COMP 204 - Assignment 4

Yue Li

Important instructions:

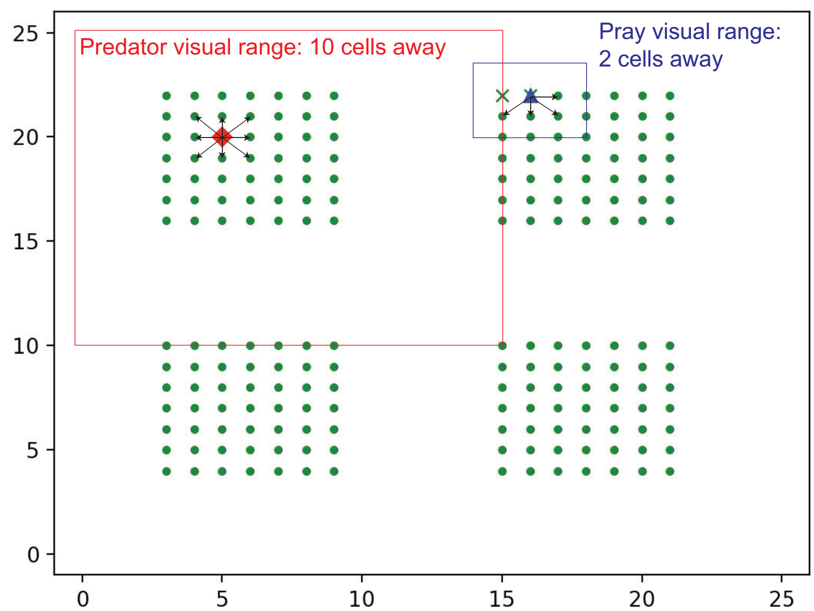
- Release date: March 15, 2019 at 12:00 AM
- Due date: **March 29, 2019 at 23:59**
- Download the following Python files from MyCourses:
  1. Animal.py: define `Animal` class attributes and methods (Questions 1-6)
  2. Plant.py: define `Plant` class attributes and methods (Questions 7 and 8)
  3. Terrain.py: define `Terrain` class attributes and methods (Question 9 and Bonus)
  4. Position.py: define `Position` class attributes (already completed)
  5. util.py: define some useful functions (already completed)
  6. ecosim_test.py: this file will be used to test your answers to the programming questions (already completed)
  7. ecosim_test_output.txt: this file contains the expected text output from running ecosim_test.py for each question except for the bonus question, which will generate a plot.
  8. ecosim_animation.py: this file is used to visualize the ecosim simulation (you can use it to debug your code or simply for fun) (already completed)
  9. ecosim.mp4: animation saved by running the ecosim_animation.py (uncomment the last line)
- Complete all of the 9 (+ 1 bonus) programming questions specified in the Animal.py, Plant.py, and Terrain.py files (Total 100 points + 10 bonus points)
- Submit the following three completed files separately on MyCourses:
  1. Animal.py
  2. Plant.py
  3. Terrain.py
- Write your name and student ID at the top of each file
- Do not use any modules or any pieces of code you did not write yourself
- For each question, your program will be *automatically* tested on a variety of test cases, which will account for 75% of the mark. To be considered correct, your program should

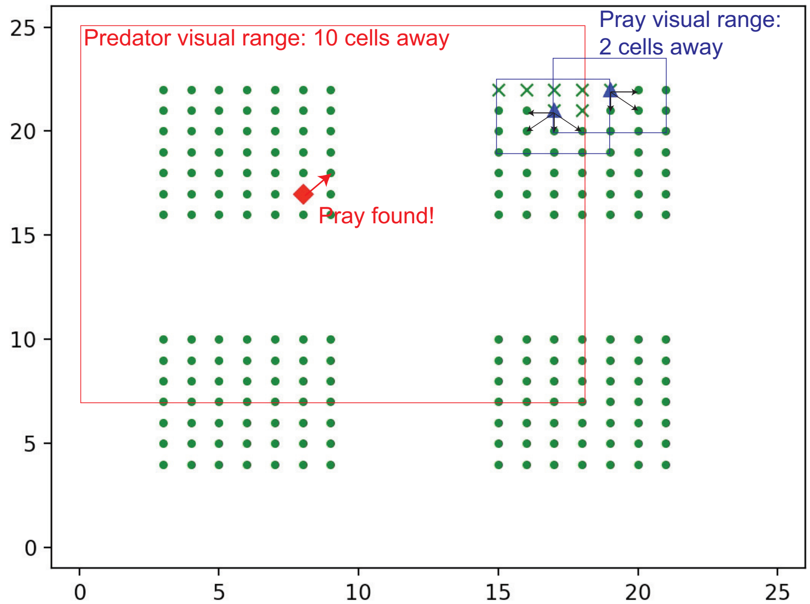
print out *exactly and only* what the question specifies. Do not add extra text, extra spaces or extra punctuation.

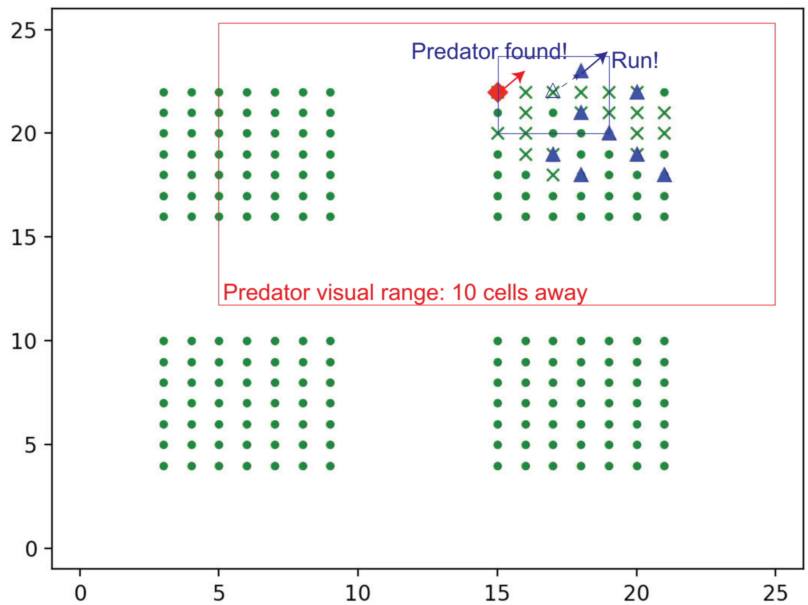
- For each question, 25% of the mark will be assigned by the TA based on (i) your appropriate naming of variables; (ii) commenting of your program; (iii) simplicity of your program (simpler = better).
- **A gentle note**: Despite the somewhat complex overall OOP design of the simulation program, the actual programming tasks that require you to complete are in fact quite straightforward. Please take this assignment as an opportunity to learn especially for those class attributes and methods that have been completely written and provided to you. Related questions may be asked in the final exam.

Ecosystem simulation

In this assignment, we will write an *object oriented program* (OOP) that simulates a simple ecosystem. As listed below, there are four main classes in this program. Each class has its own attributes and methods. To have some intuition about the simulation, take a look at the animation (ecosim.mp4) provided in this assignment. The animation is generated by the completed simulation program.







In the specific simulation run recorded in ecosim.mp4, initially there are one predator (red diamond), 20 prays (blue triangle), and 196 plants (49 green dots or crosses on each of the four 7 x 7 squares) on a 25 by 25 terrain grid.

As shown in the above figures, each predator and pray can move in 8 different directions (vertically, horizontally, and diagonally). They cannot move outside of the terrain map. In particular, we specify the two-dimensional coordinate position for the animal to be x (for vertical direction) and y (for horizontal direction). Then, $0 \geq x > 25$ and $0 \geq y > 25$.

Both the pray and predator can starve to death, age to death, and spawn off-springs (i.e., self-reproduce like bacteria) within specific age ranges and frequencies. Predators need to hunt and eat prays to avoid starving to death. Prays need to find and eat the plants to avoid starving to death. Plants never die but rather become unavailable once they are consumed by the prays and regenerate to become available again after some time.

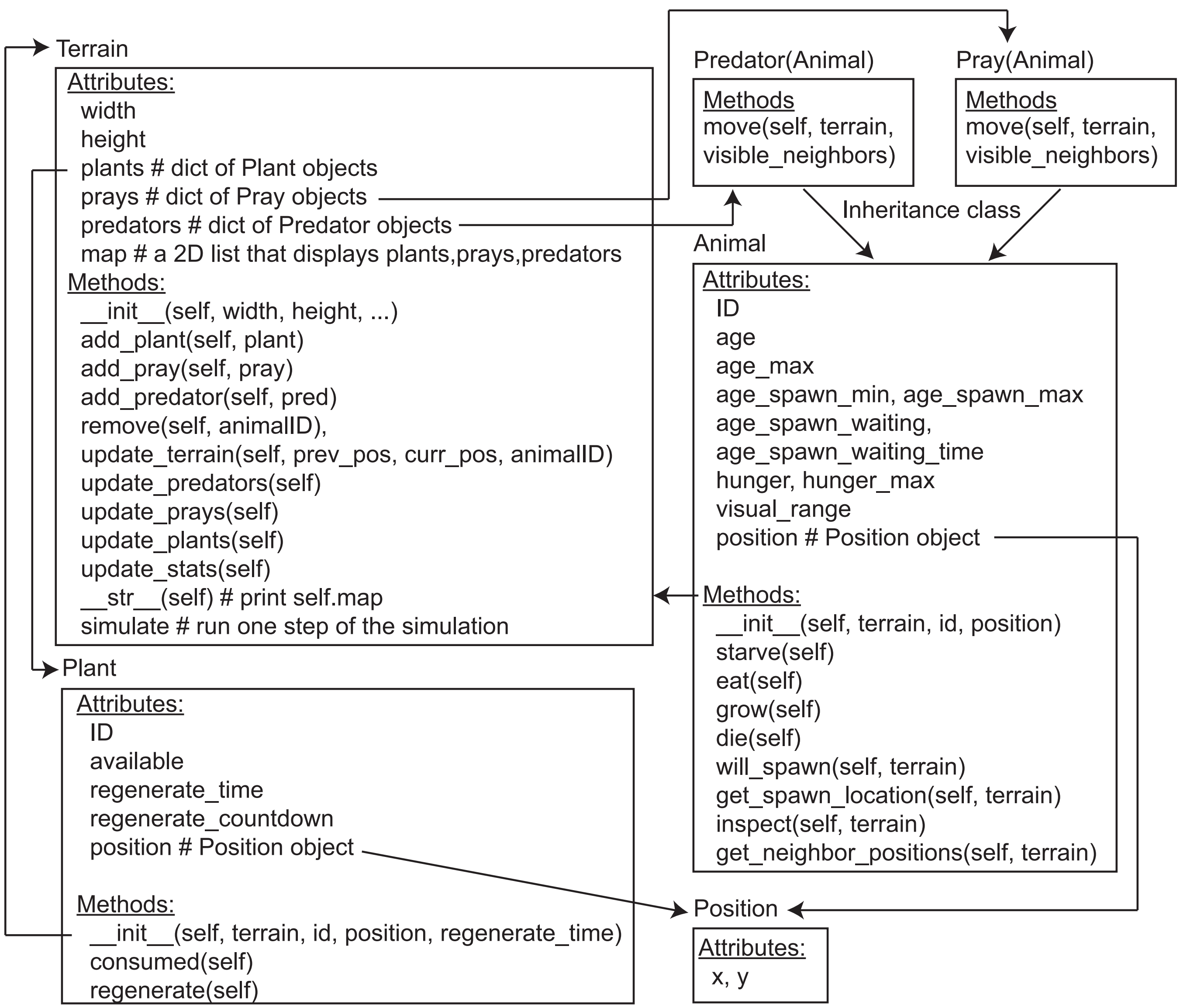
By default, each predator can see 10 steps away from its current location, whereas each pray can only see 2 steps ahead. Once predator sees a pray it will take the step that is the closest to the closet pray among all of the prays it sees; otherwise it walks randomly until it sees a pray. The predators can walk on a plant but not consume it regardless the plant is available or not. Predator cannot walk into another predator or outside of the terrain.

If pray sees a predator, it will take the step farthest from the closest predator. If pray does not see a predator, it goes for the available plant (green dot) (otherwise it walk randomly). Pray cannot walk into another pray or predator. They can walk into a plant regardless whether the plant is available.

Once the plant is eaten (green dot becomes green 'x'), it takes 5 steps to regenerate (plants never die but only become temporarily unavailable to the prays). During this regeneration time, the place occupied by the plant is no different from any of the empty spots on the terrain map.

The overall OOP design

The overall OOP design of the ecosim is illustrated here and detailed in the subsequent sections:



`Animal` class defined in `Animal.py`

The `Animal` class has the following attributes described by the comments:

```python
def __init__(self, terrain, id, position=()):
    self.id = id # animal identifier
    self.age = 0 # animal age
    self.age_max = 10 # animal max age
    self.age_spawn_min = 3 # min age animal can spawn offspring
    self.age_spawn_max = self.age_max # max age animal can spawn
    self.spawn_waiting = 0 # countdown time animal can re-spawn
    self.spawn_waiting_time = 3 # total recovery time for re-spawning
    self.hunger = 0 # hunger level (0 means not hungry)
    self.hunger_max = 3 # max hunger level
    self.visual_range=2 # how far the animal can see

    # location of animal in a xy-coordinate system
    if len(position) == 0: # random genereate location
        self.position = Position(0, terrain.width-1, 0,
         ↪  terrain.height-1)
    else: # set location to the provided position value (tuple (x,y))
        self.position = Position(0, terrain.width-1, 0,
         ↪  terrain.height-1,
            position[0], position[1], random_init=terrain.random_sim)
```

`Animal` methods

At each step of the simulation, the objects from `Predator` or `Pray` will behave as follows

- `starve`: if the animal does not eat anything, its hunger level increases by one
- `eat`: if the animal does eat something, its hunger level resumes to 0
- `grow`: the animal grows old by one year if it is not at `age_max`
- `die`: if the animal reaches to the `age_max` or `hunger_max`, it dies
- `get_neighbor_positions`: the animal obtains all of the adjacent position(s) that are 1 step away from its current positions
- `will_spawn`: if the animal reaches to `age_spawn_min` but not exceeds `age_spawn_max` and there is empty adjacent cell nearby it, it will decide to spawns a new animal of its kind
- `get_spawn_location`: it choose randomly from the empty locations adjacent to itself to give birth to its off-spring
- `inspect`: the animal "inspects" its surrounding within its `visual_range` and get a `list` of identifiers that tells it whether its neighbours are empty cells, plants, or other

animals

`Animal` inheritance classes: `Predator` and `Pray`

Two inheritance classes `Predator` and `Pray` are derived from the `Animal` class. Predator must kill and eat prays to keep its `hunger` level below `hunger_max` or it will die. Besides not being killed by the predators, prays need to find and eat plants to keep its `hunger` level below `hunger_max` to stay alive.

Both classes share the same attributes and methods except for the method `move`. Predators and prays have different move strategies. Each predator either moves towards its closest pray (once it finds it via the `inspect` method) or randomly moves to the adjacent position, that is either empty or occupied only by a plant. In other words, a predator cannot move into another predator but it can stand on top of a plant.

For the pray, it will first check whether there is one or more predators around it (within its `visual_range`). If so, it will move away from the predator(s) by choosing its next position that is the farthest from its closest predator. If it sees no predator but plants, it will move towards the closest plant. If it sees neither a plant nor a predator, it will randomly move to one of the available adjacent positions that is not occupied by other prays.

`Plant` Class and Methods in `Plant.py`

The attributes of the `Plant` class are specified as follows

```python
class Plant:
def __init__(self, terrain, id, position=(), regenerate_time=5):
    self.id = id # plant identifier
    self.type = "plant" # book keeping
    self.available = True # is the plant available for consumption by the pray
    self.regenerate_time = regenerate_time # steps taken to regenerate after being consumed
    self.regenerate_countdown = 0 # the plant will regenerate when countdown is 0
    # plant's location in the terrain
    if len(position)==0: # generate random position
        self.position = Position(0, terrain.width-1, 0, terrain.height-1)
    else:
        self.position = Position(0, terrain.width-1, 0, terrain.height-1,
            position[0], position[1], random_init=terrain.random_sim)
```

Plant will never die but regenerate if they are consumed by the pray. Specifically, if the plant is consumed, then it becomes unavailable for a certain number of simulation steps set by the `regenerate_time` attribute, after which it will become available again for prays to consume.

`Position` class (`Position.py`)

To keep track the location of each entity, we use the class `Position`:

```
class Position:
  def __init__(self, min_xcoord, max_xcoord, min_ycoord, max_ycoord,
   ↪ x=0, y=0, random_init=True):
    if random_init:
        self.x = random.choice(range(min_xcoord, max_xcoord))
        self.y = random.choice(range(min_ycoord, max_ycoord))
    elif x < min_xcoord or x > max_xcoord or y < min_ycoord or y >
     ↪ max_ycoord:
        print(f"x: {x}; y: {y}")
        raise ValueError("invalid x or y coordinate")
    else:
        self.x = x
        self.y = y
```

`Terrain` class (`Terrain.py`)

To simulate the interactions among prays, predators, and plants, we will need to create another class called `Terrain`. It stores the locations and all information about all of the prays, predators, and plants by separate dictionaries.

```
class Terrain:
  def __init__(self, terrain_width=25, terrain_height=25,
    nb_predators=1, nb_prays=20, nb_plants=4*49):

    self.width=terrain_width
    self.height=terrain_height

    self.predator_symbol = "@"
    self.pray_symbol = "&"
    self.plant_avail_symbol = "."
    self.plant_consumed_symbol = "x"
    self.empty_symbol = " "

    # these are used to give unique identifiers
    # to the new prays and predators
    self.nb_prays_ever_lived = 0 # total number of dead or alive
     ↪ prays
```

```
        self.nb_predators_ever_lived = 0 # total number of dead or alive predators

        # a 2D list display locations of plants, prays and predators based on the
        # specified symbols
        self.map = [["" for x in range(self.width)] for y in range(self.height)] # empty string
        self.plants = {}
        self.prays = {}
        self.predators = {}

        # add nb_plants of Plant objects
        # add nb_predators of Predator objects
        # add nb_prays of Pray objects

        # see Terrain.py for more details
```

To initialize the simulation or update the interactions and status of plants, prays, and predators, `Terrain` uses the following methods:

- `add_plant(self,new_plant)` : add a new Plant object to the `plants` dictionary attribute in the terrain object
- `add_pray(self, new_pray)` : add a new Pray object to the `prays` dictionary attribute in the terrain object
- `add_predator(self, new_predator)` : add a new Predator object to the `predators` dictionary attribute in the terrain object
- `remove(self, animalID)` : remove an animal object from the map due to its death based on the animal ID
- `update_terrain(self, prev_pos, curr_pos, animalID`: update the terrain map based on the previous position and the current position of the animal with animalID
- `update_predators(self)` : update each predator by the order of inspect, move, grow, spawn, and then check whether each predator is dead and remove them if all dead predators
- `update_prays(self)` : update each pray by the order of inspect, move, grow, spawn, and then check whether each pray is dead and remove them if all dead prays
- `update_plants(self)` : for each plant, let it regenerate only when it applies
- `update_stats(self)` : save the number of prays, predators, and available plants at the current simulation step for subsequent analysis
- `simulate(self)` : the main function to run one entire simulation step: `update_predators`, `update_prays`, `update_plants`, `update_stats`.

q1 (5 points) complete `starve` in `Animal`

```python
def starve(self):
    """
    Args:
        self: the animal object
    Returns:
        Nothing
    Behavior:
        Increment hunger level by one if hunger level is not at max
    """

    # WRITE YOUR CODE HERE FOR QUESTION 1 (2 lines of code)
```

Test your implementation with the the code provided to you in the file `ecosim_test.py`:

```python
print("\n----Question 1 Animal starve ----")
terrain = Terrain(nb_predators=0, nb_prays=0)
pred = Predator(terrain, "pred0")
for step in range(5):
    pred.starve()
    print(f"step{step}: {pred.id} hunger level: {pred.hunger}")
pray = Predator(terrain, "pray0")
for step in range(5):
    pray.starve()
    print(f"step{step}: {pray.id} hunger level: {pray.hunger}")
```

This should output:

```
----Question 1 Animal starve ----
step0: pred0 hunger level: 1
step1: pred0 hunger level: 2
step2: pred0 hunger level: 3
step3: pred0 hunger level: 4
step4: pred0 hunger level: 5
step0: pray0 hunger level: 1
step1: pray0 hunger level: 2
step2: pray0 hunger level: 3
step3: pray0 hunger level: 4
step4: pray0 hunger level: 5
```

q2 (5 points). complete `eat` in `Animal`

```python
def eat(self):
    """
    Args:
        self: the animal object
    Returns:
        Nothing
    Behavior:
        The hunger level resumes to 0 after the animal eats
        regardless of the animal's current hunger level
    """

    # WRITE YOUR CODE HERE FOR QUESTION 2 (2 lines of code)
```

Test your implementation with the the code provided to you in the file `ecosim_test.py`:

```python
print("\n----Question 2 Animal eat ----")
terrain = Terrain(nb_predators=0, nb_prays=0)
pred = Predator(terrain, "pred0")
for step in range(5):
    pred.starve()
    print(f"step{step}: {pred.id} hunger level: {pred.hunger}")
for step in range(2):
    pred.eat()
    print(f"step{step}: {pred.id} hunger level: {pred.hunger}")
pray = Pray(terrain, "pray0")
for step in range(5):
    pray.starve()
    print(f"step{step}: {pray.id} hunger level: {pray.hunger}")
for step in range(2):
    pray.eat()
    print(f"step{step}: {pray.id} hunger level: {pray.hunger}")
```

This should output:

```
----Question 2 Animal eat ----
step0: pred0 hunger level: 1
step1: pred0 hunger level: 2
step2: pred0 hunger level: 3
step3: pred0 hunger level: 4
step4: pred0 hunger level: 5
step0: pred0 hunger level: 0
```

```
step1: pred0 hunger level: 0
step0: pray0 hunger level: 1
step1: pray0 hunger level: 2
step2: pray0 hunger level: 3
step3: pray0 hunger level: 4
step4: pray0 hunger level: 5
step0: pray0 hunger level: 0
step1: pray0 hunger level: 0
```

q3 (5 points) complete `grow` in `Animal`

```python
def grow(self):
    """
    Args:
        self: the animal object
    Returns:
        Nothing
    Behavior:
        Increase age by one if age is not at max
    """

    # WRITE YOUR CODE HERE FOR QUESTION 3 (2 lines of code)
```

Test your implementation with the the code provided to you in the file `ecosim_test.py`:

```python
print("\n----Question 3 Animal grow ----")
terrain = Terrain(nb_predators=0, nb_prays=0)
pred = Predator(terrain, "pred0")
for step in range(5):
    pred.grow()
    print(f"step{step}: {pred.id} age: {pred.age}")
pray = Predator(terrain, "pray0")
for step in range(5):
    pray.grow()
    print(f"step{step}: {pray.id} age: {pray.age}")
```

This should output:

```
----Question 3 Animal grow ----
step0: pred0 age: 1
step1: pred0 age: 2
```

```
step2: pred0 age: 3
step3: pred0 age: 4
step4: pred0 age: 5
step0: pray0 age: 1
step1: pray0 age: 2
step2: pray0 age: 3
step3: pray0 age: 4
step4: pray0 age: 5
```

q4 (5 points) complete `die` in `Animal`

```python
def die(self):
    """
    Args:
        self: the animal object
    Returns:
        True if animal dies of age or hunger; otherwise False
    """

    # WRITE YOUR CODE HERE FOR QUESTION 4 (1-2 lines of code)
```

Test your implementation with the the code provided to you in the file `ecosim_test.py`:

```python
print("\n----Question 4 Animal die ----")
terrain = Terrain(nb_predators=0, nb_prays=0)
pred0 = Predator(terrain, "pred0", position=(4,4), age_max=10,
    hunger_max=5)
pred1 = Predator(terrain, "pred1", position=(3,3), age_max=5,
    hunger_max=10)
for step in range(5):
    for mypred in [pred0, pred1]:
        mypred.grow()
        mypred.starve()
        if mypred.die():
            if mypred.hunger == mypred.hunger_max:
                print(f"{mypred.id} died of hunger")
            elif mypred.age == mypred.age_max:
                print(f"{mypred.id} died of age")
```

This should output:

```
----Question 4 Animal die ----
pred0 died of hunger
pred1 died of age
```

q5 (20 points) complete `get_neighbor_positions` defined in the `Animal` class

```python
def get_neighbor_positions(self, terrain):
    """
    Args:
        self: the animal object
        terrain: the object containing all information about the simulation
    Returns:
        A list of available adjacent positions as Position objects
        Suppose the animal's current position is (x,y)
        The adjacent position is (x+i,y+j), where i is in [-1,0,1] and j in [-1,0,1]
        The neighbor positions *exclude* animal's own position (x,y) (i.e., i!=0 or j!=0)
        An available position is defined as a position not occupied by any animal
        An available position can contain a plant
    Hint:
        You should use terrain.map[x][y] to obtain the identifier of a pray, a predator, or
        a plant that is in position (x,y).
        If there is nothing in position (x,y), terrain.map[x][y] will return an empty string ""
    Note:
        plant ID always starts with the word "plant" followed by a numereic value (e.g., plant0)
        predator ID always starts with the word "pred" followed by a numereic value (e.g., pred0)
        pray ID always starts with the word "pray" followed by a numereic value (e.g., pray0)
        Therefore, using regular expression we can figure out whether terrain.map[x][y] contains
        a plant, a pray, a predator, or empty
     Reminder:
         if the positions occupied only by plants are considered as *available* neighbor positions
```

```
    """
avail_neighbor_positions = []

    # WRITE YOUR CODE HERE FOR QUESTION 5  (20-30 lines of code)

    return avail_neighbor_positions
```

Test your implementation with the the code provided to you in the file `ecosim_test.py`:

```
print("\n---Question 5 Animal get_neighbor_positions ---")
terrain = Terrain(nb_predators=0, nb_prays=0, nb_plants=0)
terrain.add_predator(Predator(terrain, "pred0", position=(5,5)))
terrain.add_predator(Predator(terrain, "pred1", position=(5,6)))
terrain.add_predator(Predator(terrain, "pred2", position=(5,4)))
terrain.add_pray(Predator(terrain, "pray0", position=(4,5)))
terrain.add_plant(Plant(terrain, "plant0", position=(6,4)))
print(terrain)
print("pred0 at position", terrain.predators["pred0"].position, "has neighbor positions:")
for position in terrain.predators["pred0"].get_neighbor_positions(terrain):
    print(position)
```

This should output the following. Notice when we call `terrain.add_*`, it adds the object to the corresponding dictionary and also update the `terrain.map` that will display where the animals/plants are based on their positions.

The symbols for prays, predators, and available and unavailable plants are: &, @, ., x, respectively. Here we want to obtain all of the available the neighbour positions for the predator pred0 at position (5,5). As shown below, pred1 and pred2 are on the left and right side of pred0, and pray0 is on the top of pred0. There is also a plant on at the bottom left diagonal direction of pred0. Despite being surrounded, there are still exactly *five* available positions around pred0, namely as printed below, namely (4,4), (4,6), (6,4), (6,5), (6,6).

```
---Question 5 Animal get_neighbor_positions ---
***************************
*                         *
*                         *
*                         *
*                         *
*     &                   *
*    @@@                  *
*    .                    *
*                         *
*                         *
```

```
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
***************************
pred0 at position (5,5) has neighbor positions:
(4,4)
(4,6)
(6,4)
(6,5)
(6,6)
```

In the code snippet below, we are testing a “boundary case" where the pray named pray0 is at position (0,0). Although nothing is surrounded pray0, the fact that it is at the top left corner of the terrain map limits its available positions to only (0,1), (1,0), (1,1).

```python
terrain = Terrain(nb_predators=0, nb_prays=0, nb_plants=0)
terrain.add_pray(Pray(terrain, "pray0", position=(0,0)))
print(terrain)
print("pray0 at position", terrain.prays["pray0"].position, "has neighbor positions:")
for position in terrain.prays["pray0"].get_neighbor_positions(terrain):
    print(position)
```

```
***************************
*&                        *
*                         *
*                         *
*                         *
*                         *
*                         *
```

```
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
*                         *
***************************
pray0 at position (0,0) has neighbor positions:
(0,1)
(1,0)
(1,1)
```

q6 (15 points) complete `will_spawn` in the `Animal` class

```python
def will_spawn(self, terrain):
    """
    Args:
        self: the animal object
        terrain: the object containing all information about the simulation
    Returns:
        False if animal cannot spawn; otherwise True
    Behavior:
        Animal will only spawn if ALL 4 conditions are satified:
        (1) at specified age range;
        (2) hunger level lower or equal to 2
        (3) spawn waiting time is 0
        (4) there is at least one available adjacent position around the animal to let it spawn
        If animal does not satisfy (1) and (2), return False
```

```
        If animal satisfies (1) and (2) but not (3), decrease spawn_waiting by 1 and returns False
        If animal satisfies (1),(2),(3) but no (4), return False
        If animal satisfies all conditions, set spawn_wating to spawn_waiting_time and return True
    """

    # WRITE YOUR CODE HERE FOR QUESTION 6 (10-20 lines of code)
```

Test your implementation with the the code provided to you in the file `ecosim_test.py`:

```python
print("\n---Question 5 Animal will_spawn ---")
terrain = Terrain(nb_predators=0, nb_prays=0)
pred0 = Predator(terrain, "pred0", position=(1,2), age_spawn_min=3, age_spawn_max=7, spawn_waiting_time=2)
pred1 = Predator(terrain, "pred1", position=(5,6), age_spawn_min=3, age_spawn_max=7, spawn_waiting_time=2)
pred1.starve()
pred1.starve()
pred1.starve() # pred1 should never spawn at hunger level 3
for step in range(10):

    pred0.grow()
    pred0_spawn_waiting = pred0.spawn_waiting
    if pred0.will_spawn(terrain):
        print(f"step{step}: {pred0.id} will spawn at age {pred0.age}, hunger {pred0.hunger}, waiting {pred0_spawn_waiting}")
    else:
        print(f"step{step}: {pred0.id} will *not* spawn at age {pred0.age}, hunger {pred0.hunger}, waiting {pred0.spawn_waiting}")

    pred1.grow()
    pred1_spawn_waiting = pred1.spawn_waiting
    if pred1.will_spawn(terrain):
        print(f"step{step}: {pred1.id} will spawn at age {pred1.age}, hunger {pred1.hunger}, waiting {pred1_spawn_waiting}")
    else:
        print(f"step{step}: {pred1.id} will *not* spawn at age {pred1.age}, hunger {pred1.hunger}, waiting {pred1.spawn_waiting}")
```

```
---Question 5 Animal will_spawn ---
step0: pred0 will *not* spawn at age 1, hunger 0, waiting 0
step0: pred1 will *not* spawn at age 1, hunger 3, waiting 0
step1: pred0 will *not* spawn at age 2, hunger 0, waiting 0
step1: pred1 will *not* spawn at age 2, hunger 3, waiting 0
step2: pred0 will spawn at age 3, hunger 0, waiting 0
```

```
step2: pred1 will *not* spawn at age 3, hunger 3, waiting 0
step3: pred0 will *not* spawn at age 4, hunger 0, waiting 1
step3: pred1 will *not* spawn at age 4, hunger 3, waiting 0
step4: pred0 will *not* spawn at age 5, hunger 0, waiting 0
step4: pred1 will *not* spawn at age 5, hunger 3, waiting 0
step5: pred0 will spawn at age 6, hunger 0, waiting 0
step5: pred1 will *not* spawn at age 6, hunger 3, waiting 0
step6: pred0 will *not* spawn at age 7, hunger 0, waiting 1
step6: pred1 will *not* spawn at age 7, hunger 3, waiting 0
step7: pred0 will *not* spawn at age 8, hunger 0, waiting 1
step7: pred1 will *not* spawn at age 8, hunger 3, waiting 0
step8: pred0 will *not* spawn at age 9, hunger 0, waiting 1
step8: pred1 will *not* spawn at age 9, hunger 3, waiting 0
step9: pred0 will *not* spawn at age 10, hunger 0, waiting 1
step9: pred1 will *not* spawn at age 10, hunger 3, waiting 0
```

q7 (5 points) complete `consumed` in `Plant`

```python
def consumed(self):
    """
        Args:
            self: the plant object
        Returns:
            Nothing
        Behavoir:
            Set plant to unavailable
            Set regenerate_countdown to regenerate_time
    """

    # WRITE YOUR CODE HERE FOR QUESTION 7 (2 lines of code)
```

```python
print("\n----Question 7 Plant consumed----")
terrain = Terrain(nb_predators=0, nb_prays=0, nb_plants=0)
plant0 = Plant(terrain, "plant0", position=(2,2), regenerate_time=3)
print(f"Before consumed, plant0.available: {plant0.available}")
print(f"After consumed, plant0.available:
 ↪  {plant0.regenerate_countdown}")
plant0.consumed()
print(f"After consumed, plant0.available: {plant0.available}")
print(f"After consumed, plant0.available:
 ↪  {plant0.regenerate_countdown}")
```

```
----Question 7 Plant consumed----
Before consumed, plant0.available: True
After consumed, plant0.available: 0
After consumed, plant0.available: False
After consumed, plant0.available: 3
```

q8 (10 points) complete `regenerate` in `Plant`

```python
def regenerate(self):
    """
        Args:
            self: the plant object
        Returns:
            Nothing
        Behavoir:
            If regenerate_countdown is greater than 0 and plant is not available
            decrease regenerate_countdown by one
            If regenerate_countdown is 0 and plant is not available
            set plant to be available (i.e., regenerated)
         Note:
             A plant can reduce regenerate_countdown by one and become available at the
             same simulation step
    """

    # WRITE YOUR CODE HERE FOR QUESTION 8 (4 lines of code)
```

```python
print("\n----Question 8 Plant regenerate----")
terrain = Terrain(nb_predators=0, nb_prays=0, nb_plants=0)
plant0 = Plant(terrain, "plant0", position=(2,2), regenerate_time=3)
plant0.consumed()
for step in range(4):
    print(f"Step{step}, plant0.available: {plant0.available}", end=', ')
    print(f"plant0.regenerate_countdown: {plant0.regenerate_countdown}")
    plant0.regenerate()
```

```
----Question 8 Plant regenerate----
Step0, plant0.available: False, plant0.regenerate_countdown: 3
Step1, plant0.available: False, plant0.regenerate_countdown: 2
Step2, plant0.available: False, plant0.regenerate_countdown: 1
Step3, plant0.available: True, plant0.regenerate_countdown: 0
```

q9 (20 points) complete `update_prays` in `Terrain`

```python
def update_prays(self): # Q9
    """
    Args:
        self: terrain object
    Returns:
        Nothing
    Behavoir:
        Update each pray in the following order
            (1) pray inspects to get a list of neighbors within its visual range
            (2) prays moves to get a tuple containing previous and current positions
            (3) terrain update the map according to the prev_pos, curr_pos of the pray
            (4) pray checks whether it will spawn a new off-spring
                if will_spawn is True, then get the spawn location and add the new pray
                    to the terrain
        Check each pray to see whether it dies
            if it dies:
                print out one of the two:
                    (1) {prayID} died of hunger
                    (2) {prayID} died of age
                remove the pray from the terrain
    Hint:
        update_prays is very similar to update_predators
    """
    # WRITE YOUR CODE HERE FOR QUESTION 9 (20-30 lines of code)
```

```python
print("\n----Question 9 complete update_prays in Terrain----")
terrain = Terrain(nb_predators=4, nb_prays=1)
print(terrain)
for step in range(10):
```

```
    print(f"\n------Step {step}------")
    terrain.update_prays()
    print(terrain)
```

```
# Check output in assignment4_output.txt
```

Bonus Question (10 points) complete `update_stats` in `Terrain`

Complete the following function.

```
def update_stats(self):
    """
        Args:
            self: terrain object containing all information about the simulation
        Returns:
            Nothing
        Behavior:
            Modify three attributes:
            (1) nb_prays_over_time:
                a list containing the numbers of predators at each simulation step
            (2) nb_prays_over_time:
                a list containing the numbers of prays at each simulation step
            (3) nb_avail_plants_over_time:
                a list containing the numbers of *available* plants at each simulation step
    """

    # WRITE YOUR CODE HERE FOR THE BONUS QUESTION (6 lines of code)
```

You can run the plotting code provided to you to generate some statistics:

```
print("\n----Bonus Question update ecosim statistics----")
terrain = Terrain()
for i in range(500):
    terrain.simulate()

plt.close()
plt.clf()  # clears figure to generate a new one
```

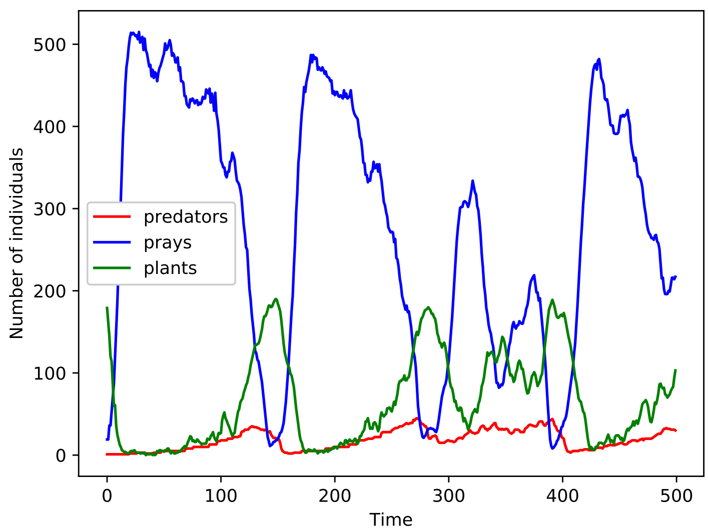
```python

plt.plot(terrain.nb_preds_over_time, 'r', label="predators")
plt.plot(terrain.nb_prays_over_time, 'b', label="prays")
plt.plot(terrain.nb_avail_plants_over_time, 'g', label="plants")

plt.xlabel('Time')
plt.ylabel('Number of individuals')
plt.legend(loc="best")

plt.savefig("ecosim.eps")
```

This should generate exactly the following plot to get the bonus points.



Just for fun (not to be marked)

- Modify the `Terrain` to let it incorporate more attributes based on the following `__init__` method:

```python
class Terrain:
    def __init__(self, terrain_width=25, terrain_height=25,
                 random_sim=False,
                 nb_predators=1, nb_prays=20, nb_plants=4*49,
                 pred_age_max=35, pred_age_spawn_min=20,
                 pred_age_spawn_max=32,
                 pred_spawn_waiting_time=6,
                 pred_hunger_max=13, pred_visual_range=10,
                 pray_age_max=30, pray_age_spawn_min=2,
```

```
                pray_hunger_max=10,
                pray_spawn_waiting_time=1,
                pray_visual_range=2,
                plant_regenerate_time=5)
```

- Play around with the animation function by changing various simulation settings when creating the `Terrain` object (e.g., what happen if you change Predator `pred_age_spawn_max` from 20 to 10 or `pred_age_max` from 50 to 5000 but with `pred_age_spawn_min` also set to 5001 (i.e., an immortal but not self-reproducible predator).
- Can we find the settings that will keep both the predators and prays alive for over 500 steps with at least 50 random simulations (i.e., `Terrain(random_sim=True)` while keeping a good number of predators and prays alive (i.e., not making predator immortal and offspringless)?
- Based on the simulation data for 500 steps, can we predict the number of prays, predators and available plants in the next 500 or more steps? In other words, is there a pattern that we can extract from this simulation?
- Conversely, given a desirable pattern over 500 steps, can we predict its simulation settings?
- What if the prays can communicate with each other to make them aware of the predators even if the predator is outside of their “visual range"?