**: an integer that represents the organism's current "energy level".

[1] You might wonder why this method and the **updateOrganisms** method take **world** as a parameter, but do not use it. The reason is that, in higher grade levels, these methods will need to use the **world** parameter, so we chose to require it here at the C-level too.

- **age**: an integer that represents the number of days the organism has been alive.
- **lifespan**: an integer that represents the organism's maximum age in days.
- **location**: a tuple (integer, integer) that represents the organism's (x,y) coordinate within its world's grid.
- **upkeep:** an integer that represents how much energy the organism needs to live one day.

*Constructor*

- The constructor should accept energy, lifespan, location, and upkeep as parameters. The constructor does not need to take age as a parameter, but instead should simply set the age instance variable to 0.

*Methods*

- **is_alive(self):** return true if an organism is alive and false if it isn't. An organism dies if its energy is less than its upkeep (that means it doesn't have enough energy to make it through the day!) or if its age is greater than its lifespan.
- **live_one_day(self, world):** increase the age of the organism by 1, and subtract the energy the organism needs to live a day (its upkeep) from its energy.

## Plant

The **Plant** class is a subclass of the **Organism** class. At the C-level, this class does not introduce any additional instance variables or methods. However, you must define a constructor for the class, which simply calls the constructor of the superclass.

## Animal

The **Animal** class is a subclass of the **Organism** class. At the C-level, this class does not introduce any additional instance variables or methods. However, you must define a constructor for the class, which simply calls the constructor of the superclass.

## Grass

The **Grass** class is a subclass of the **Plant** class. This class does not introduce any additional instance variables or methods. However, you must write a constructor for the class, which specifies default values for some (inherited) attributes (you will do this by calling the constructor of the superclass):

- energy=50
- lifespan=5
- upkeep=2

### Wildebeest

The Wildebeest class is a subclass of the Animal class. This class does not introduce any additional instance variables or methods. However, you must write a constructor for the class, which specifies default values for some (inherited) attributes (you will do this by calling the constructor of the superclass):

- energy=7500
- lifespan=30
- upkeep=100

### Lion

The Lion class is a subclass of the Animal class. his class does not introduce any additional instance variables or methods. However, you must write a constructor for the class, which specifies default values for some (inherited) attributes (you will do this by calling the constructor of the superclass):

- energy=5000
- lifespan=25
- upkeep=50

## B-Level

**Overview**. At this level you will not define any new classes. However, you will add some instance variables and methods to the classes you've already defined in order to enable a more realistic simulation.Specifically, you will implement exchange of energy between organisms -- animals will be able to consume other organisms to obtain energy, plants will be able to produce their energy from sunlight and water in the cell, and an animal may move to an adjacent cell if it fails to find food in its current location.

### World

*New method*

- **move_organism(self, organism):** change the location of an organism within the world's grid (intuitively, an organism might move if it cannot find enough food in its current location). Randomly select the organism's new location from one of the cells adjacent to its current location, horizontally, vertically, or diagonally. Update the grid as well as the organism's own location (using the appropriate method). You also will use the new **add_immigrant** method of the **Cell** class (see below) to ensure that when an organism moves it doesn't get to "live again" on the same day in its new location.

*Modified method*

- **simulate()**: update this method to call the new **move_immigrants** method of the Cell class (see below) on each cell after all cells have been updated.

## Cell

*New instance variables*

- **sunlight**: a float value randomly initialized between 0 and 1 that represents the intensity of sunlight in the cell.
- **water**: a float value randomly initialized between 0 and 1 that represents the water level in the cell.
- **immigrants**: a list that contains the organisms that immigrated to this cell during the course of the current (simulated) day; initially an empty list.

*New methods*

- **add_immigrant(self, organism):** adds **organism** to the list of immigrants.
- **move_immigrants(self):** adds the immigrants to the end of the list of organisms, then resets the immigrants to an empty list.

## Plant

*New instance variables:*

- **production_power**: an integer that represents the ability of the plant to produce energy.

*New method*

- **produce_energy(self, water, sunlight):** simulate the production of energy by a plant in a day. The amount of energy produced is based on the amount of water and sunlight available, calculated according to the formula:

$$e = p * min(⅛ * water, sunlight) + max(⅛ * water, sunlight)^2 * 100$$

  where p is the **production_power** of the plant.
  Update the energy of the plant with the newly produced energy.

*Overridden method*

- **live_one_day(self, world)**: this method first calls the **live_one_day** method of the superclass (that is, it “lives” as a normal organism would), then call the **produce_energy()** method. Use appropriate get methods of the Cell class to obtain the water and sunlight values to pass to **produce_energy()** method.

## Animal

*New instance variables*

- **fill**: a float value between 0-1 that represents how full the animal's stomach is (0=empty, 1=full).
- **foods**: a dictionary whose keys are organisms the animal can eat, and values are floats between 0-1 that specify how much more the animal's stomach will get filled if it eats the organism. For example, if an animal's stomach was 0.2 full, and the animal ate an organism with a value of 0.25, the animal's stomach would become 0.45 full.

*Modified constructor*

- Modify the constructor; add a foods parameter to the end of the parameter list. Initialize the animal's **fill** to 0.

*New methods*

- **consume(self, organism, world):** simulates the consumption of an organism (plant or animal) by an animal. When an organism is consumed, 1/10th of its energy is added to the energy of the consuming animal. This method should also update the value of **fill** and make appropriate function call to delete the consumed organism from the animal's cell (note: an animal can consume only organisms from the same cell).
- **hunt(self, world):** simulates hunting activity for animals. In a given day, an animal will keep hunting until either it has lost more than ⅓ of its initial energy for the day or its stomach is full. An animal hunts by choosing another organism from its cell at random. If the chosen organism is a potential **food** source for the animal, it will **consume** it. If the organism is not a food source for the animal, then the animal's energy will drop by 100.
- **migrate(self, world):** checks whether the animal has consumed enough food to be **full**. If not, it should call the appropriate method to move the organism to a different cell on the grid.

*Overridden method*

- **live_one_day(self, world):** this method overrides the **live_one_day()** method of Animal's superclass. It should start the day with a hungry animal (**fill** = 0), call the superclass's **live_one_day()** method, make the animal **hunt**, and **migrate** (if necessary).

## Grass

Update the constructor to have a default value for the additional attribute:

- **production_power**=10

### Lion

Update the constructor to have a default value for the additional attribute:

- **foods**={Wildebeest: 0.25}

### Wildebeest

Update the constructor to have a default value for an additional attribute:

- **foods**={Grass: 0.05}

## A-Level

**Overview**. You again will add some new instance variables and methods, and update and override some previously defined methods. The main new behavior you will implement at this level is reproduction. Oh, and animals also will be able to die of thirst.

### Cell

*New methods*

- **update_sunlight(self):** updates the sunlight attribute of the cell with a new, random float value between 0-1.
- **update_water(self):** updates the water attribute of the cell with a new, random float value between 0-1.

*Modified method*

- **update(self, world):** modified to include calls to our new methods above; water and sunlight should be updated at the end of the Cell's update process.

### Organism

*New method*

- **reproduce(self, world):** adds a new organism of the same type to the same cell (as an immigrant). They are treated as immigrants so that newborns will begin living on the following day, not the day they were born.

*Modified method*

- **live_one_day(self, world):** modified to include reproduction.

### Animal

*New instance variables*

- **gestation_time**: an integer that represents the number of days until an animal can reproduce; update the constructor to accept this as a parameter.

- **days_until_fertile**: an integer that is initialized to be equal to the **gestation_time** and counts down daily to know when an animal can reproduce.
- **thirst**, a float value between 0-1 that represents how thirsty an animal is (0=not thirsty, 1=*extremely thirsty*); initialize to 0.

*New methods*

- **reproduce(self, world):** simulates reproduction. If the animal is ready to reproduce (check the value of **days_until_fertile**), do so by calling the **reproduce** method of the superclass and resetting **days_until_fertile**.
- **found_water(self, world):** simulates a random chance of an animal finding water in the cell. It returns true if any out of 5 tries generates a random number less than the water level of the cell; think of the animal looking in five different places for water.
- **drink(self, world):** checks whether water is available. If so, animal's thirst is quenched entirely. If not, add 0.25 to the value of thirst.

*Modified method*

- **live_one_day(self, world):** decrement **days_until_fertile**; add reproduction and drinking water.

*Overridden method*

- **is_alive(self):** returns true if the superclass's **is_alive()** method returns true and the animal is not *extremely thirsty*.

## Wildebeest

- Update the constructor to accept a new parameter for gestation_time; default value is 15.

## Lion

- Update the constructor to accept a new parameter for gestation_time; default value is 10.

*Overridden method*

- **migrate(self, world):** makes lions less likely to migrate when they're hungry. Recall that at the B-level, all animals migrate at the end of the day if they didn't eat enough to get full. Now you'll say that lions who don't get their fill have only a 1 out of 3 chance of migrating.