Hawks and Doves

In this project you will be simulating a small ecosystem. In this ecosystem birds of various species look for food. Sometimes they find it and eat it. Other times they find it, but so does another bird. The birds then have the choice to fight for the food or to be passive. Depending on their decisions, they could get the food or not, but they could also be injured in the fight. You will create different species of birds with varying behaviors and see how those species thrive (or not) in certain situations.

The scenarios you'll be investigating here are commonly talked about in a variety of fields. We'll be focusing on the programming, but there are lots of important ideas here, and the analysis of this sort of situation is of interest to biologists, economists, mathematicians, and others. Computer simulation is only one way to go about thinking about these issues, but it is a tool that can add help greatly with this analysis (and with more complicated scenarios of a similar type).

The model

We'll be simulating a very simple ecosystem, without most of the complexities of the real world. The only active animals in our world will be birds. Each bird has a number representing its health, which starts at 100. If a bird eats, its health goes up. If it is injured, its health goes down. Birds also lose a small amount of health each unit of time. (This represents their normal rate of burning calories – a bird that doesn't eat slowly gets less healthy over time.) If the health of a bird hits 0, it dies. If the health goes high – over 200 – then the bird reproduces. When it reproduces, its health goes down by 100, and a new bird of the same time is created.

Birds find food in one of two ways. Each time step, there are 10 units of food that are found by single birds on their own, who then get to eat the food without worry. There are also 50 units of food that are contested – one bird finds the food, but another approaches before they can eat it. In cases of contested food, each bird has two options – they can be passive or aggressive. Depending on what they choose, one of the following scenarios will occur:

- **Two passive birds**
  When both birds are passive, they will engage in displays meant to get the other bird to leave, and this will continue until one bird does so. The display costs a small amount of health for each bird, and it is random which of the birds ends up with the food.
- **One aggressive and one passive bird**
  If one bird is aggressive and the other is passive, the passive bird will flee and the aggressive bird will get the food.
- **Two aggressive birds**
  When both birds are aggressive, they will fight. One bird (chosen randomly) will win, and they will get the food. The loser instead is injured, reducing their health.

We will be modeling several types of birds. We'll begin with hawks and doves. Hawks will always be aggressive, while doves will always be passive. Once we have that simulation working correctly, we'll add other types of birds. Along the way we'll get a bunch of interesting results.

Programming note: The goal of this project is to show you object oriented programming used in a particular way. As a result, we will be very precise about exactly what classes there should be, what methods those classes should have, and what those methods should do. Follow the instructions carefully.

Experiment 1: Hawks and doves

Our first goal is to run the simulation with just hawks and doves. In order to do this, we'll need to write a lot of code. Your starter code file, birds.py, contains an outline of what we will need. It has the needed import statements, and it has a set of constants defined at the top. These constants determine the relative values of food, injuries, and displays, as well as how many of each type of bird we create at the start of the simulation. It also has the code for actually running the simulation, but that code won't work yet. In between those things, we'll need to add several classes, each with several methods, that will represent the world of our simulation and the birds that live within it. Follow the steps below to add these pieces.

1. Create a `World` class. This will represent the world as a whole. (We will only ever create one actual world object.) The initialization function should take no input. The class should only have one variable, a list of currently-existing birds, and that list should initially be set to the empty list.
2. Create a `Bird` class. We won't actually create any birds, but we'll have other classes that inherit from `Bird` and represent specific types of birds. It will be useful to define some things about birds here so that we don't need to repeat them many times later. The `Bird` initialization function should take a single input, a world object. The bird should then add itself to the list of birds that exist in that world. It should also keep an object variable storing the world that it is in, and it should have an object variable tracking its current health, which starts at 100.
3. Add an `eat` method to the `Bird` class. This method should take no inputs. When it is called, it should increase the bird's health by the amount specified by the `foodbenefit` constant.
4. Add an `injured` method to the `Bird` class. This method should take no inputs, and should reduce the bird's health by the amount specified by the `injurycost` constant.
5. Add a `display` method to the `Bird` class. This method should take no inputs and should reduce the bird's health by the amount specified by the `displaycost` constant.
6. Add a `die` method to the `Bird` class. This method should take no arguments and should remove the bird from its world's bird list.
7. Add an `update` method to the `Bird` class. This method takes no arguments and it represents one unit of time passing. It should reduce the bird's health by 1 (representing

calories being burned) and then should check whether the health is at or below 0. If so, the bird should die (which you can make happen using the `die` method). We now have a `Bird` class that has all the basic functionality.

8. We now add additionally functionality to the `World` class. First, create an `update` method in this class. It should take no inputs, and should simply call the `update` method for each bird in the bird list. This way have a single method we can call that makes time pass for everything in our simulation.
9. Add a `free_food` method to the `World` class. This represents the uncontested food that birds can find. It should take as input a single number. It should then award free food that many times. Each award of free food is done by picking a random bird from the bird list and having that bird eat. (Each choice of bird should be independent – it is possible a bird will get free food multiple times in a single `free_food` call if the bird is lucky.)
10. Now create `Dove` and `Hawk` classes, representing our two types of birds. These classes should inherit from `Bird`, so we won't need to add all that much to them. Each should have a class variable called `species` that is equal to a string that represents the species of that bird. The `Dove` class should have species set to `"Dove"` and the `Hawk` class should have species set to `"Hawk"`. There is no need to add initialization functions to either class – the initialization function in `Bird` will do just fine.
11. Add `update` methods to `Dove` and `Hawk`. These are modifications of the method for `Bird`. They should do the same things `Bird` does. (Don't repeat the code – use the function from `Bird`.) Then they should *also* reproduce if necessary. After the standard update (including the health reduction) these methods should check whether the dove/hawk has health of at least 200. If so, the health should be reduced by 100 and a new bird (of the same type) should be created (in the same world).
12. Add a `defend_choice` method to `Dove` and `Hawk`. This method takes no input and "asks" the bird whether it will defend itself if another bird finds it with food. Doves never will, so their function should always return `False`, while hawks should have a function that always returns `True`.
13. Finally, we need to add a method that represents a hawk/dove coming upon another bird with food. This method will be called `encounter`, and it takes as input another bird (the one that has already found the food). It should use `defend_choice` to find out if the other bird will defend itself, and then respond accordingly. A dove will run away if the other bird defends itself, having no effect on the dove, and letting the other bird eat. (Remember, you need to explicitly have the other bird eat.) If the other bird does not defend itself, both birds should display, then a random one of the two should eat the food. A hawk on the other hand will always eat if the other bird does not defend itself. If it does, the two fight with a random winner. The winner eats and the loser gets injured. Once this is done, your `Hawk` and `Dove` classes are complete.
14. We can now complete the `World` class. First we add a `conflict` method. This is the analogous function to `free_food`, and it creates encounters between birds over food. It should take as input a number and then have that many encounters occur. In each encounter, two (different) birds should be randomly selected. An encounter should then

occur where one of the birds finds the other. (Do this by calling an `encounter` method.) Like with `free_food`, the same bird can be chosen for multiple encounters.

15. Add a `status` method to world. This method should count the number of birds of each species that currently exist and print a summary of that information to the terminal.

We can now run our first experiments. Set `init_hawk` and `init_dove` to 50, and `init_defensive` and `init_evolving` to 0 (these are two types of birds we haven't made yet, so we don't want them in this experiment). Then set `injurycost`, `displaycost`, and `foodbenefit` to 11, 1, and 8 respectively. Run the file. The code provided at the bottom will run the experiment for 10,000 units of time, at each step providing free food 10 times and contested food 50 times. At the end, you should see a mix of hawks and doves. Try changing `injurycost` to 7. Now you should see that only hawks survive. Play around with different numbers. You should find that you can create outcomes that mix hawks and doves and outcomes that are all hawks, but no outcomes that are all doves.

Experiment 2: Defensive birds

Next we want to add a new type of bird. This type of bird is "defensive" – if it finds food first, it will fight to defend it from other birds, but it will not attack other birds to take their food. (It will, however, display in the way doves do.) It also of course creates additional defensive birds when it reproduces. You should create a `Defensive` class to represent this bird. Remember that `defend_choice` determines what a bird does when it has already found food and another bird appears, while `encounter` determines what it does when it finds another bird already with food. Remember to avoid repeating code whenever possible!

When you have such a class created, set `init_defensive` to 50 so that we now have three types of birds in our experiment. Now try several experiments. Set `injurycost` to 10, `displaycost` to 1 and `foodbenefit` to 12. You should see that only hawks survive. If you instead change `foodbenefit` to 8, you should see only defensive birds. Can you find settings that give a mix of defensive birds and hawks? Can you find settings where doves survive?

Experiment 3: Evolution

Now we make a bigger change. Instead of specifying exactly what type of birds we have and how they behave, we want to allow birds to behave in a variety of ways and let them evolve over generations. We'll get rid of doves and hawks and defensive birds. Instead we'll use a new class, `Evolving`, that can represent birds of various kinds. We'll let these new birds vary in two ways. One is that they will have different levels of aggression, choosing to fight more or less frequently. The other is that they'll vary by weight, with heavier birds winning fights more often but also burning more calories over time. Then, we'll let these birds evolve – children of a bird will have roughly the same (but not identical) attributes as the parent – and we can see how the birds change over time.

First, you'll need to write `defend_choice` and `encounter`. These methods need to account for the new way birds choose whether to fight. A bird should choose to fight with a probability equal to its aggressiveness (which will always be between 0 and 1). (Note that this is true whether encountering another bird or defending – we're dropping the distinction we made in `Defensive`.) `encounter` also needs to account for the new, more complicated method of fighting. We want heavier birds to win fights more often, so we will say that if the birds have weights w_1 and w_2, then the probability of bird 1 winning is $w_1/(w_1+w_2)$.

Programming tip: `random.random()` outputs a uniform random number between 0 and 1. You can use this to make things happen with a certain probability.)

Next you need to write `update` and `__init__`. These functions will be used to create two changes from our previous birds. The first is that heavier birds burn more calories. As a result, we want health to decrease by .4+.6w at each unit of time. (This is still 1 for the smallest birds, but it scales up as the birds get larger.)

The bigger change, however, will be to how reproduction works. It is no longer enough to just create another bird of the same type. We want the offspring of a successful bird to be similar to its parent, but with a little variability. (This is a rough approximation of the mutations that occur in nature.) You should modify the initialization function for `Evolving`. It should take two additional *optional* arguments. (Remember that you can put optional arguments in a function by using an equal sign and giving the arguments default values that they will take if inputs for those variables are not included.) If these arguments are missing (such as when creating the initial population of birds at the start of the experiment) then the weight and aggression should be chosen as uniform numbers from the allowed ranges (0 to 1 for aggression and 1 to 3 for weight). (Tip: `random.uniform(a,b)` returns a uniform random number between `a` and `b`.)

If, however, these arguments *are* given as the input, we should take them to be the aggression and weight of the new bird's parent. We want this new bird to have weight and aggression close to but not necessarily identical to that of its parent. In order to do that, the initialization function should set each variable equal to that of its parent plus a small amount of random noise. In particular, weight should be the parent's weight plus a uniform value between -0.1 and 0.1. Similarly, aggression should be the parent's aggression plus a uniform value between -0.05 and 0.05. However, we want these values to stay in the allowed ranges (0 to 1 for aggression and 1 to 3 for weight.) If the random variation would put a value above/below the allowed maximum/minimum, then instead set it to the maximum/minimum exactly.

The reproduction part of `update` needs to take advantage of this change, creating a new bird that has the right inputs to its initialization function. (The birds we create in the beginning for the initial population will be created without these optional arguments, giving completely random values.)

Finally, we need a way to show what the evolving birds are like at a given point in time. We can't just print counts of each type of bird anymore, so instead we'll create a simple scatter plot of the weights and aggressiveness of the birds. A simple function for producing scatter plots, `plot`, has been provided for you. It takes as input two lists. The first is the list of x coordinates of points being plotted, which will for us be weights. The second is a list of the corresponding y coordinates (in this case aggressiveness). The two lists must correspond in order – the nth x coordinate must correspond to the same point as the nth y coordinate. Add a method `evolvingPlot` to the `World` class that creates a scatter plot of all the current `Evolving`-class birds.

We can now run an experiment with these birds. At the bottom of the file, uncomment the line that makes a plot of evolving birds. Then set the initial bird counts so that the only birds we have are 150 evolving birds. Finally, set `injurycost` to 10, `displaycost` to 1 and `foodbenefit` to 8. When you run the file, you should see that our evolving birds naturally separate into two "species" clumps – one passive, light clump fulfilling the "dove" role, and one aggressive, heavier clump fulfilling the "hawk" role. Try varying these constants to see what is possible. You should find that some settings will result in two clumps (though one of these clumps can move around), while others result in a single clump.

What all of this means

The above description focused on the programming, since that's what this class is about. The phenomena that you have simulated, though, are part of a deep and interesting area of study. *Game theory* is the mathematical modeling and analysis of strategic interactions between rational actors. It is widely used in economics to model market interactions – 11 game theorists have now won Nobel prizes is economics. Political scientists use it to model any number of things, including international relations and domestic politics.

Modern game theory began as a result of groundbreaking research by John von Newmann and his 1944 book with Oskar Morgenstern, though some of the basic ideas had been seen earlier in works by Cournot and others. The most famous game theorist is probably John Nash, in part due to the movie *A Beautiful Mind*. Nash developed the definition of Nash equilibria, the standard solution concept for games, and proved that such equilibria exist in all games. For a more detailed overview of game theory, I recommend the Wikipedia article.

The game we simulated in Experiment 1, Hawk-Dove, is a special case of the game of Chicken, a standard example in economics. The Hawk-Dove version was introduced by John Maynard Smith and George Price in a 1973 paper, "The logic of animal conflict." Smith also introduced the idea of an "evolutionarily stable strategy," a solution concept for games specifically tailored to modeling evolution. Again, I recommend the Wikipedia article on the game to see more explanation.

Games can be (and usually are) analyzed without computer simulation. A large set of mathematical tools has been developed so that one can tell what the equilibrium of a game will be without need to simulate it. However, simulation can provide a concrete means to see how the game behaves. It can also be used to answer difficult questions when the theory becomes very difficult in complicated games. For example, see this paper by Bergstrom and Godfrey-Smith. This is an example of modern research in biology that uses computer simulation of the hawk-dove game.

The results of these models do really tell us something about how the world works. The mixed results of the hawk-dove simulation help explain why a variety of animals exist and behave in different ways. The fact that these mixed results disappear in certain cases, with hawks becoming dominant (but never the reverse dominance for doves) also tells us something about which animals might succeed in which sort of situation.

The strength of the defensive strategy also has wide-ranging implications. It has been used as an argument for why territoriality might develop among certain animals. It can even be used to explain why social norms or a sense of "ownership" might evolve. Rules that specify who should get what reduce conflict, and in some cases it can do so in a way that makes every individual benefit from following the rules, even if they are arbitrary. (Of course, there are many other potential explanations of these things as well.)

The simulation of evolving birds also of course has relevance to biology. You can see how the mixed equilibrium of hawks and doves can appear even if the initial animals do not fall into two separate species. It can even be used to model one way that speciation might occur, with a previous homogenous population separating into two distinct types.

Game theory is also used in a variety of fields in computer science. Algorithms research now includes a substantial effort to find efficient ways of analyzing games to find various sorts of equilibria. (This turns out to be a very difficult problem.) Game theory is also used to model interactions between multiple actors. This includes network protocols, peer-to-peer file sharing protocols, and any other situation where protocols are being written for multiple agents to interact. (The goal here is to make it so that self-interested participants can't deviate from the prescribed protocol in order to take more resources for themselves.) It also has a natural home in cryptography, which routinely studies the interaction of untrustworthy agents. I (Adam) have done research in this area of cryptography, and I would be happy to talk to you about it if you're interested.

Submitting

All you need to submit for this project is the main file containing all your work. Upload it to the homework submission site whenever you are ready.