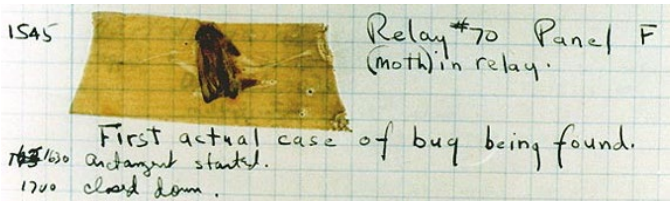
Bug Farm

It's nice to do simulations with large populations, and just extract the relevant information at the very end, but there is a danger that, by not observing directly the steps in between, we can miss out on insights that might otherwise jump out at us. This is especially true when trying to debug a program (identify why a particular error or unexpected behaviour has occurred).

A bug is:

>An error, flaw or fault in a program that causes incorrect results or unexpected behaviour.



>A small creature, usually an insect or microbe.

The basics of debugging:

Not every error in Python produces an error report, but – especially when unfamiliar with the syntax or nature of the language – these error reports are pretty common. Maybe you missed out a **self.** when referring to the attribute of an object inside a class method. Maybe you just used **=** instead of **==** when comparing two values. In either case, once you know how to read the Python error report, these are fairly quick issues to fix:

```
2   pets = ["dog", "cat", "fish"]
3   print(pets["dog"])
4
```

```
Traceback (most recent call last):
  File "main.py", line 3, in <module>
    print(pets["dog"])
TypeError: list indices must be integers or slices, not str
```

The error report tells you which line to look at, and gives suggestions for what went wrong. In this case, it was expecting a list index or slice, but we gave it an element instead.

Easily fixed:

```
2   pets = ["dog", "cat", "fish"]
3   print(pets[0])
4   # or, alternatively:
5   print(pets[0:2])
```

```
dog
['dog', 'cat']
>
```

Sneaky cases:

The real problems occur when an error is more insidious – it's sneakier, because it doesn't throw an error report. This may be because the error is part of the code that only gets used once in a while, so you'll hardly ever see it. It may also be because what's happening is perfectly logical, and the code is being accurately processed, but what it's doing is not what you thought you'd asked it to do. The code below seems to be saying that 45 is greater than 123, but because of the data type (strings), the > operator doesn't behave the way we expect.

```
2   nums = ["123", "45", "678"]
3
4   if nums[1] > nums[0]:
5     print(nums[1])
```

```
45
>
```

It's (roughly speaking) checking to see if the text strings are in alphabetical order.

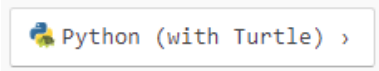
Finding sneaky bugs:

If a car stops working, the first thing a mechanic does is try to replicate the problem ("it makes a weird noise in second gear? Let's have a listen"), and the second thing is to open it up and take a look. There are too many things to look at for complex machinery (or a long piece of code), so narrowing down when an error or unexpected result occurs can help us identify likely trouble spots. The coding equivalent to 'pop the hood' is adding **print** statements in likely problem areas to give you the inside information you need.

But it's more fun if we can actually *see* the code in action…

Making Bugs

Step 0: ***Figure out what you want, and how we can describe it using class attributes and methods.***



I want to generate a population of creatures which will move about the screen independently, possibly following a random path or maybe pursuing a target. The creatures may have the capacity to interact in various ways: maybe when two creatures meet, they fight, or trade, or mate. I will probably need some way of ensuring they don't wander off the screen, and they may need attributes such as health (perhaps they die if their health runs out) or fertility (so that they can only mate once they mature, and after reproducing may need to wait before being able to again. In order to interact, it would be helpful to have a method for determining if another bug is 'close to' them, for a given value of 'close'.

You may have all sorts of ideas for how you want your own population of bugs to behave, but I'm going to illustrate a few simple ideas here which you can build on, adapt, or use the ideas behind to develop your own.

Step 1: Design your bugs…

Answer the following questions as clearly as you can, thinking carefully about how you might use Python to implement the behaviour you want:

- How should my bugs move? What will determine their direction and speed?

- Will my bugs have a lifespan, and if so, how will I keep track of dead or alive bugs?

- What should a bug do when it encounters another bug? And how will I know it has?

- Will my bugs be capable of reproduction? If so, will it require just one bug, or should my bugs have genders? How would I implement this in code?

```
1   import turtle, random
2
3   class Bug:
4     def __init__(self, x=0, y=0):
5       self.alive = True
6       self.reproducing = False
7       self.v = random.gauss(6, 2)
8       self.health = 255
9       self.t = self.create_turtle(x, y)
10
11    def create_turtle(self, x, y):
12      t = turtle.Turtle()
13      t.shape("turtle")
14      t.color(255 - self.health, 0, self.health)
15      t.speed(0)
16      t.tracer(5)
17      t.penup() # or t.pu()
18      t.hideturtle() # or t.ht()
19      t.goto(x, y)
20      t.setheading(random.randint(0, 359)) # or t.seth(
21      t.showturtle() # or t.st()
22      return t
23
24    def move(self):
25      if self.t.xcor()**2 + self.t.ycor()**2 > 200**2:
26        self.t.seth(self.t.towards(0, 0))
27      else:
28        if random.random() < 0.5:
29          self.t.left(random.randint(0, 10))
30        else:
31          self.t.right(random.randint(0, 10))
32      dist = random.uniform(0, self.v)
33      self.t.fd(dist)
34      self.health -= 1
35      self.t.color(255 - self.health, 0, self.health)
36      if self.health <= 0:
37        self.alive = False
38        self.t.ht()
39
40    def close_to(self, other, distance=20):
41      x, y = self.t.xcor(), self.t.ycor()
42      X, Y = other.t.xcor(), other.t.ycor()
43      return (x - X)**2 + (y - Y)**2 < distance**2
44
45    def try_to_reproduce(self, p=0.005):
46      if self.health >= 50:
47        if random.random() < p:
48          self.reproducing = True
49
50    def try_to_eat(self, other):
51      if other.health < 100 and random.random() < 0.5:
52        self.health += other.health // 2
53        self.health = min(self.health, 255)
54        other.alive = False
55        other.t.ht()
```

(We import the **random** module first).

My bugs will all spawn at the centre of the screen (my default x and y values are 0 and 0), and will be represented by a turtle object, allowing me to simplify the process of moving and displaying them. If a bug is produced through reproduction, I can set its x and y coordinates based on its parent's position.

I have separated out the process of building a Turtle object (which is an object within our Bug object).

When told to move, the bugs will first check to see if they are more than 200 pixels from the centre, and if so, will reorient themselves so they will be moving back towards it.
The distance they move has been set with a partially random element, but the maximum is limited by their top speed.
Each time they move, health drops by 1, and if they lose it all, they die.

Quite a few methods will depend on knowing the distance of a bug from another bug, so I've made the 'helper function', **close_to** to enable me to reuse this functionality.

If a bug tries to reproduce, an element of randomness determines how likely this is to happen. By increasing p, we increase the birth-rate.

If a bug tries to eat another, it may or may not be successful, but the other bug is only at risk if it is low on health.

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.