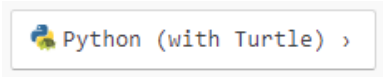
Contagion

While a computer simulation can't accurately encompass all the complexity behind the spread of a virus in a human population, in the same way that random events become startlingly predictable with a large enough sample size, even with some broad assumptions we can model the spread of a contagious disease.

The main phases within an infection are:

>Incubation period: hosts are infectious, but asymptomatic (display no visible signs of the disease)
>Infection period: hosts are visibly sick, to a more or less extreme degree. The body is actively fighting the bug.
>Recovery period: hosts have successfully overcome the sickness, but are still potentially contagious to others.

Simulating Contagion with Python



Program overview:
Before diving into creating creatures which manifest a contagious disease, it can be helpful to consider the overall structure of the program. From there, it will become clearer what objects we need to create, and exactly what their attributes and methods should be.

Ask yourself the following initial questions:

- How should the creatures move? What will determine their direction and speed?

- How will we see on screen which creatures are in which phases of the infection?

- What should happen when a creature encounters another creature? Consider each of the various cases: Will a sick creature definitely infect a healthy one if they meet? Should the creatures behave differently when they are sick (eg not move?)

- How should other creatures react to sick creatures?

The previous code sets up what it means to be a bug, and determines how they will behave in different situations. To see how this plays out, we also need the correct code to build our bug farm and make it work the way we want:

```
56
57  num_bugs = 5
58  bugs = [Bug() for i in range(num_bugs)]
59
60  while True:
61    for bug in bugs:
62      if not bug.alive:
63        bugs.remove(bug)
64      else:
65        bug.move()
66        bug.try_to_reproduce()
67        if bug.reproducing:
68          bugs.append(Bug(bug.t.xcor(), bug.t.ycor()))
69          bug.reproducing = False
70        for other in bugs:
71          if bug != other and bug.close_to(other):
72            bug.try_to_eat(other)
73            break
```

For a more involved script to modify and adapt: https://repl.it/@thechalkface2/Bug-Farm-A

The most efficient way to construct a population of bugs is using list comprehensions: **bugs** is a list which will be populated by **num_bugs** bugs.

The **while True** loop ensures that this simulation continues till we forcibly stop the program. Within this loop, each bug is looked at in turn. Firstly, if it is no longer alive, it will be removed from the list (saving memory so we don't continue to animate dead bugs). For the living bugs, they move, try to reproduce (and if they were successful, this is where we create a new bug, with the parent's position), and then the bug looks at all other bugs, and if it finds one nearby, tries to eat it.

If you run your script, you will almost certainly find a few of the more normal bugs – the obvious things that stop your code from working completely. This may be that you have used **self.x** rather than **self.t.xcor()** (because to access the position of our bug we are relying on the **xcor()** method of its personal turtle object).

You will probably also find, especially as you modify various values or try to add functionality, that – while the script executes just fine – the turtles don't behave the way you expect. Maybe you build in a 'follow-me' function, but you find two turtles start following each other, or perhaps parents eat their offspring, and you'd like to simulate a less barbaric bug farm. By tracing your code line by line, you can usually identify where such behaviours originate and alter it.