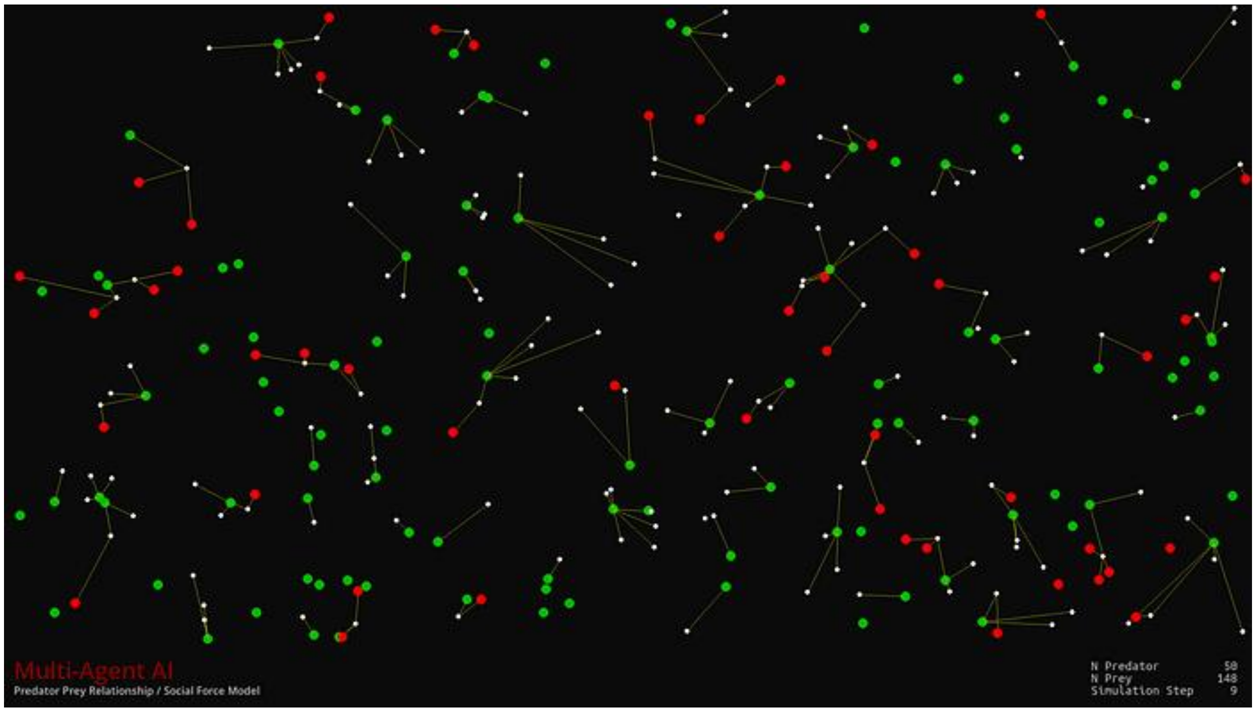
How to program a simple multi-agent predator-prey simulation in Python using Pygame

Chris the Multi-Agent Guy



Typical predator-prey systems feature various animals that compete for resources.

This article demonstrates how to program a simple **multi-agent predator-prey simulation** in Python. While we program a simple system, this example covers spacial multi-agent simulations, object-oriented Python programming, the Pygame package, biological population dynamics, and differential equations.

Nothing here is new — the author and many others already created such simulations 20 years ago. What has changed is the speed of computers and the domination of **Python** in (data) **science**. It is now possible to build such simulations with just a few lines of code and integrate the whole Python ecosystem. You can create a simulation system like the one in this video in less than one hour.

Some Background Information

Biological Predator-Prey Systems have been analyzed for a long time. Applicable mathematical models were developed independently by Alfred J. Lotka in 1920 and by Vito Volterra in 1926. The ordinary **differential equations** they found are now known as classic *Lotka-Volterra equations*.

If you are interested in these systems, see, for example, this introduction for data scientists by Andrea C.[1].

These equations model the **dynamics** of two animal species. However, things get complex as soon as a third species or other food is involved.

More importantly, these differential equations say nothing about the spatial distribution of the two animal species or the plant food. To model this, we need a simulation.

Each animal has a position, x and y. It also has a velocity vector, Δx and Δy. The new position is calculated in each simulation time step by adding the Δ to the old position.

$x = x + \Delta x$
$y = y + \Delta y$

Easy. And yet we arrive at another set of differential equations, but since they are now discrete, we can handle them with a computer program.

Step 1: The Pygame Framework

Pygame is a python module for 2D graphics written by Pete Shinners in the early 2000s. It is not the fastest graphics framework available, but nowadays, computers are super quick, and our simulation will be pretty simple anyways.

A scientific simulation is not a computer game, but we need a way to generate a **2-dimensional graphic** display that refreshes at a sensible frequency. Pygame offers this and is, therefore, an excellent choice to start with. There are many other packages available that you can use as well.

The following code snippet is directly from the Pygame documentation, only slightly adapted. It shows the basic concepts of the **main loop**, which handles all events like keyboard interaction or updates of the animal's positions.

```
import pygame

# Define constants for the screen width and height
SCREEN_WIDTH = 2560
SCREEN_HEIGHT = 1440

pygame.init()
screen = pygame.display.set_mode((SCREEN_WIDTH, SCREEN_HEIGHT))
clock = pygame.time.Clock()

# Insert Agent class definitions here
# ...

# Initial agent lists go here
# ...

while True:
    # Process inputs
    for event in pygame.event.get():
        if event.type == pygame.QUIT:
            pygame.quit()
            raise SystemExit

    screen.fill("black")  # Fill the display with a solid color

    # Do agent updates here
    # ...

    # Agent housekeeping here
    # ...

    pygame.display.flip()  # Refresh on-screen display
    clock.tick(24)         # wait until next frame (at 24 FPS)
```

This code will draw a window of the size 2560 × 1440, as specified in the screen = … line. It then fills the background with solid black and refreshes the screen at 24 Hz.

Step 2: Agents

All animals, predators and prey, are modeled as ‘**agents**’. In standard object-oriented programming, objects are abused for everything. Here, they are a natural fit. Every agent is an object, and we have different types of agents.

We define the generic behavior of an agent in the class Agent(). There is a lot of boilerplate but in essence, we initialize the default values and provide functionality that the agent can draw a representation of itself on the screen.

It is the responsibility of the update() function to calculate the new position of an agent and update its position on the screen. For now, the agent is stationary.

```
class Agent(pygame.sprite.Sprite):
    def __init__(self, size, color, x=None, y=None):
        super().__init__()

        # draw agent
        self.surf = pygame.Surface((2*size, 2*size), pygame.SRCALPHA, 32)
        pygame.draw.circle(self.surf, color, (size, size), size)
        self.rect = self.surf.get_rect()

        # default values
        self.vmax = 2.0

        # initial position
        self.x = x if x else random.randint(0, SCREEN_WIDTH)
        self.y = y if y else random.randint(0, SCREEN_HEIGHT)

        # initial velocity
        self.dx = 0
        self.dy = 0

        # inital values
        self.is_alive = True
        self.target = None
        self.age = 0
        self.energy = 0

        # move agent on screen
        self.rect.centerx = int(self.x)
        self.rect.centery = int(self.y)

    def update(self, screen, food=()):
        self.age = self.age + 1

        # update graphics
        self.rect.centerx = int(self.x)
        self.rect.centery = int(self.y)
        screen.blit(self.surf, self.rect)
```

The agent needs to **react** to the environment and **move** around. It is supposed to move towards something it can eat; prey or plants.

We develop the content of the update() function backward, starting with the position update. As mentioned before, the new position is the current position plus the velocity. We add the equations and ensure the agents stay within the window boundaries.

```
# update position based on delta x/y
self.x = self.x + self.dx
self.y = self.y + self.dy

# ensure it stays within the screen window
self.x = max(self.x, 0)
self.x = min(self.x, SCREEN_WIDTH)
self.y = max(self.y, 0)
self.y = min(self.y, SCREEN_HEIGHT)
```

To update the position, we need to know the velocity vector, Δx/Δy. Every agent has a *desired velocity*. This is how fast and in which direction the agent wants to move if it can.

It makes sense to think of a force that pushes the agent in a certain direction. This force can have various components, as described in the paper **Social Force Model for Pedestrian Dynamics** by Dirk Helbing and Peter Molnar [2].

We use the force that points toward the next bit of food as a start. Certainly, it would make sense to implement the full social force model for more realistic behavior. This might be covered in another article.

For now, we add the force to the agent's velocity. The *desired velocity* is slowly (by 5% per time step, therefore the constant 0.05 in the code) turned toward the nearest bit of food.

```
# update our direction based on the 'force'
self.dx = self.dx + 0.05*fx
self.dy = self.dy + 0.05*fy

# slow down agent if it moves faster than it max velocity
velocity = math.sqrt(self.dx ** 2 + self.dy ** 2)
if velocity > self.vmax:
    self.dx = (self.dx / velocity) * (self.vmax)
    self.dy = (self.dy / velocity) * (self.vmax)
```

For the force calculation, we take the difference between the agent's position and the position of the next food item.

```
# initalize forces to zero
fx = 0
fy = 0

# move in the direction of the target, if any
if self.target:
    fx += 0.1*(self.target.x - self.x)
    fy += 0.1*(self.target.y - self.y)
```

Now the agent can move toward food. For this, it needs to know where the food is and which bit is nearest to the agent's position.

The agent needs to **iterate** over all food items and calculate their distance. It then remembers the nearest and moves in that direction until it has reached the item. Once there, the agent eats the food and gains energy.

```
# target is dead, don't chase it further
if self.target and not self.target.is_alive:
    self.target = None
```

```
# eat the target if close enough
if self.target:
    squared_dist = (self.x-self.target.x)**2 + (self.y-self.target.y)**2
    if squared_dist < 400:
        self.target.is_alive = False
        self.energy = self.energy + 1

# agent doesn't have a target, find a new one
if not self.target:
    min_dist = 9999999
    min_agent = None
    for a in food:
        if a is not self and a.is_alive:
            sq_dist = (self.x - a.x) ** 2 + (self.y - a.y) ** 2
            if sq_dist < min_dist:
                min_dist = sq_dist
                min_agent = a

    if min_dist < 100000:
        self.target = min_agent
```

Defining the behavior of the agents was the hardest part of our simulation. We can now move on to create predators, prey, and food and then populate our simulation.

Step 3: Different Types of Agents

Since we took the time to define a class with the generic behavior of the agents, it is now easy to create predators and prey. Both inherit the generic functionality from the Agent() class.

They have different sizes, colors, and different maximum velocities. Of course, they eat different things, too. We will cover that aspect later.

```
class Predator(Agent):
    def __init__(self, x=None, y=None):
        size = 4
        color = (255, 0, 0)
        super().__init__(size, color)
        self.vmax = 2.5

class Prey(Agent):
    def __init__(self, x=None, y=None):
        size = 2
        color = (255, 255, 255)
        super().__init__(size, color)
        self.vmax = 2.0
```

Step 4: Populate the Simulation with Agents

It is now possible to create as many predators and prey as we want. You can play with initial numbers. As it turns out, the system is quite robust. It will not convert to an equilibrium, but the population sizes will **oscillate** within stable boundaries. For now, we start with 10 predators that chase 10 prey.

```
preys = [Prey() for i in range(10)]
predators = [Predator() for i in range(10)]
```

Step 5: Add Plants

The predators eat prey, while the preys eat plants. Interestingly, we can model plants as agents that can't move (but dies when eaten). We just set the color to green and the maximum velocity to zero.

```
class Plant(Agent):
    def __init__(self, x=None, y=None):
        size = 4
        color = (0, 128, 0)
        super().__init__(size, color)
        self.vmax = 0
plants = [Plant() for i in range(100)]
```

Step 6: Let them Chase Each Other

The bulk of the work is done. We have defined our simulation world and agent behavior. All we have to do is call all agents' update() functions in every simulation timestep. In the call for this function, we list what these agents can eat.

```
[a.update(screen) for a in plants]
[a.update(screen, food=plants) for a in preys]
[a.update(screen, food=preys) for a in predators]
```

They will chase the next available food item, prey or plant. If something is eaten, its is_alive property is set to False in the update() method. We then have to update the lists of agents accordingly and remove dead agents.

Also, plants **reproduce** automatically. At every timestep, we add 2 new plants to the simulation to compensate for the ones eaten.

Predators and preys **gain energy** each time they eat something. If their energy reaches a certain level, they reproduce and duplicate. Their offspring appears at a location close to them.

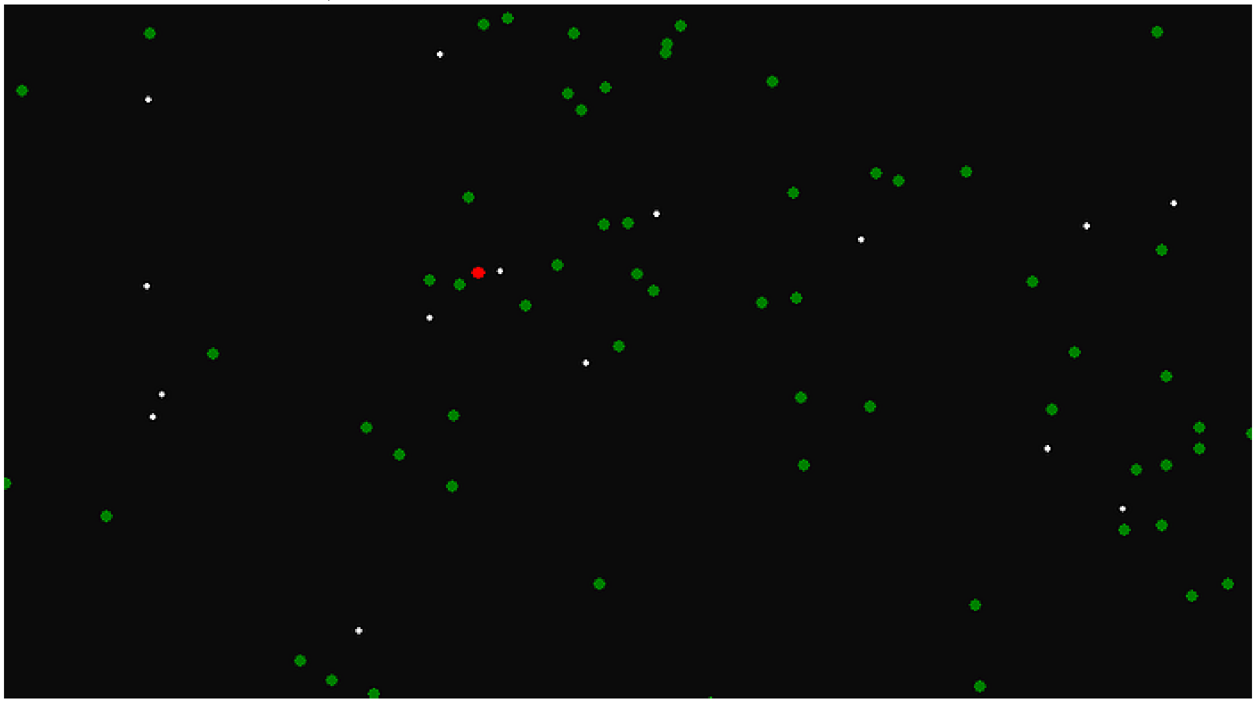
Finally, even predators have to **die** eventually. We say that they live for 2000 timesteps and are removed from the simulation if they are older than that.

```
# handle eaten and create new plant
plants = [p for p in plants if p.is_alive is True]
plants = plants + [Plant() for i in range(2)]

# handle eaten and create new preys
preys = [p for p in preys if p.is_alive is True]
```

```
for p in preys[:]:
    if p.energy > 5:
        p.energy = 0
        preys.append(Prey(x = p.x + random.randint(-20, 20), y = p.y +
random.randint(-20, 20)))

# handle old and create new predators
predators = [p for p in predators if p.age < 2000]

for p in predators[:]:
    if p.energy > 10:
        p.energy = 0
        predators.append(Predator(x = p.x + random.randint(-20, 20), y = p.y
+ random.randint(-20, 20)))
```

The red dot is a predator, the white dots are prey, and the green dots are plants.

Conclusions

This article has shown how a simple multi-agent predator-prey system can be programmed in Python. It covers just the basics, but it works. The agents move around and eat each other. Simple **population dynamics** can be observed, they match the findings of Lotka and Volterra.

It can't be used yet for solving scientific questions in biology. However, it is a good starting point for a more specialized simulation tailored to model a specific scenario.

Depending on your goals, the next possible steps could be:

- Improve the **visualization**. The small dots can be replaced by images loaded from disk.
- **Fine-tune** the agents' behavior so they reflect the behavior of actual individuals studied.

- Add more **forces** to the *social force model* and calibrate them using your own observations or values from literature. Without additional forces, for example, the agents tend to clump together after they follow the same target.
- **Measure and plot** certain values from the simulation that correspond to a scientific experiment, e.g., the predator-prey ratio.
- Add a **machine learning** element to the agents so they can learn and adapt.
- Also, from a software design perspective, **separating** the agent implementation from the visualization is strongly suggested.

Resources and References

The complete code for this article can be downloaded from GitHub: https://github.com/multi-agent-ai/examples

The author's youtube channel featuring this simulation and related work: https://www.youtube.com/@multi-agent-ai

[1] Andrea C. (2021) Prey and predators — a model for the dynamics of biological systems. https://towardsdatascience.com/prey-and-predators-a-model-for-the-dynamics-of-biological-systems-747b82d2ea9e

[2] Helbing, D., & Molnar, P. (1998). Social Force Model for Pedestrian Dynamics. *arXiv*. https://doi.org/10.1103/PhysRevE.51.4282