Curves of Pursuit

If we have a Python object which can represent a moving creature, we can investigate what happens if that creature is programmed to relentlessly pursue another creature. The shapes created in such a scenario are known as curves of pursuit.

Constantly adjusting your aim:

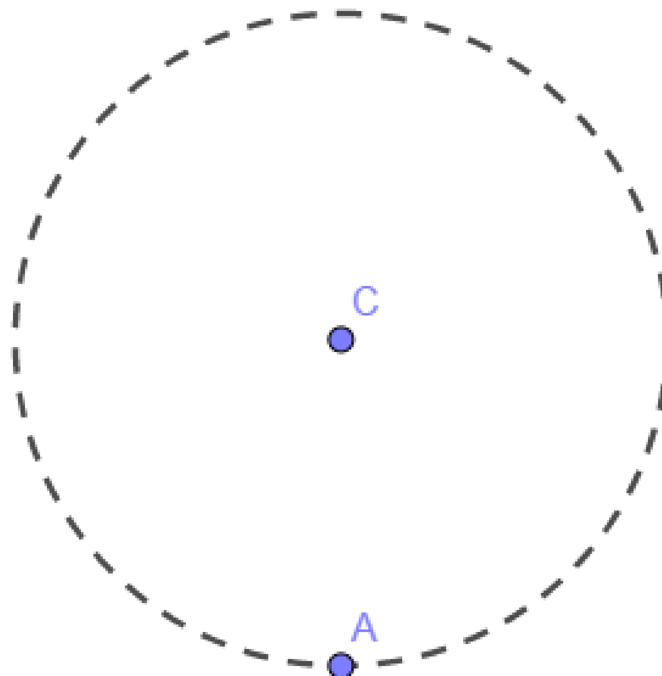
If a cheetah is chasing an antelope, rather than aiming for a point beyond them, to intercept their path, the cheetah runs straight towards the antelope. This makes sense: if you run towards a different point, the antelope could change direction to avoid you. Provided the cheetah is faster than the antelope, it should be able to catch it with this strategy, regardless of the direction the antelope takes. (In reality, a cheetah can only manage great speeds for short bursts, however).

Give it a go:

Imagine you are a cheetah, starting from point C, and you constantly adjust your direction of motion to ensure you always point towards the antelope A, while the antelope is moving at a constant speed along the dotted line.

Sketch below what you would expect the path of the cheetah to look like:

Did the cheetah catch the antelope? Draw some more possible curves for different speeds. Or see what would happen if the antelope ran in a circle (again, try to represent a range of possible scenarios for different speeds):



What if we could make turtles pretend to be cheetahs and antelopes…?

Coding Pursuit

Python (with Turtle) ›

python 3

The Cheetah

We are going to want a **Cheetah** class, which will be capable of chasing (moving in the direction of) an instance of the **Antelope** class. Provided the antelope has **x** and **y** attributes (its position), we can use that to define the targeting behaviour:

```
1   from math import *
2   import turtle, random
3
4 ⊞ def create_turtle(x, y, shape, color):···
16
17  class Cheetah:
18      def __init__(self, x, y, v, target=None):
19          self.x, self.y, self.v = x, y, v
20          self.target = target
21          self.t = create_turtle(x, y, "arrow", "red")
```

When I create a **Cheetah** object, I want to be able to define its initial position, its velocity and its target (which will be an **Antelope** when that's made).

To avoid cluttering up my code with the lines necessary to make a turtle object, I've delegated that to an entirely separate function (yet to be written), which will generate and return a turtle object. This code (lines 4 to 15) has been minimised above for clarity.

```
23      def move(self, dt=0.1):
24          if self.target != None:
25              X, Y = self.target.x, self.target.y
26              self.t.seth(self.t.towards(X, Y))
27              self.t.fd(self.v * dt)
28              self.x, self.y = self.t.xcor(), self.t.ycor()
29              x, y = self.x, self.y
30              if (x - X)**2 + (y - Y)**2 < 5**2:
31                  self.target = None
```

When the cheetah moves, it needs to turn towards its target, then move forwards in that direction according to its velocity. I also keep track of the **x** and **y** coordinates by extracting the turtle's coordinates at each move.

This code will only run if the cheetah has a target, and the cheetah will stop once it reaches its target. We check by finding the distance between the cheetah and its target.

Note: when you are writing your code, try to think about the possible modifications you might want to try later. What if we want to create a predator which wanders randomly around the screen until it comes within range of its prey? Could we make a string of creatures, each of which chases the one in front?

```
4   def create_turtle(x, y, shape, color):
5       t = turtle.Turtle()
6       t.ht()
7       t.speed(0)
8       t.tracer(15)
9       t.pu()
10      t.goto(x, y)
11      t.shape(shape)
12      t.color(color)
13      t.st()
14      t.pd()
15      return t
```

The **create_turtle** helper function is defined separately from the classes, because I'll be making use of the same code for both the **Cheetah** and **Antelope** functions.

This code creates a turtle, hides it, sets the speed and refresh rate, lifts the pen, goes to the given position, sets the shape and color, reveals the turtle and lowers the pen. It then returns the turtle object.

The Antelope

This class shares some features, but the motion will be fundamentally different: rather than being updated dynamically based on the motion of another creature, each antelope will follow a predefined path. The best way to define this is *parametrically*. That is, we will have a function which can provide an x-coordinate for any given value of t (time), and another function which will provide a y-coordinate for any value of t. These functions (to be written later), can be passed to the **Antelope** class right at the start like anything else.

```
33  class Antelope:
34      def __init__(self, fx, fy):
35          self.time = 0
36          self.fx, self.fy = fx, fy
37          self.x = self.fx(0)
38          self.y = self.fy(0)
39          self.t = create_turtle(self.x, self.y, "turtle", "blue")
40
41      def move(self, dt=0.1):
42          self.time += dt
43          self.x = self.fx(self.time)
44          self.y = self.fy(self.time)
45          self.t.seth(self.t.towards(self.x, self.y))
46          self.t.goto(self.x, self.y)
```

While it may not be obvious what **fx** and **fy** represent from here, we know they are functions.

The **create_turtle** helper function we made before is used here again.

The **move** function updates **self.time** (the time for which the antelope has been moving). The new position is generated using **fx** and **fy**, then the heading is updated and the antelope moves to its required position.

Parametric Functions

The functions we make can be as simple or complex as we like. As long as we can produce a value for an input, it'll work. Provided we use nice continuous functions, the antelope will move in a plausible fashion. One slight drawback is that, depending on the complexity of your functions, it may not be obvious what the speed of the antelope might be (and may even change during different parts of the motion). Here's some examples:

```
48  def fx(t):
49      return -200 + 2 * t
50
51  def fy(t):
52      return -100
```

The antelope will move to the right at a rate of two pixels per unit time, from the point $(-200, -100)$.

```
48  def fx(t):
49      return 100 * cos(0.1 * t)
50
51  def fy(t):
52      return 100 * sin(0.1 * t)
```

The antelope will move in a circle of radius 100 pixels. 0.1 represents the frequency of oscillation (or angular speed). The actual speed will be equal to the radius multiplied by the angular speed (here, 10).

```
48  def fx(t):
49      return t - 200
50
51  def fy(t):
52      return 200 * sin(0.05 * t) * exp(-0.01 * t)
```

The antelope moves right at a steady rate, but combines oscillation with exponential decay vertically (damped oscillation). Speed goes from 10 towards 1.

The Chase

The only thing left to do is create our antelope and our cheetah, and let them run.

```
54    ant = Antelope(fx, fy)
55    che = Cheetah(-250, 250, 5, target=ant)
56
57    while True:
58        ant.move()
59        che.move()
```

That's it! All the hard work has already been done by writing the classes and helper functions. All that remains is to create our animals (ant and che), and to move them repeatedly.

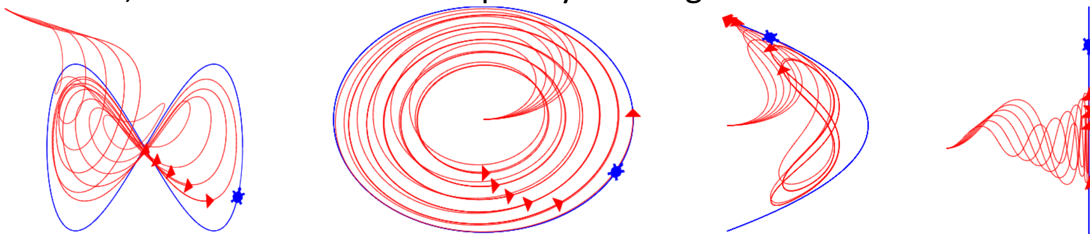
Experiment with different starting positions, different speeds or different functions for ant.

```
54    ant = Antelope(fx, fy)
55    cheetahs = []
56    for i in range(5):
57        cheetah = Cheetah(-250, 250, i, target=ant)
58        cheetahs.append(cheetah)
59
60    while True:
61        ant.move()
62        for c in cheetahs:
63            c.move()
```

Since classes have made it really straightforward to generate new objects, there's nothing stopping us making a whole herd (pack? coalition, apparently) of cheetahs. Our antelope won't know what hit it.

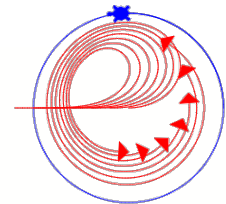
*Tip: we gave the **move** function a default time period of 0.1. If motion is progressing too slowly, you can set something different by changing **ant.move()** to **ant.move(0.4)**, etc.*

Experiment with the parametric functions **fx** and **fy** to generate different paths for your antelope to follow. Set your coalition of cheetahs to have different velocities, or different starting positions, and see what kinds of path you can generate. Here's a few of mine:



What's next?

A nice extension to this is follow-my-leader. With a bit of minor modification, we can make our cheetahs chase each other. Make a list where each new cheetah targets the one before (and also starts some way behind, and is slightly slower):



```
54    ant = Antelope(fx, fy)
55    cheetahs = []
56    cheetahs.append(Cheetah(0, 0, 8, target=ant))
57    for i in range(6):
58        new_cheetah = Cheetah(-20 -i * 20, 0, cheetahs[-1].v * 0.9, target=cheetahs[-1])
59        cheetahs.append(new_cheetah)
```

Or get rid of the antelope entirely, and see what happens if you make cheetah 1 chase cheetah 2 while cheetah 2 chases cheetah 3, and cheetah 3 chases cheetah 1…

```
54    n = 6
55    cheetahs = []
56    for i in range(n):
57        x, y = 100 * cos(2 * pi / n * i), 100 * sin(2 * pi / n * i)
58        cheetahs.append(Cheetah(x, y, 5))
59    for i in range(n):
60        cheetahs[i].target = cheetahs[(i + 1) % n]
```