Kill All Mutants!

(Intro to Mutation Testing)

by Dave Aronson

T.Rex-2023@Codosaur.us

Codosaur.us

@davearonson

(Blank slide so I can flip to a new one to start my timer, ignore this.)

Kill All Mutants!

(Intro to Mutation Testing)

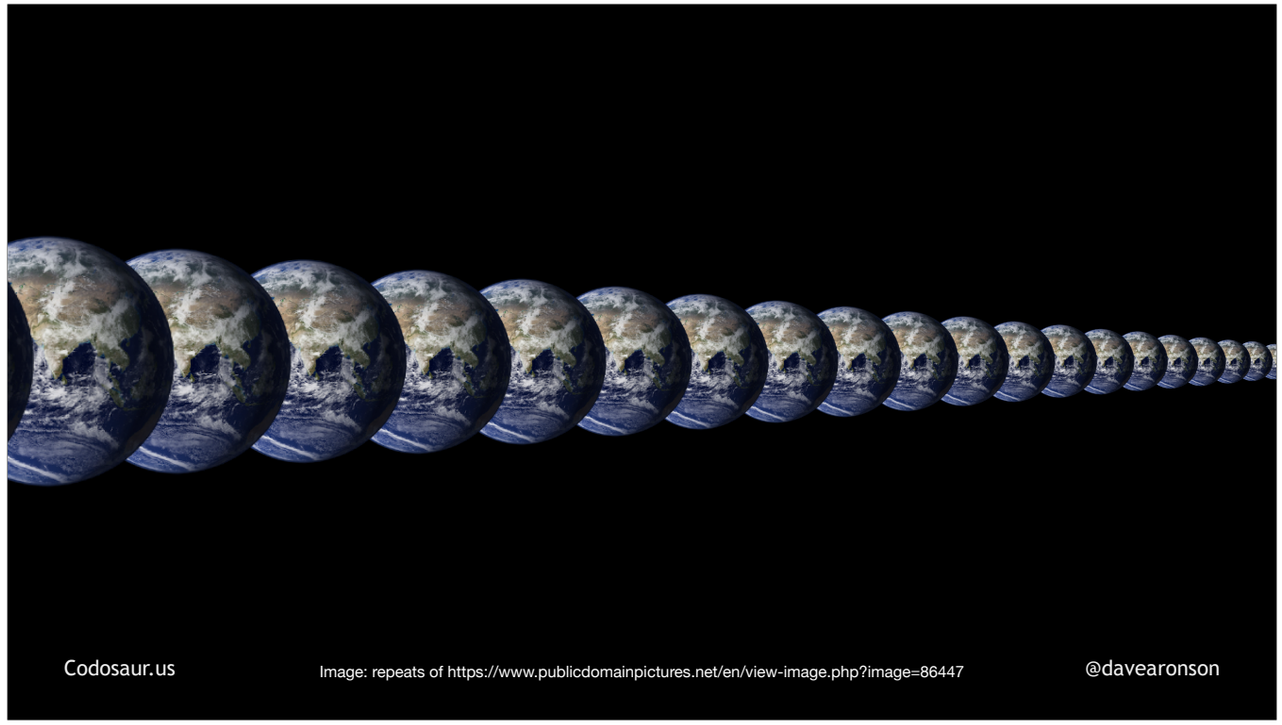
by Dave Aronson

T.Rex-2023@Codosaur.us

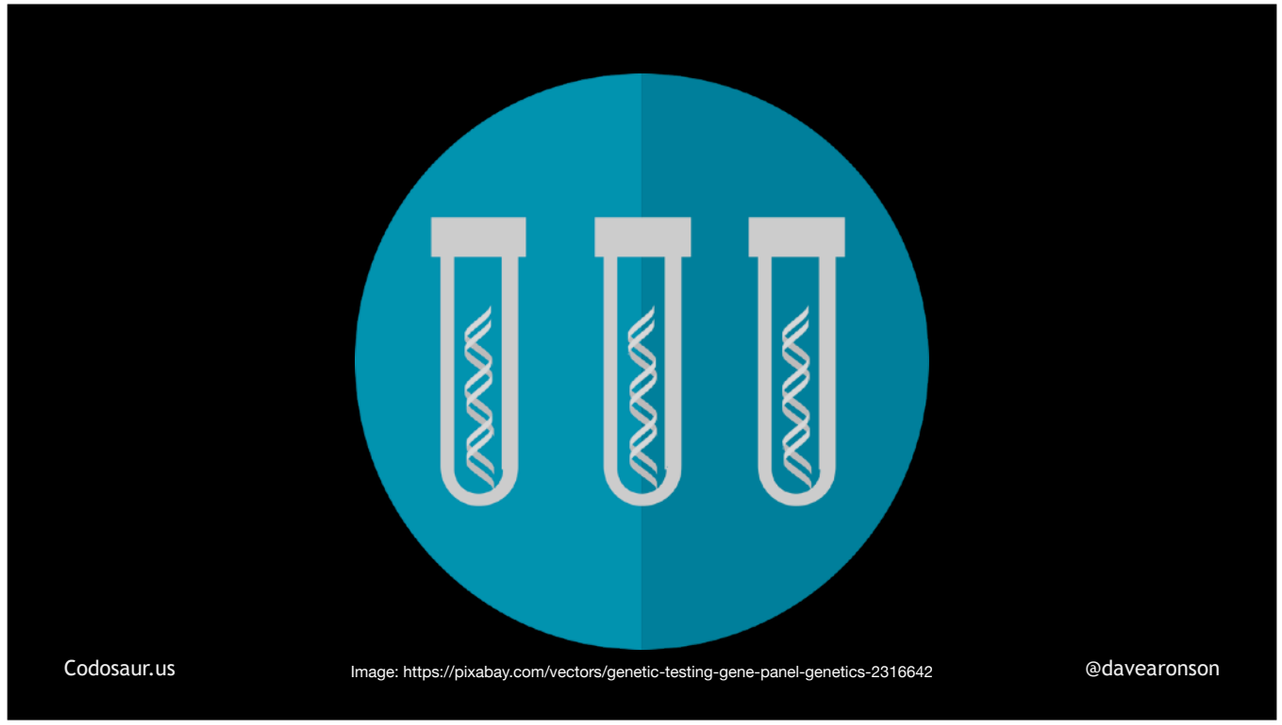
Codosaur.us

@davearonson

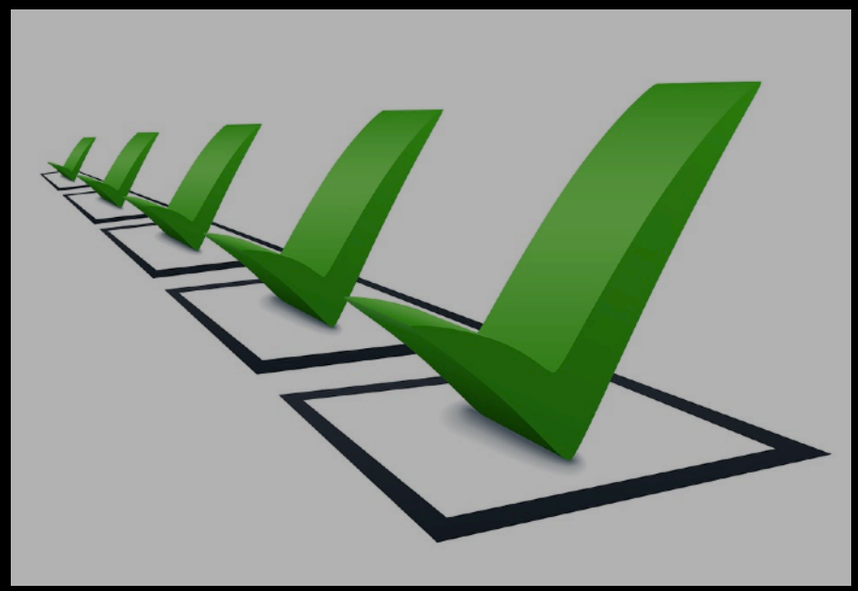
Hello, Salt Lake City! I'm Dave Aronson, the T. Rex of Codosaurus, and I flew over here on my pet pterodactyl to teach you to *kill mutants!* (PAUSE!) So, what on . . .



. . . Infinite Earths, makes . . .



. . . mutation testing different from all the *other* software testing techniques? The main difference is that most others are about . . .

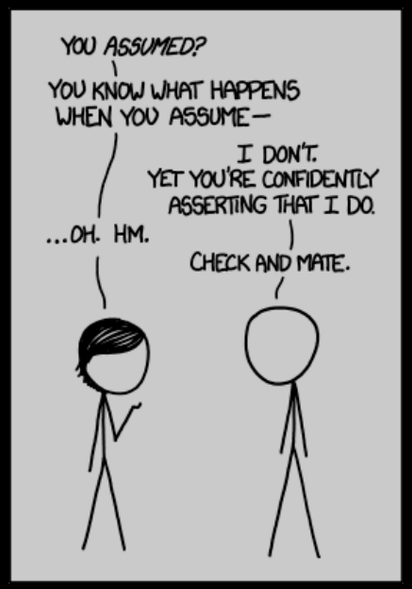


Codosaur.us

Image: https://pixabay.com/illustrations/tick-green-tick-correct-642162/

@davearonson

. . . checking whether our code is correct. But mutation testing . . .

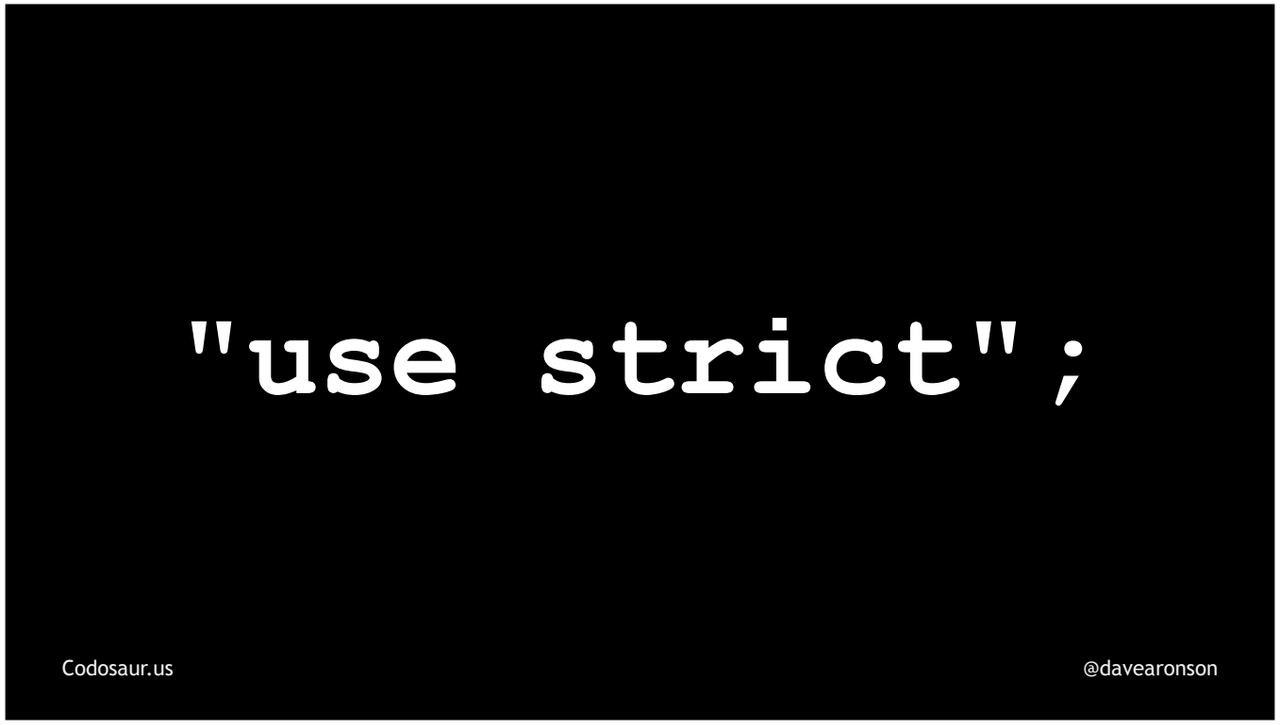


Codosaur.us

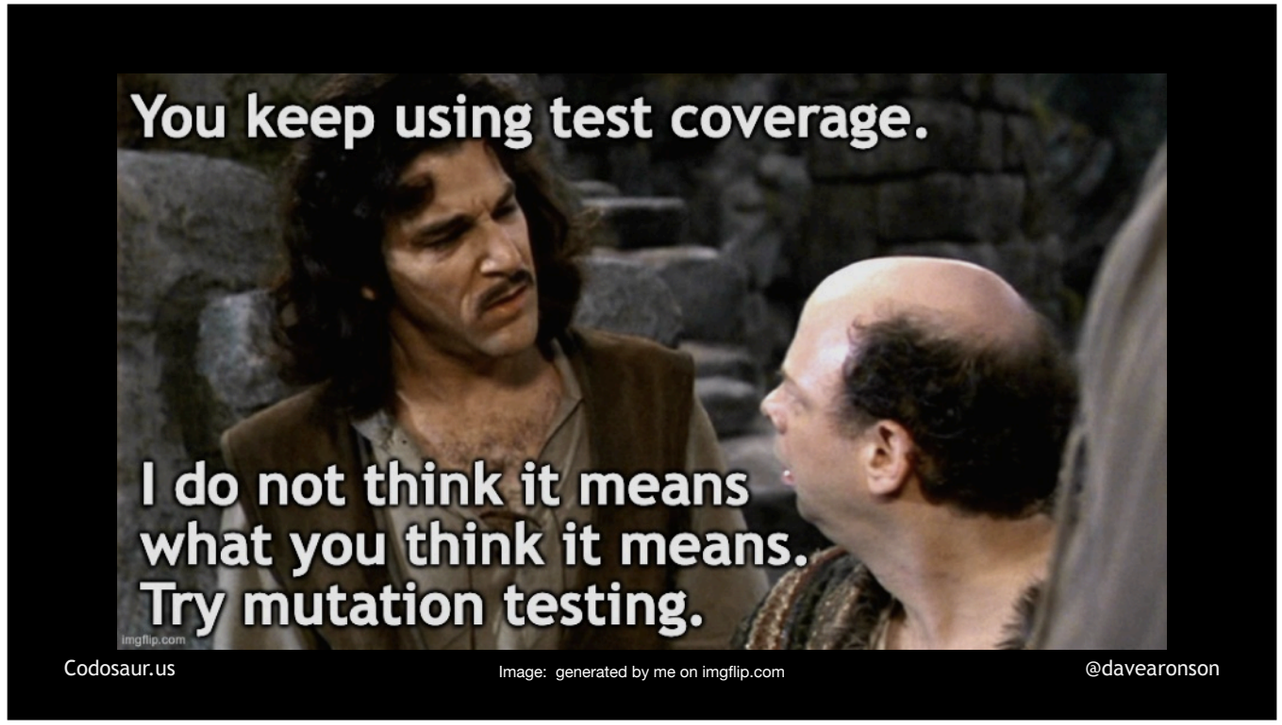
Image: https://xkcd.com/1339/

@davearonson

. . . *assumes* that our code is correct, at least in the sense of passing its tests. Instead, mutation testing checks for two *other* qualities. In a typical codebase, I think the more *important* one is that our test suite is . . .



. . . *strict*. Now you may be thinking, “Isn’t that what test coverage is for?”



The *only* thing that test coverage tells us is that at least one test *ran* . . .

```
class Conway:
  ALIVE = "*"
  DEAD  = " "


  @classmethod
  def next_state(cls, cur_state, neighbors):
    if cur_state == cls.ALIVE:
      r = cls.ALIVE if neighbors in [2,3] else cls.DEAD
    else:
      r = cls.ALIVE if neighbors == 3 else cls.DEAD
    return r

  def another_func:
    # whatever
```

Codosaur.us @davearonson

. . . the code it claims is “covered”, and *none* ran the rest of it. It tells us NOTHING about whether the *correctness* of the code *made any difference* to whether a test passed or failed — and isn’t that what we really *mean* by “tested”?

This is where mutation testing comes in. To check that our test suite is *strict*, a mutation testing tool will try to . . .

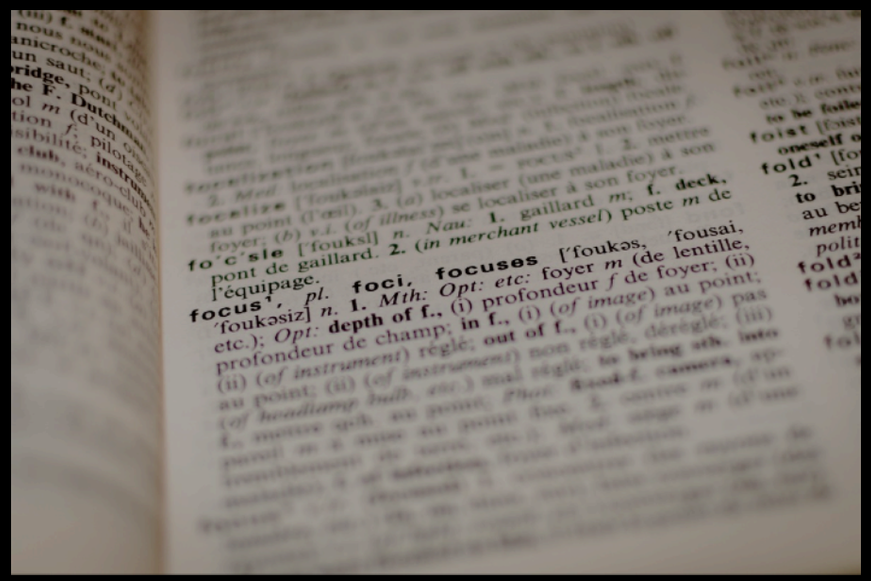
Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Mind_the_gap_2.JPG

@davearonson

. . . find the gaps in our test suite, that let our code get away with unwanted behavior. Once we find gaps, we can close them by either adding tests or improving existing tests. Lack of strictness comes mainly from *lack* of tests, or poorly *written* tests.

The other thing mutation testing checks is that our code is . . .



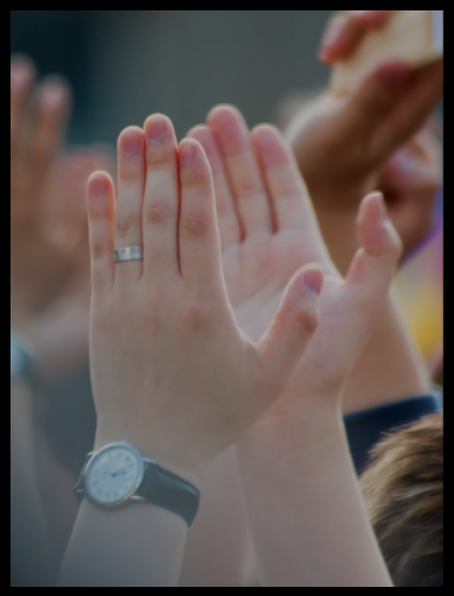
Codosaur.us

Image: https://pxhere.com/en/photo/825760

@davearonson

. . . *meaningful*, so that any semantic change to the code, will produce a noticeable change in its behavior. Lack of *meaning* comes mainly from code being unreachable, redundant, or otherwise just not having any real effect. Once we find "meaningless" code, we can figure out *why* it's meaningless, then make it meaningful, if that fits our intent, but the usual fix is just to remove it.

Mutation testing . . .



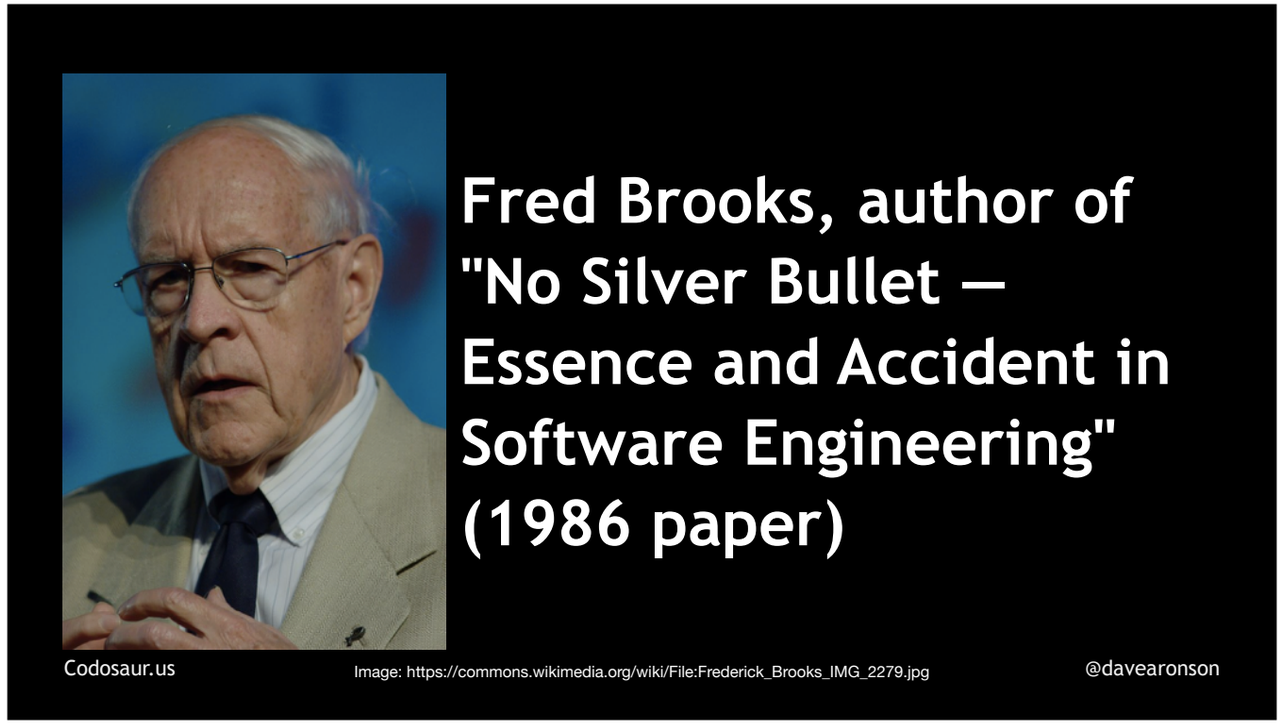
Codosaur.us

Image: https://www.flickr.com/photos/garryknight/2565937494

@davearonson

. . . puts these two together, by checking that every change to the code, that the tool knows how to do, does indeed make a noticeable change to its behavior, *and* that the test suite is indeed strict enough that at least one test will notice that change, and fail.

That's the positive side, but there are some drawbacks. As . . .



. . . Fred Brooks told us back in 1986, there's no . . .

Codosaur.us

Image: https://www.flickr.com/photos/sdasmarchives/4590226412

@davearonson

. . . silver bullet! Besides, those are for killing . . .

. . . werewolves, not mutants!

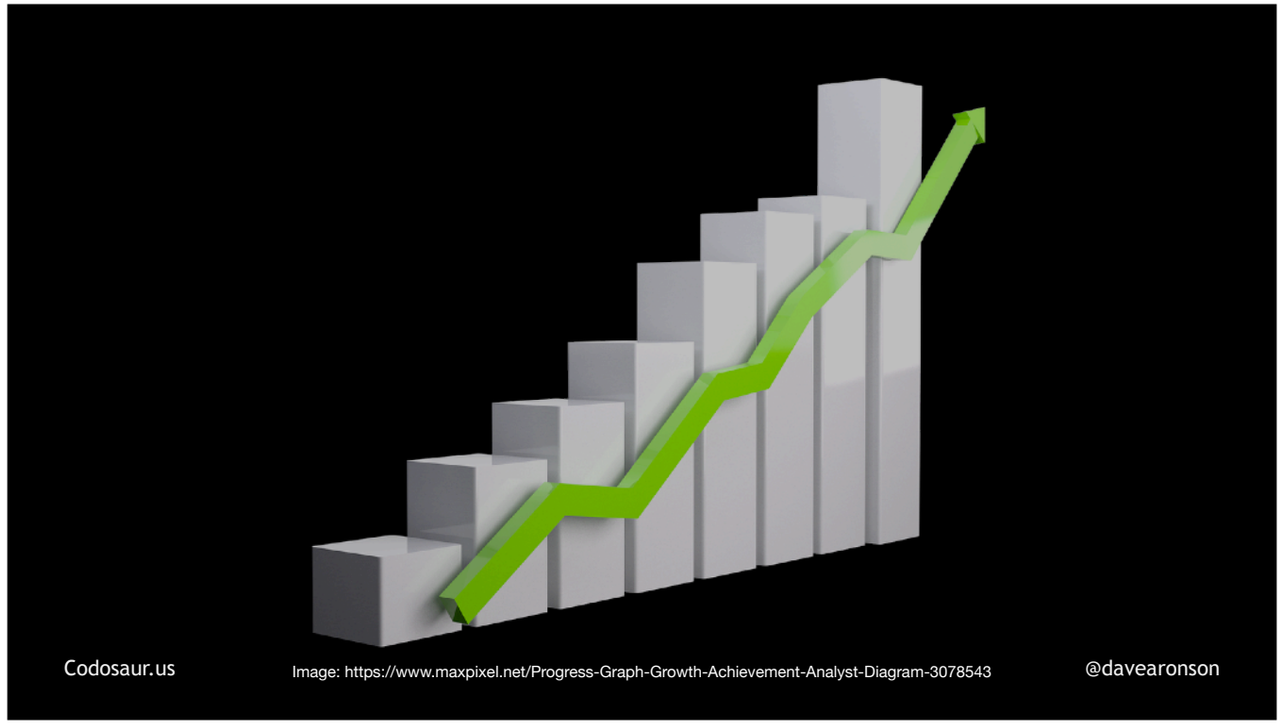
The first drawback is that it's rather . . .

Codosaur.us

Image: https://www.jtfb.southcom.mil/Media/Photos/igphoto/2000888525/

@davearonson

. . . hard labor for the CPU, and therefore usually ra-ther sloooow. We certainly won't want to mutation-test our whole codebase on every save! Maybe over a lunch break for a smallish system, or a weekend for a large one. Fortunately, most tools let us just check specific functions, classes, files, and so on. Also, they usually include some kind of . . .



. . . incremental mode, so that we can test only the changes since the last mutation test, or the last git commit, or the main branch, or some such difference. With such filtering, maybe we can test the changes on each save, or at least over a much shorter break.

Another drawback is that it’s often . . .

. . . not at all clear what to do about the results! It tells us that some particular change to the code made no difference to the test results, but what does *that* even mean? It takes a lot of interpretation to figure out what a mutant is trying to tell us. Their accent is verrah strayinge, and they're almost as incoherent as . . .

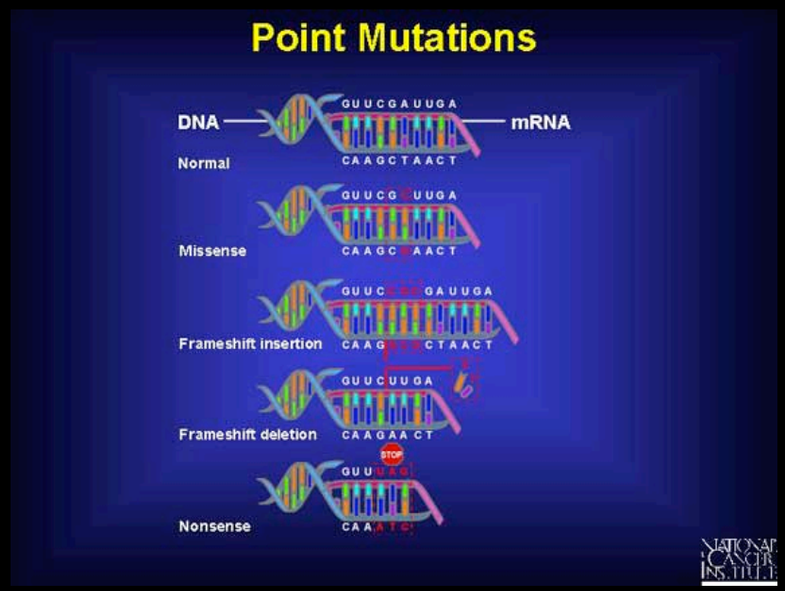
. . . zombies, but with a much bigger vocabulary, so they’re not always on about braaaaaaains. They're *usually* trying to tell us that our code is meaningless, or our tests are lax, or both, but it can be very hard to figure out exactly *how!* Even worse, sometimes it's a . . .

. . . false alarm, because the mutation didn't make a test fail, but it didn't make any difference in the first place. It can still take quite a lot of time and effort to figure *that* out.

And even if a mutation *does* make a difference, most programs have quite a lot of code that we just . . .

. . . *shouldn't bother* to test, like debugging traces! Fortunately, most tools have ways to tell them "don't bother mutating this line", or even this whole function, class, file, or whatever . . . but that's usually with comments, which can clutter up the code, and make it less readable.

Now that we've seen the pros and cons, how does mutation testing work, unlike this guy? It . . .

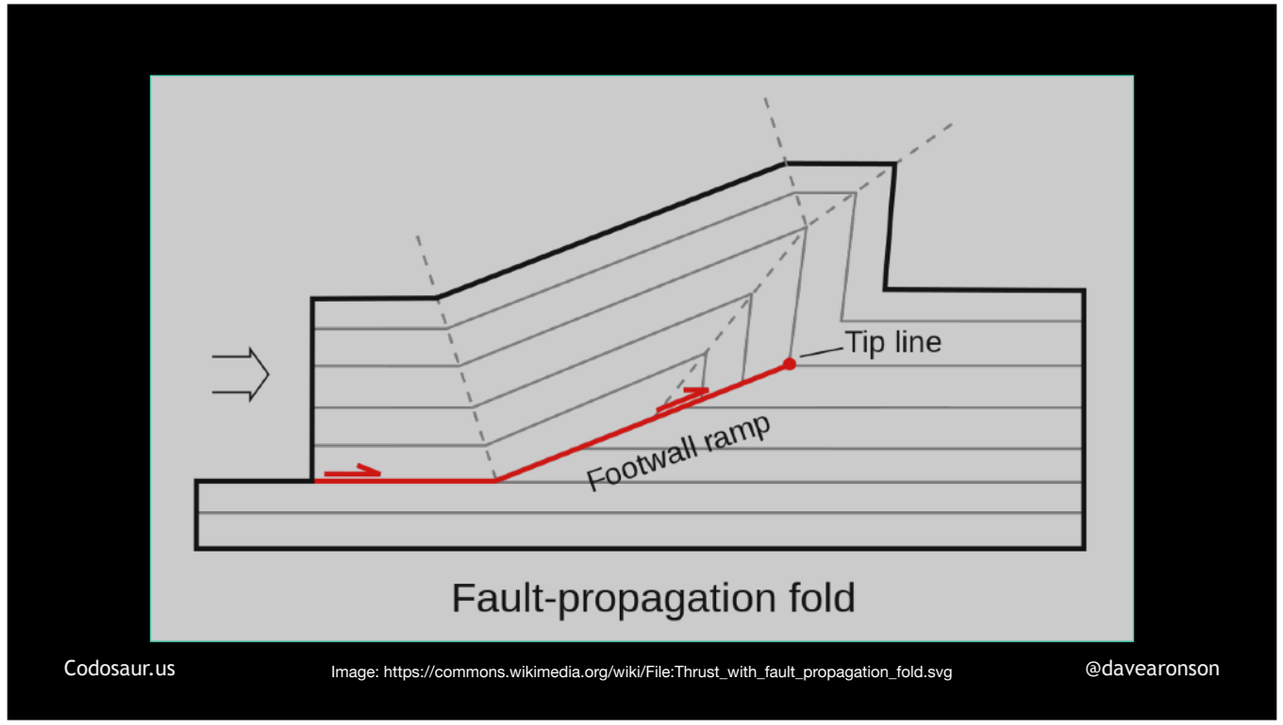


Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Thrust_with_fault_propagation_fold.svg

@davearonson

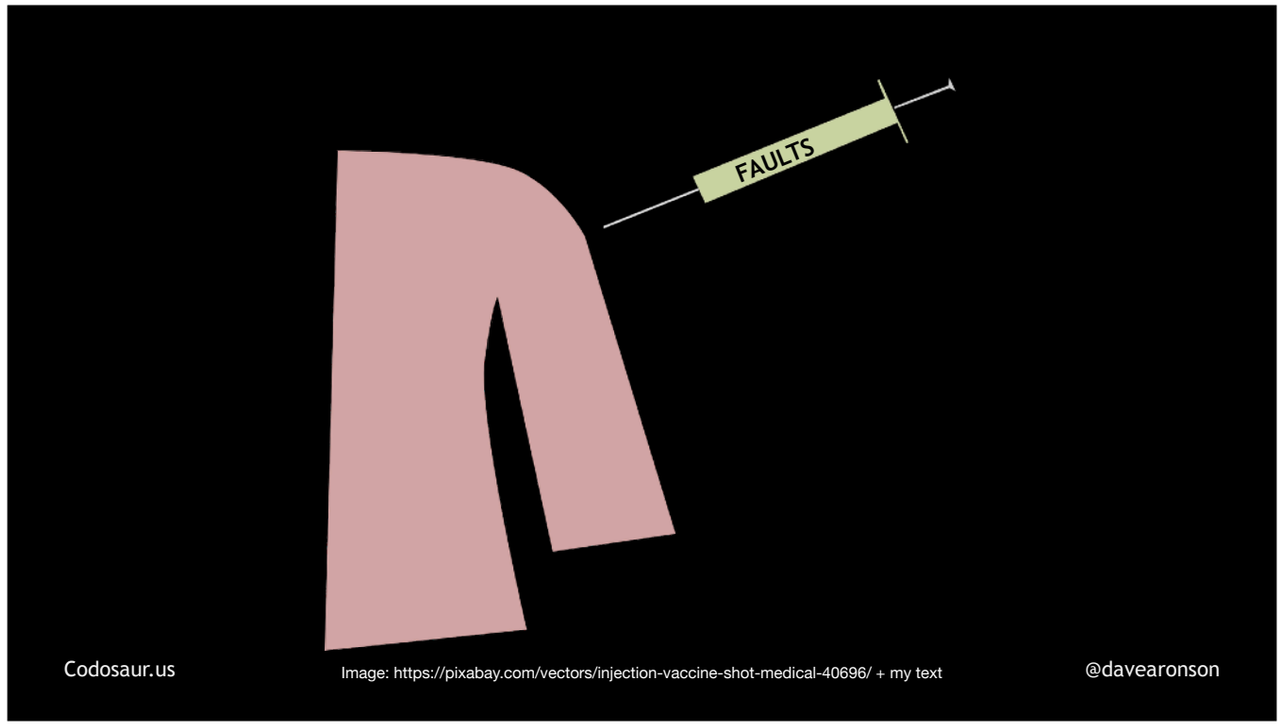
. . . *mutates* copies of our code, hence the name. It does this to create test failures, also called . . .



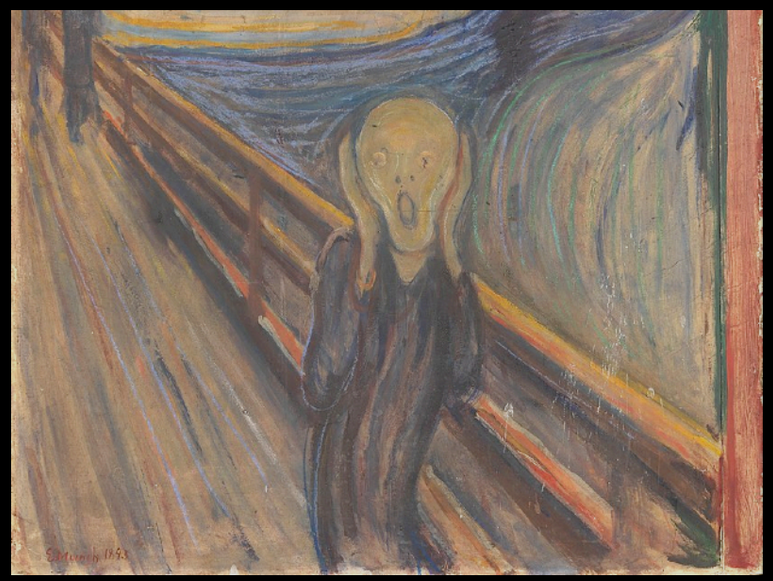
. . . faults. So, mutation testing is a *fault-based* testing technique. This means it is related to something you might already be familiar with:

. . . Chaos Monkey, from Netflix. Just like Chaos Monkey uses faults to help Netflix discover flaws in their error recovery, mutation testing uses faults to help us discover certain flaws in our tests and our code. But the way mutation testing does it, is sort of . . .

. . . upside down from what Chaos Monkey does. Chaos Monkey is best known for . . .



. . . injecting faults into Netflix's production network.



Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Edvard_Munch,_1893,_The_Scream,_oil,_tempera_and_pastel_on_cardboard,_91_x_73_cm,_National_Gallery_of_Norway.jpg

@davearonson

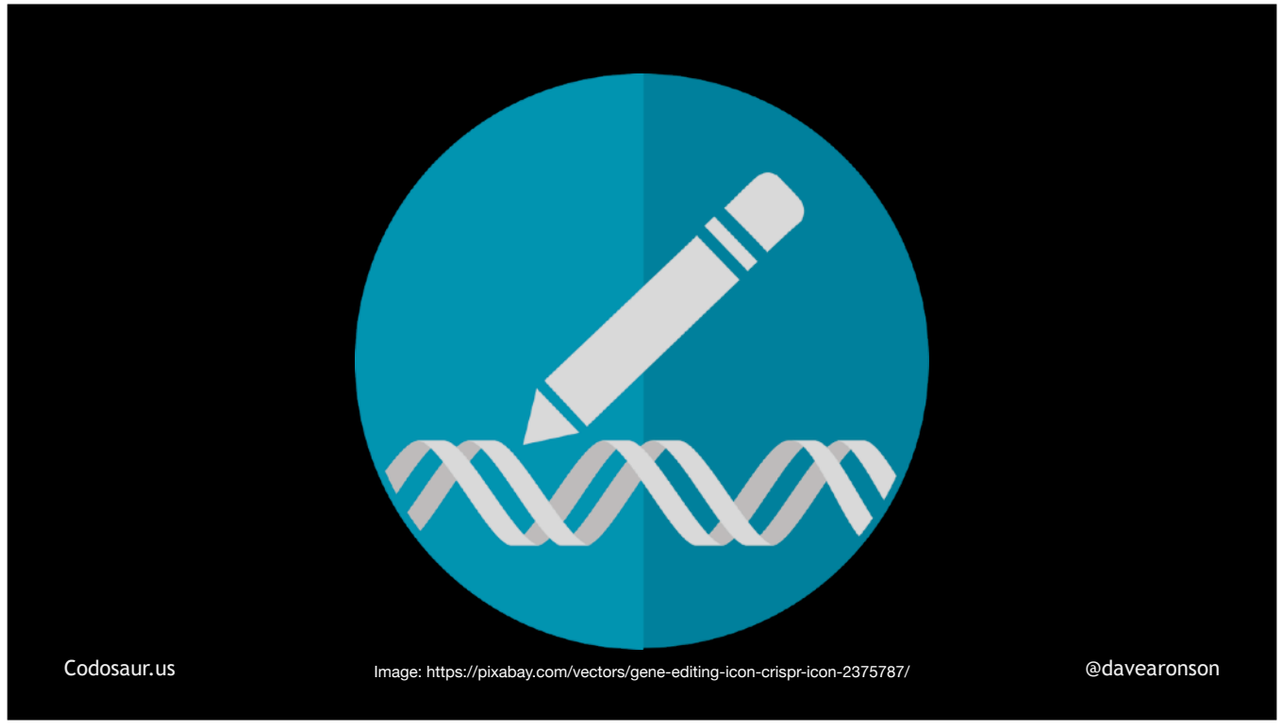
If all still goes well, in the sense that Netflix's customers don't notice, and their metrics still look good, then Netflix knows that their error recovery is working fine. Mutation testing, however, injects semantic . . .

Codosaur.us

Image: https://www.flickr.com/photos/funkblast/16937812322

@davearonson

. . . *changes*, not necessarily *problems*. We *hope* all these changes will create faults, but that depends on the test suite. It injects them *into* . . .



. . . copies of our code, not our actual network. It does its work in our . . .

Codosaur.us

Image: https://sservi.nasa.gov/articles/ladee-vibration-testing-complete/

@davearonson

. . . *test* environment, not production. (Whew!) And if everything still goes well, *in the sense that* . . .

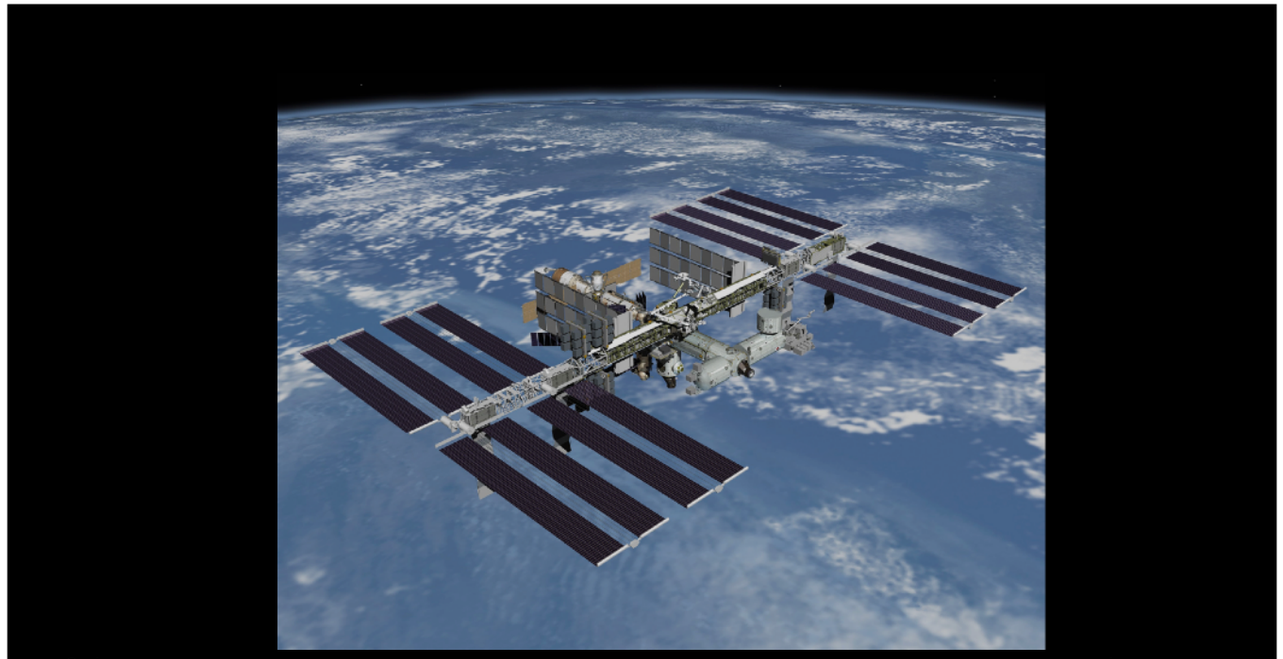
```
$ mutation_test
........................................
........................................
........................................
........................................
........................................
........................................
........................................
280 tests, 420 assertions, 4,987 mutants,
0 failures, 0 errors, 0 excluded
```

Codosaur.us @davearonson

. . . our tests all still pass, that *doesn't* mean that all is well, that means that . . .

. . . there *is* a problem! Remember, each change to our code should make *at least* one test *fail*.

But enough about differences. How does mutation testing *work?* Let's start with . . .

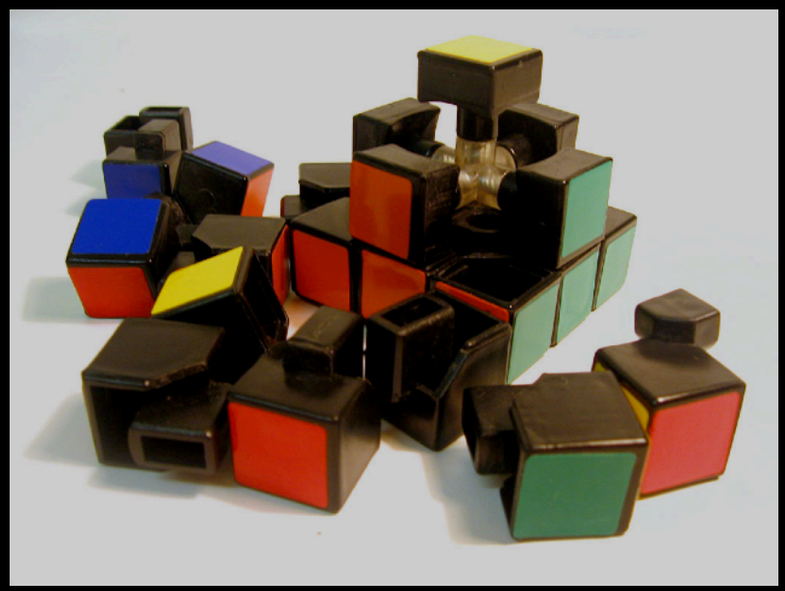


Codosaur.us

Image: https://www.flickr.com/photos/nasacommons/8980505397

@davearonson

. . . a high-level view. First, our chosen tool . . .

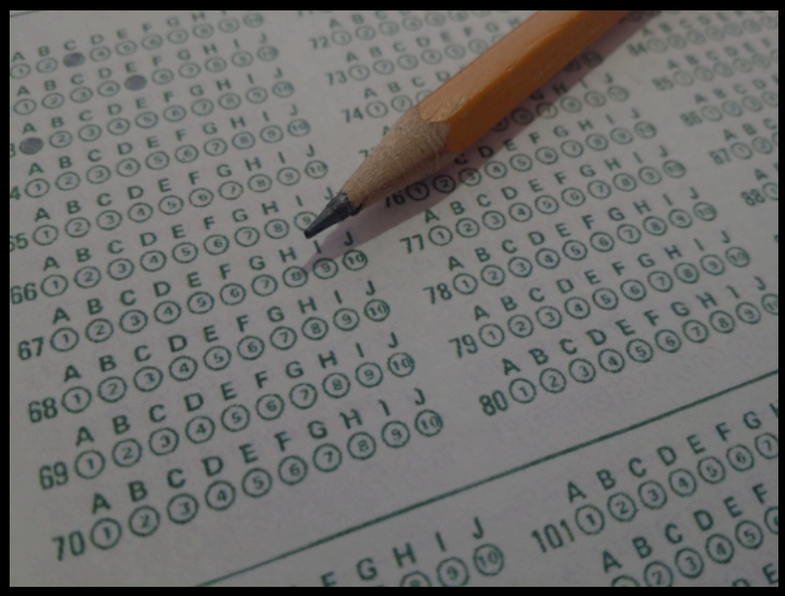


Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Disassembled-rubix-1.jpg

@davearonson

. . . breaks our code apart into pieces to test. Usually, these are our functions -- or methods if we're using objects, but I'm just going to say functions. Then, for each function, it finds . . .



Codosaur.us

Image: https://pixabay.com/fi/photos/testi-testaus-kupla-muoto-986935/

@davearonson

. . . the *tests* that cover that function. After find a function's tests, the tool . . .

. . . makes the mutants from that function. To do that, it looks closely at it to see how it can be changed. For each tiny little way the tool sees to change it, the tool makes . . .

Codosaur.us

Image: https://pixabay.com/en/genetic-modification-mutant-mutation-549889/

@davearonson

. . . one mutant, with *that one mutation*.

Once our tool is done creating all the mutants it can for a given function, it iterates over . . .

Codosaur.us

Image: https://www.flickr.com/photos/39160147@N03/15074089655

@davearonson

. . . that list. And now we get to the heart of the concept.

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

This chart represent the progress of our tool. The tools generally don't give us quite all this information, let alone so neatly organized, but it's a conceptual model I use to help illustrate the point.

For each . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . mutant, derived from . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . a given function, the tool runs the function's . . .

Mutating function `whatever` at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . tests, but it runs them . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . using the *current mutant* in place of the original function.

(PAUSE) If any test . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

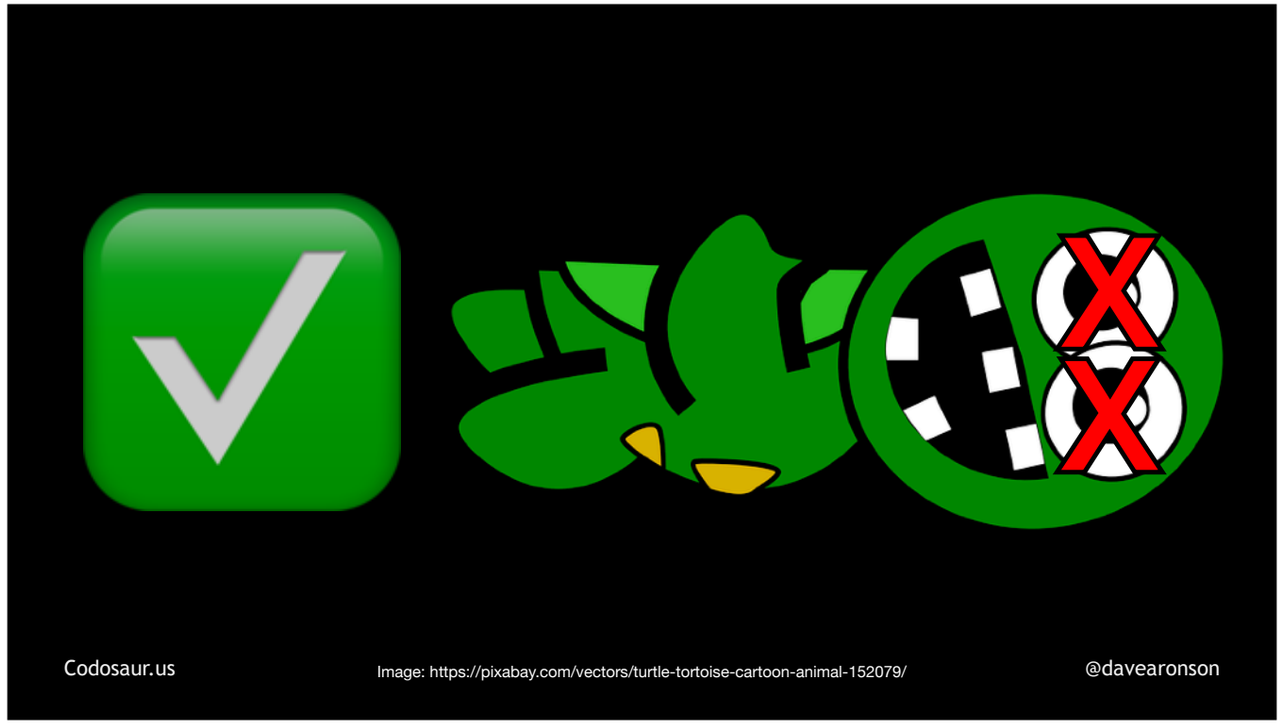
. . . *fails*, this is called . . .

Codosaur.us

Image: https://pixabay.com/id/illustrations/tengkorak-dan-tulang-bersilang-mawar-693484/

@davearonson

. . . “killing the mutant”, and it’s a . . .



. . . *good* thing. It means that our code is *meaningful* enough that the change that the tool made, to *create* this mutant, made a noticeable difference in the function's behavior, *and* that at least one test *noticed* that difference, and failed. Then, the tool will . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . mark that mutant killed, . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . stop running any more tests against it, and . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	❌						Killed
2	⏳										In Progress
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . move on to the next one. Once a mutant has made *one* test fail, we don't care how many more it *could* make fail, like perhaps some of . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2	⏳										In Progress
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . those blank cells. Like so much in computers, we only care about ones and zeroes.

On the other claw, if a mutant . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	In Progress
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

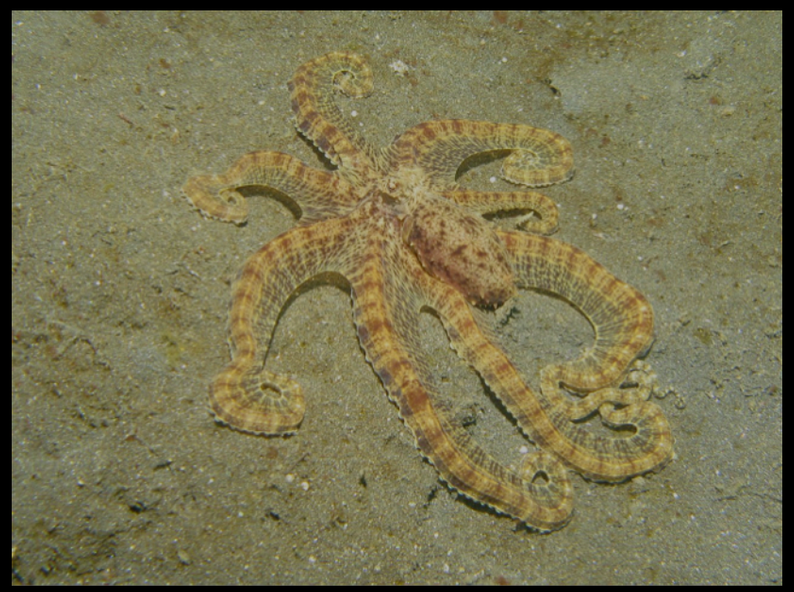
. . . lets all the tests pass, then the mutant is said to have . . .

Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						Killed
2	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	Survived!
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . *survived*. That means that the mutant has the . . .



Codosaur.us

Image: https://nl.wikipedia.org/wiki/Bestand:Mimic_Octopus2.jpg

@davearonson

. . . superpower of mimicry, skilled enough to *fool our tests!* This usually means that our code is meaningless, or our tests are lax, or both — and now it’s up to us to figure out how.

Now let's peel back one . . .

Codosaur.us

Image: https://pixabay.com/fi/photos/avaruusolento-marsin-vihreä-hirviö-722415/

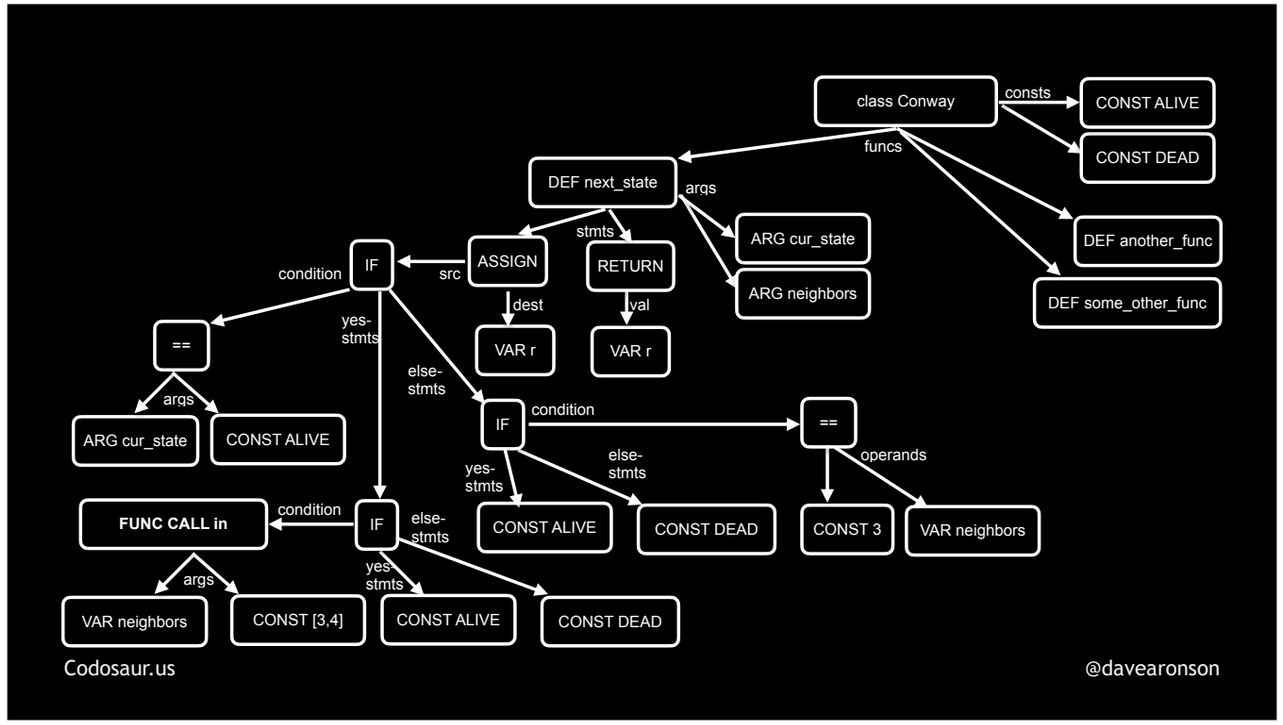
@davearonson

. . . layer of the onion, and look at some *technical details* of how this works. First, our tool parses . . .

```
class Conway:
  ALIVE = "*"
  DEAD  = " "

  @classmethod
  def next_state(cls, cur_state, neighbors):
    if cur_state == cls.ALIVE:
      r = cls.ALIVE if neighbors in [2,3] else cls.DEAD
    else:
      r = cls.ALIVE if neighbors == 3 else cls.DEAD
    return r

  def another_func:
    # whatever
```

Codosaur.us @davearonson

. . . our code, usually into an Abstract Syntax Tree. So, this code becomes . . .



. . . this AST (don’t worry about reading it). Then it . . .

Codosaur.us Image: https://www.needpix.com/photo/download/667144/cat-tree-climb-young-cat-pet-nature-cat-in-the-tree-domestic-cat-in-the-free @davearonson

. . . traverses the tree, looking for sub-trees, or branches if you will, that represent each function. After finding *them*, it handles each one as I described before, starting with looking for each one's *tests* . . . so how does it do *that?* That usually relies mainly on us developers, either . . .

```
@mumu tests-for foo
def test_foo_turns_3_into_6:
  foo(3).must_equal 6
end

def test_foo_turns_4_into_10:
  foo(4).must_equal 10
end
```

Codosaur.us @davearonson

. . . annotating our tests, or following some kind of . . .

```
def test_foo_returns_3_into_6:
  foo(3).must_equal 6
end

def test_foo_returns_4_into_10:
  foo(4).must_equal 10
end
```

Codosaur.us @davearonson

. . . naming convention. These manual techniques are often supplemented and sometimes even replaced by . . .

```
def test_foo_turns_3_into_6:
  foo(3).must_equal 6
end

def test_foo_turns_4_into_10:
  foo(4).must_equal 10
end
```

Codosaur.us

@davearonson

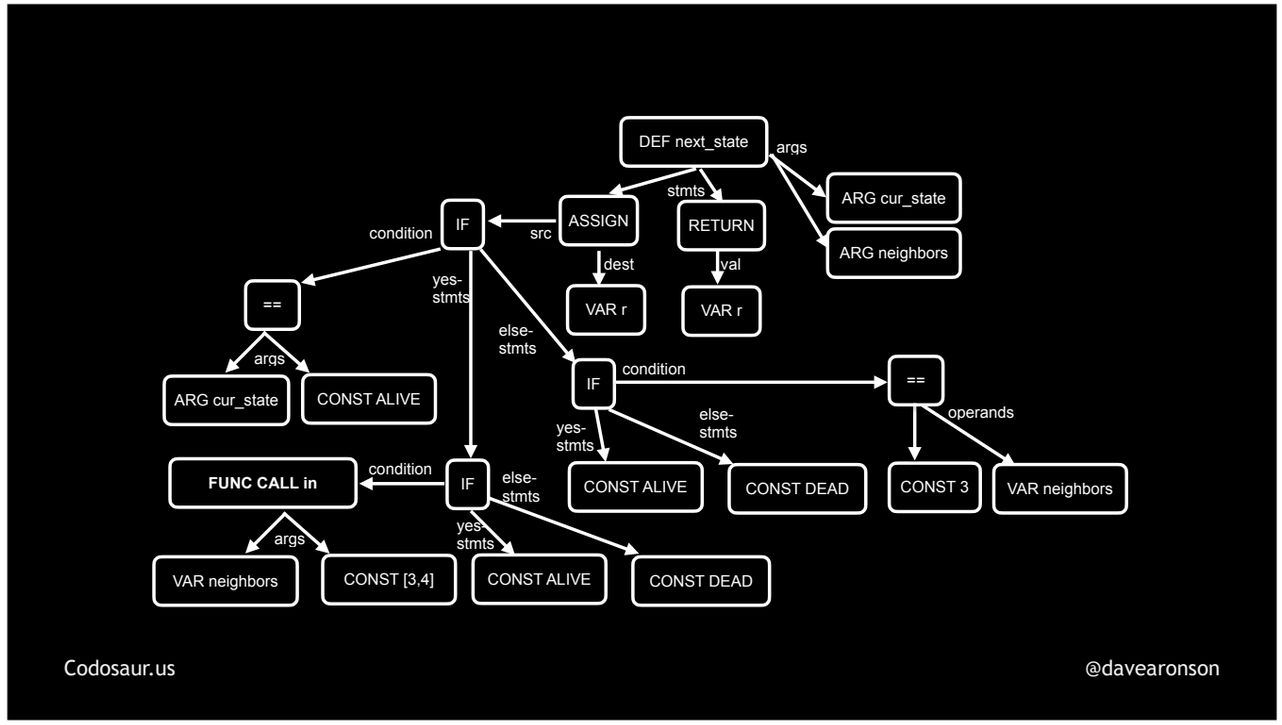
. . . the tool looking at what tests call what functions, though that can get tricky and unreliable. Anyway, after the tool has found the function's tests, it makes the mutants. To make mutants *from* an AST subtree, it . . .

Codosaur.us

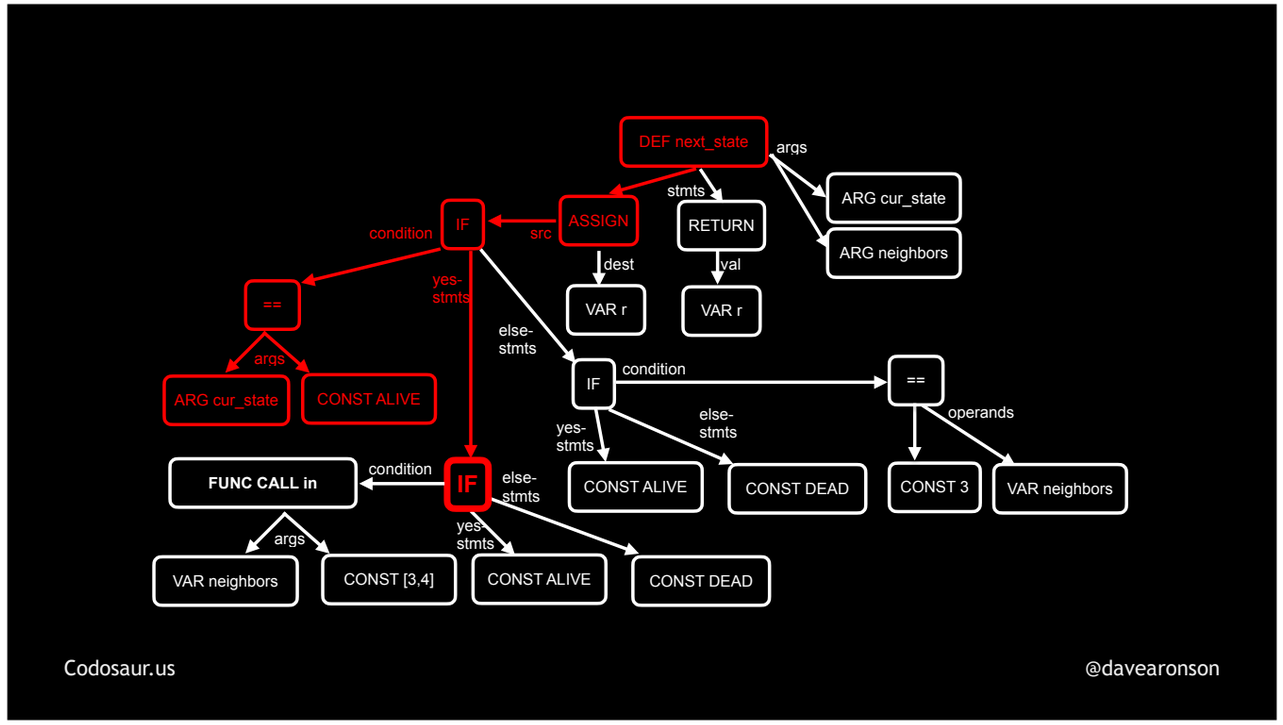
Image: https://pxhere.com/en/photo/1230969

@davearonson

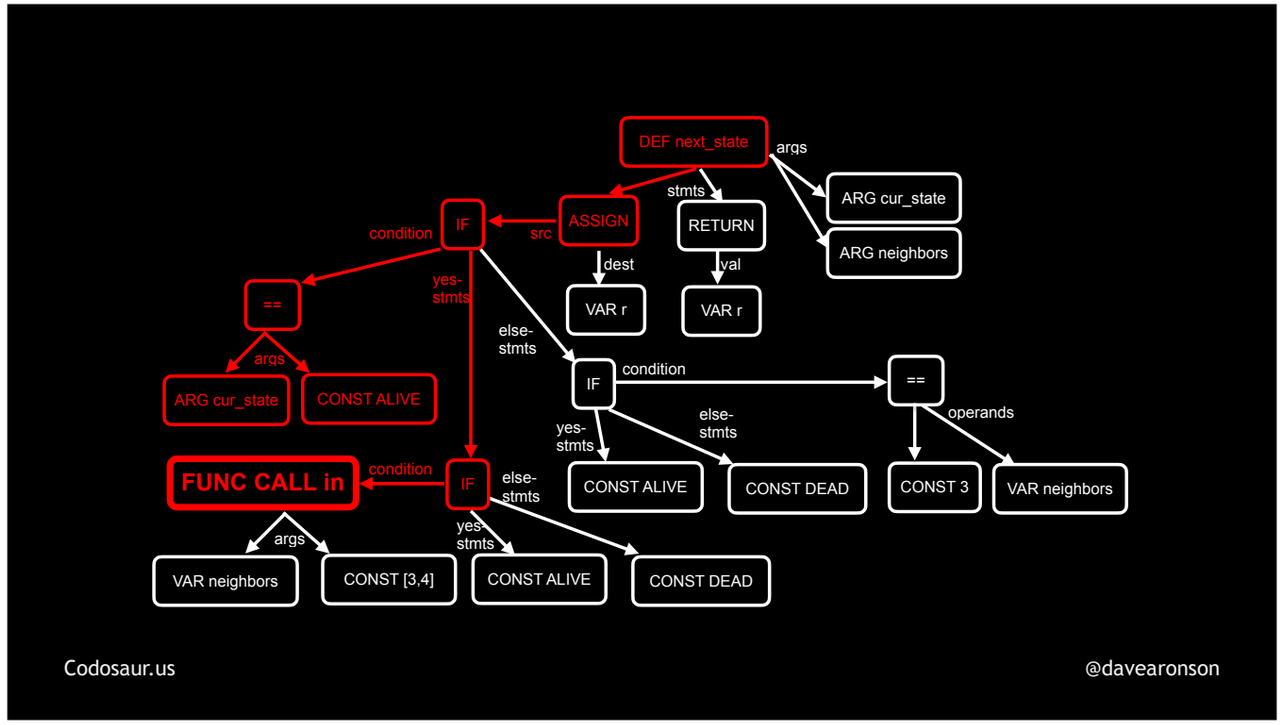
. . . traverses that subtree, just like it did to the whole thing. However, now, instead of looking for even *smaller* subtrees it can *extract*, like twigs or something, it's looking for *nodes* where it can *change* something. Each time it finds one, then for each way it can change that node, it makes one copy of the function's AST subtree, with that one node changed, in that one way. For instance, suppose our tool has started traversing . . .



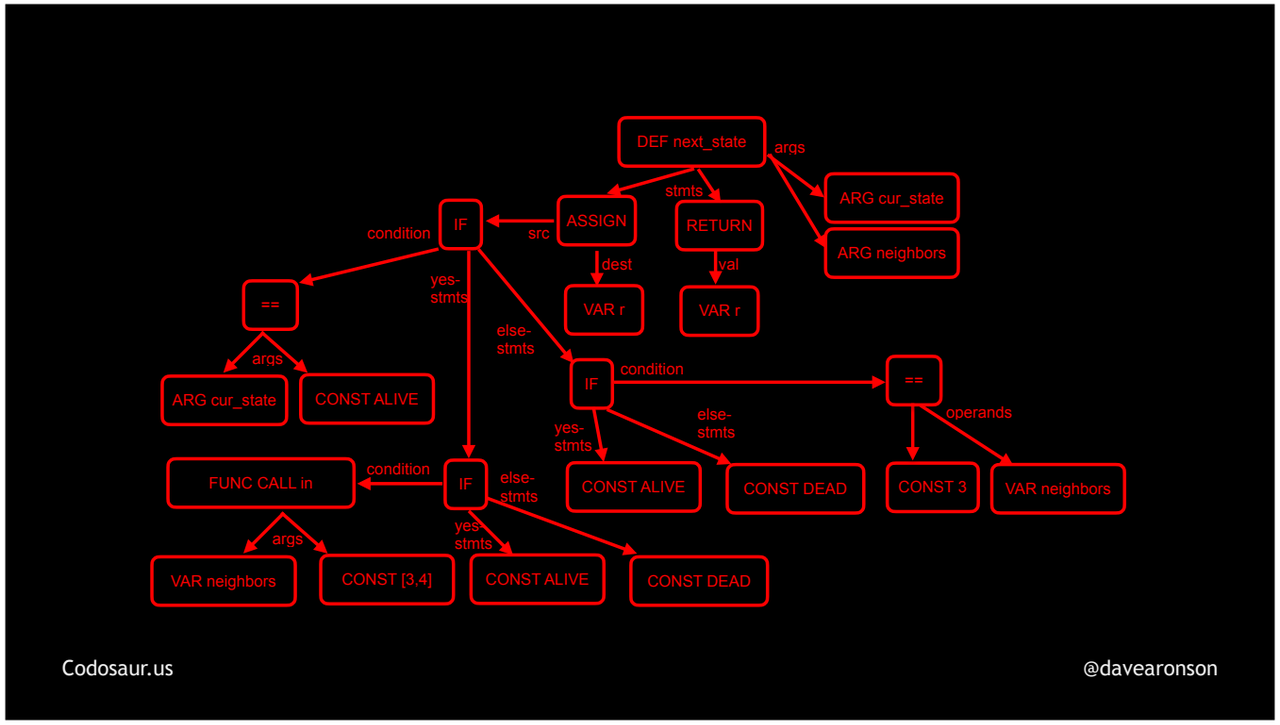
. . . the function subtree from that AST I showed earlier, and has gotten down to . . .



. . . this if statement. For each way the tool could change that node, it would make a fresh copy, of this whole subtree, with only that one node changed, in that one way. After it's done making as many mutants as it can by mutating *that* node, it would continue traversing the subtree, to . . .



. . . the *next* node. Again, for each way it could change *that* node, it would make a copy of this whole subtree, with only that mutation. And so on, until it has . . .



. . . traversed the entire subtree.

Now, I've been talking a lot about changing things, so what kind of changes are we talking about? There are quite a lot!

```
x + y   could become: x - y
                      x * y
                      x / y
                      x ** y

x || y  could become: x && y
                      x ^ y

x | y   could become: x & y
                      x ^ y
```

Maybe even swap *between* sets!

Codosaur.us

@davearonson

It could change a mathematical, logical, or bitwise operator from one to another, even across categories when the language allows.

`x - y` could *also* become `y - x`

`x / y` could *also* become `y / x`

`x ** y` could *also* become `y ** x`

`"x" <> "y"` could *also* become `"y" <> "x"`

Codosaur.us

@davearonson

When the *order* of operands matters, it could *swap* them.

```
x < y
```

could become:

```
x <= y
x == y
x != y
x >= y
x >  y
```

Codosaur.us

@davearonson

It could change a *comparison* from one to another.

`x`

could become:

`-x`
`!x`
`~x`

. . . or vice-versa!

Codosaur.us

@davearonson

It could insert *or remove* a mathematical, logical, or bitwise *negation*.

```
a = foo(x)
b = bar(y)
```

could become:

```
a = foo(x)
```

or

```
b = bar(y)
```

Codosaur.us

@davearonson

It can remove an entire *statement*.

```
if x == y:
  foo(z)
```

could become:

```
foo(z)
```

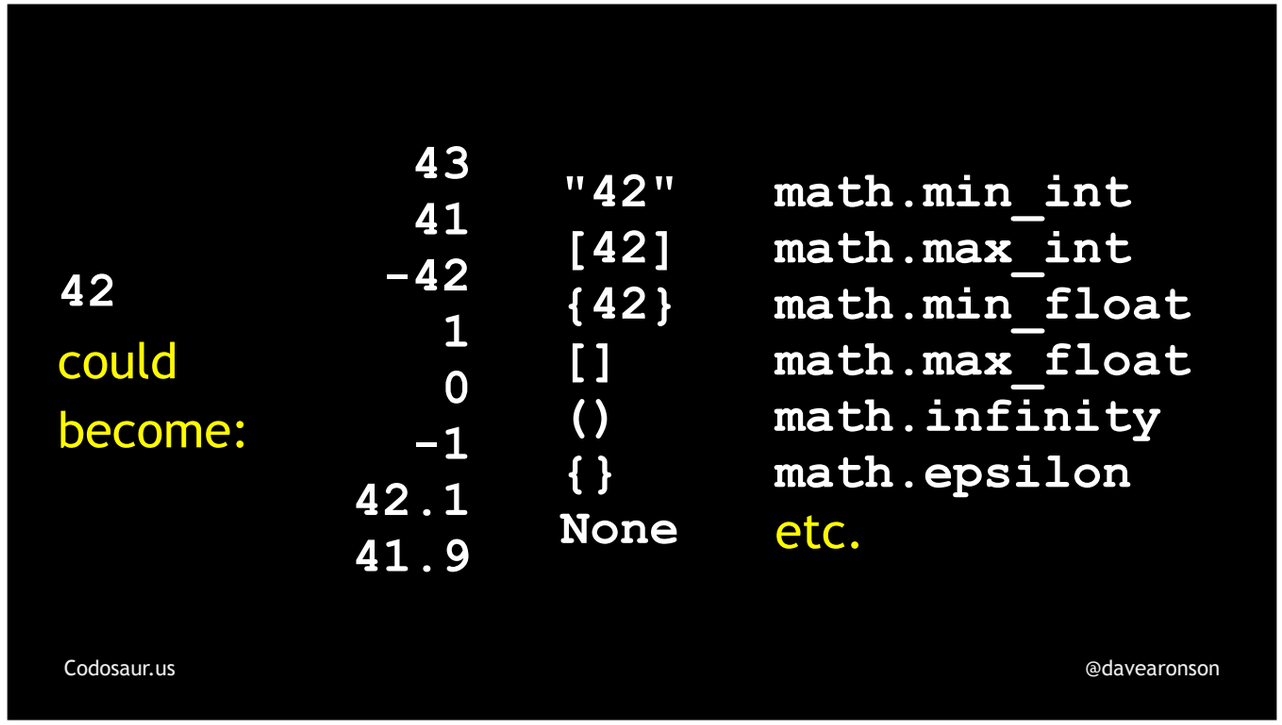

It can remove an if-condition . . .

```
while x == y:
  foo(z)
```

could become:

```
foo(z)
```


. . . or a looping condition.



It could change a value to some other value, such as changing 42 to any of these, and many more but I had to stop somewhere. It could even change it to something of a different and possibly incompatible type, such as changing a number into a, if I may quote . . .

Codosaur.us

Image: https://www.flickr.com/photos/herval/50674160

@davearonson

. . . Smeagol, “string, or nothing!”

There are *many* many more types of changes, but I trust you get the idea!

From here on, there are no more low-level details I want to add, so let’s *finally* walk through some *examples!* We’ll start with an easy one. Suppose we have a function . . .

```
def power(x, y):
    x ** y
```

Codosaur.us @davearonson

. . . like so. Never mind *why*, it makes a good simple example, so let’s just roll with it.

Think about what a mutant made from this might *return*, since that's what our tests would probably be looking at. It sure doesn’t look like it has side effects.

Mainly, such a mutant could return results such as . . .

```
x + y                   math.min_int
x - y                   math.max_int
x * y                   math.max_float
x / y                   math.min_float
y ** x                  math.infinity
x                       math.epsilon
y                       raise(DeliberateError)
0                       "some random string"
1                       []
-1                      ()
0.1                     {}
-0.1                    None
```

Codosaur.us @davearonson

. . . any of *these* expressions or constants, and, again, many more but I had to stop somewhere.

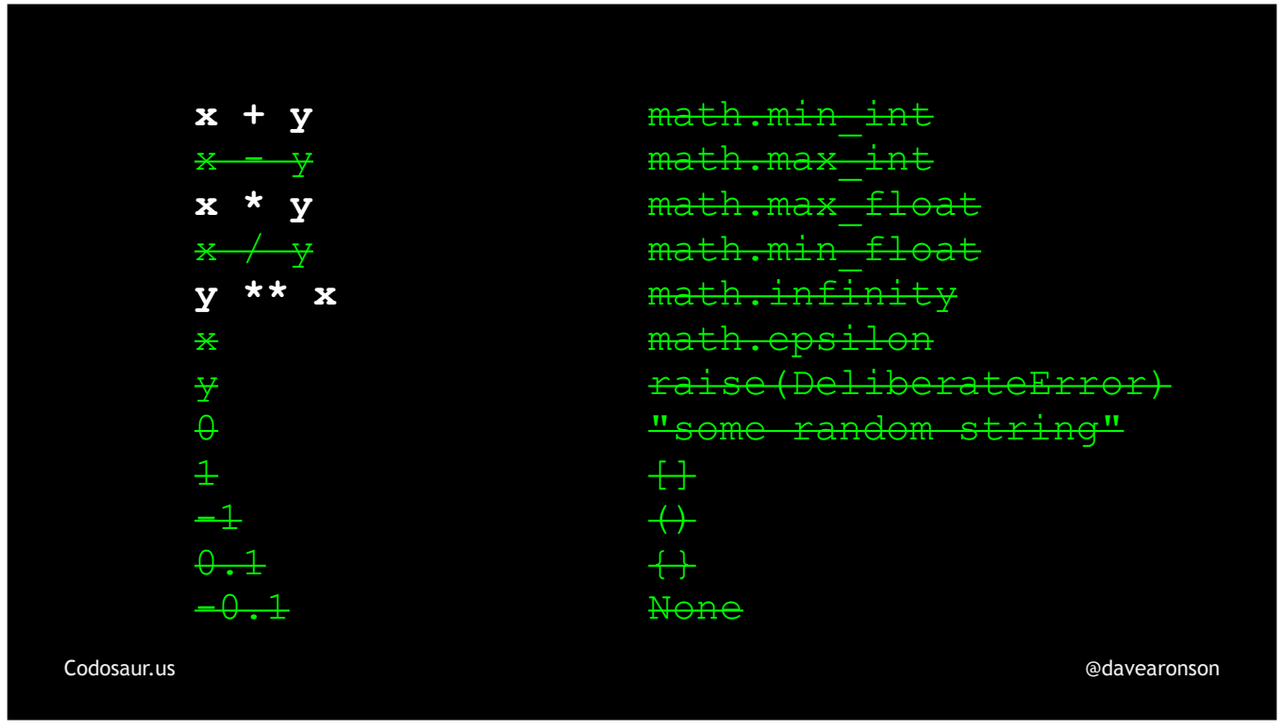
Now suppose we had only one test . . .

```
assert power(2, 2) == 4
```

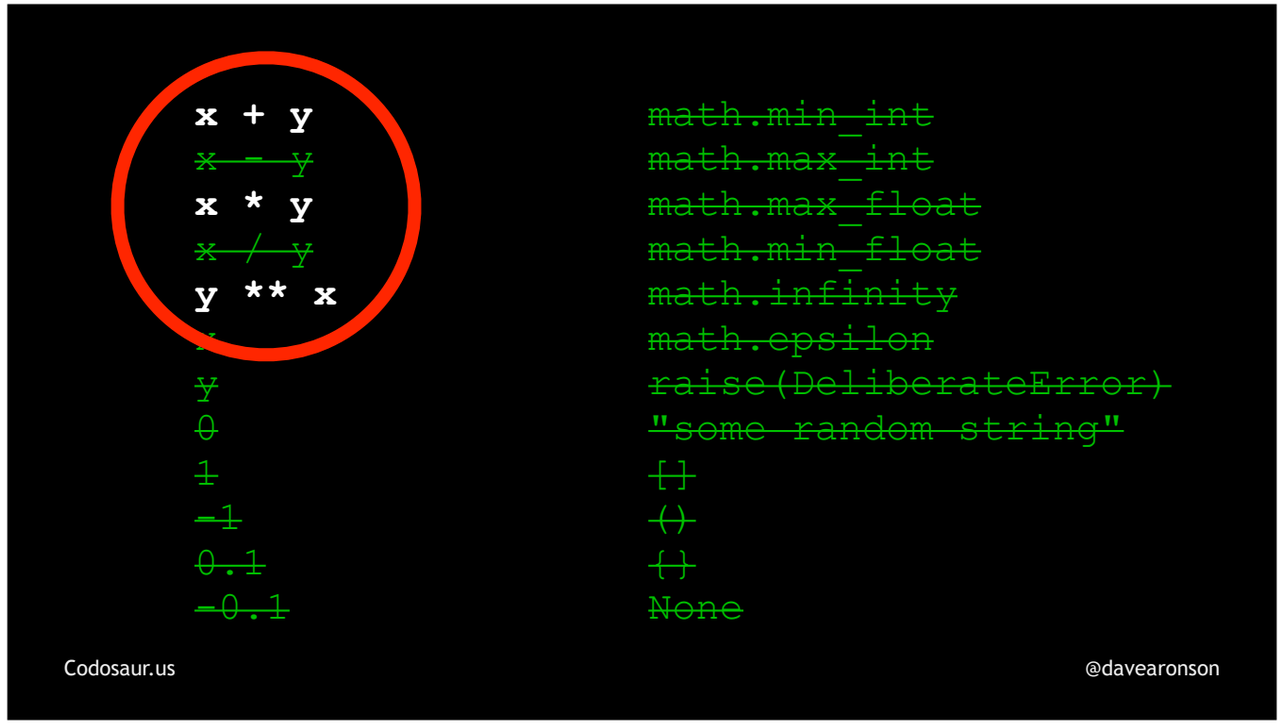
Codosaur.us

@davearonson

. . . like so. This is a rather poor test, and I think at least one reason why is immediately obvious to most of us, but even so, *most* of those mutants on the previous slide *would get killed* by this test, the ones shown . . .



. . . here in crossed-out green. The ones returning constants, are very unlikely to match. There's no particular reason a tool would put a 4 there, as opposed to zero, 1, -1, minimum and maximum signed and unsigned integers and floats, infinity, minus infinity, and other such significant numbers. Subtracting one argument from the other gets us zero, dividing them gets us one, returning either argument alone gets us two, and the mismatched types and deliberate errors will at *least* make the test not pass. But . . .



. . . addition, multiplication, and exponentiation in the reverse order, all get us the correct answer. Mutants based on *these* mutations will therefore *"surivive"* our test.

So how do we see that happening? When we run our tool, it gives us a report, that looks roughly like . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

43 -   x ** y
43 +   y ** x
```

Codosaur.us

@davearonson

. . . this. The format will vary greatly depending on exactly which tool we use, but *semantically*, the information should be the same. And that is that if we changed . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

43 -   x ** y
43 +   y ** x
```

Codosaur.us

@davearonson

. . . the function called power, in . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

43 -   x ** y
43 +   y ** x
```

Codosaur.us

@davearonson

. . . file demo.py, at line 42 . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

43 -   x ** y
43 +   y ** x
```

Codosaur.us

@davearonson

. . . in any of four ways, then all its tests still pass.

And, that those ways are: . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -    x ** y
43 +    x + y

43 -    x ** y
43 +    x * y

43 -    x ** y
43 +    y ** x
```

Codosaur.us

@davearonson

. . . to change the function declaration to swap the arguments, or . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -    x ** y
43 +    x + y

43 -    x ** y
43 +    x * y

43 -    x ** y
43 +    y ** x
```

Codosaur.us

@davearonson

. . . change the function body to change the exponentiation into addition or multiplication, or . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

43 -   x ** y
43 +   y ** x
```

Codosaur.us

@davearonson

. . . to change the body to swap the exponentiation’s operands.

So what is . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

43 -   x ** y
43 +   y ** x
```

Codosaur.us

@davearonson

. . . this set of surviving mutants trying to tell us? We can tell from a glance at . . .

```
def power(x, y):
  x ** y
```

Codosaur.us

@davearonson

. . . our code, that it’s probably not trying to tell us about redundant or unreachable code. The body is just one line so that sort of problem is extremely unlikely. So it’s probably a test gap! The question now boils down to, how are . . .

```
function "power" (demo.py:42)
has 4 surviving mutants:

42 - def power(x, y):
42 + def power(y, x):

43 -   x ** y
43 +   x + y

43 -   x ** y
43 +   x * y

43 -   x ** y
43 +   y ** x
```

Codosaur.us

@davearonson

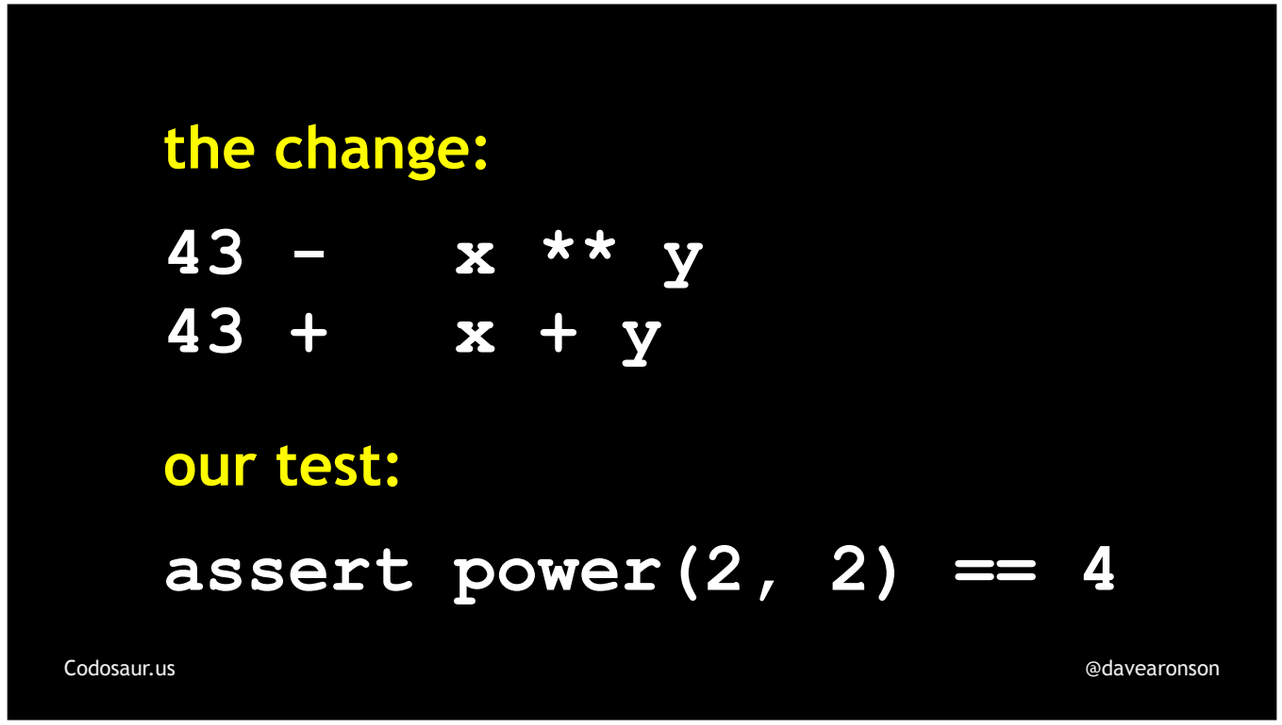
. . . these mutants surviving? The usual answer is that . . .

```
mutant_power(x, y)

==

original_power(x, y)
```

Codosaur.us

@davearonson

. . . they return the same result as the original function. Or they have the same side effect — whatever our tests are looking at. To determine how *that* happens, it helps to take a closer look at one mutant, and a test it passes. Let's start with . . .



. . . the "plus" mutant. Looking at the change, together with our test, makes it clear that this one survives because . . .

Codosaur.us

Image: meme going around, original source unfindable, sorry

@davearonson

. . . two *plus* two equals two *to* the two. (And so does two *times* two, but he's in the background, we can save him for later.)

So how can we *kill* . . .

the change:

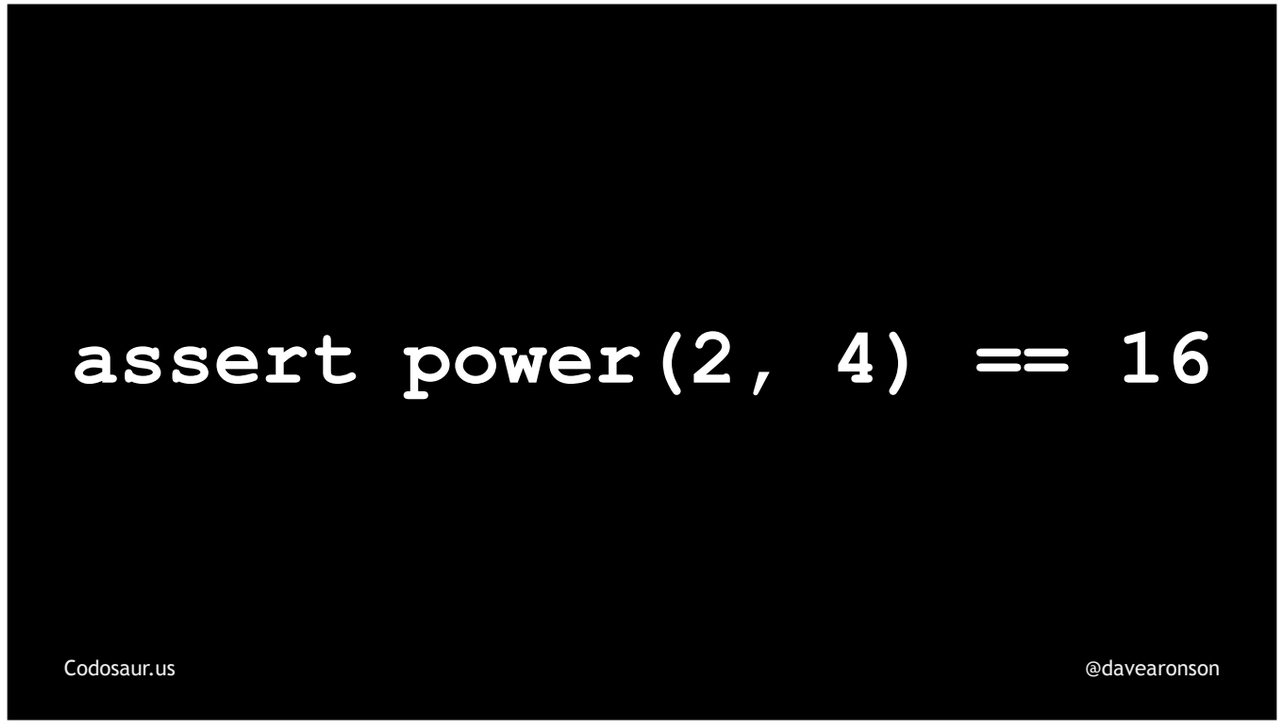
```
43 -    x ** y
43 +    x + y
```

our test:

```
assert power(2, 2) == 4
```

Codosaur.us @davearonson

. . . this mutant, in other words, make at least one test fail when run against it, that would pass when run against the original code? We just need to make at least one test use inputs such that x *plus* y is different from x *to the* y. For instance, we could add a test or change our existing test to . . .



. . . assert that two to the *fourth* power is *sixteen*. All the mutants that our original test killed, *this would still kill*. But in addition, two *plus* four is six, not six*teen*, so **this kills the plus mutant.** (See how that works?)

Better yet, two *times* four is eight, which is *also* not sixteen! So, this kills the "times" mutant as well. Killing one mutant often kills many other mutants of the same function.

But . . .

. . . the pair of argument-swapping mutants survive! What, how can that be? It’s because . . .

```
4 ** 2 == 16

2 ** 4 == 16
```

Codosaur.us

@davearonson

. . . four squared is the same as two to the fourth, they’re both sixteen. But that’s not a big deal, we can . . .

. . . attack these mutants separately, no need to kill all the mutants in one shot and be some kind of superhero about it. To kill *them*, again, we can either add a test, or adjust an existing test, to something like . . .

```
assert power(2, 3) == 8
```

Codosaur.us @davearonson

. . . this, asserting that two to the *third* power is *eight*. Three squared is nine, not eight, so this kills the argument-swapping mutants. Better yet, two *plus* three is five, two *times* three is six, and both of those are not eight, so the "plus" and "times" mutants *stay* dead, and we don't get any . . .

. . . zombie mutants wandering around, even if . . .

```
assert power(2, 3) == 8
```

Codosaur.us @davearonson

. . . this were still our one and only test. (PAUSE!) With these inputs, the correct operation is the only simple common one that yields the correct answer. This isn't the *only* solution, though; even if we stuck to single digits, there are *lots* of ways to skin . . .

Codosaur.us

Image: https://www.flickr.com/photos/greyloch/48214242842

@davearonson

. . . *that* flerken!

This may make mutation testing sound . . .

Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Simple_Simon_LCCN2003677693.jpg

@davearonson

. . . simple, but this was a downright trivial example.

So let’s look at a more *complex* example!

Suppose we have a function to send a message, . . .

```
def send_message(buf, len):
  sent = 0
  while sent < len:
    sent += send_bytes(buf + sent,
                       len - sent)
  return sent
```

Codosaur.us @davearonson

. . . like so. This function, send_*message*, uses send_*bytes* to send as many bytes as send_bytes *could* send, like a woodchuck, looping to pick up where it left off, until the message is all sent. This is a very common pattern in communication software.

A mutation testing tool could make lots of mutants from this, but one of particular interest, would be . . .

```
def send_message(buf, len):
  sent = 0
  while sent < len:
    sent += send_bytes(buf + sent,
                       len - sent)
  return sent
```

Codosaur.us

@davearonson

. . . this, an example of removing a looping control.

That would make the code read effectively like . . .

```
def send_message(buf, len):
  sent = 0
  sent += send_bytes(buf + sent,
                     len - sent)
  return sent
```


. . . this.

Now suppose that this mutant does indeed survive our test suite, which consists mainly of . . .

. . . *this*. (PAUSE!) There's a bit more that I'm not going to show you *quite* yet, dealing with setting the size and actually creating the message. But even without seeing that test code, what does the survival of that non-looping mutant tell us? (PAUSE!)

If a mutant that only goes through . . .

```
def send_message(buf, len):
  sent = 0
  while sent < len:
    sent += send_bytes(buf + sent,
                       len - sent)
  return sent
```

Codosaur.us @davearonson

. . . that loop once, acts the same as our normal code, as far as our tests can tell, that means that our *tests* are only making our *normal* code go through that loop once. So, what does *that* mean? (PAUSE!) By the way, you'll find that interpreting mutants involves a lot of asking yourself *“so, what does that mean"*, often deeply recursively!

In this case, it means that we’re not testing sending a message larger than send_*bytes* can handle in one chunk! There are many ways that can happen, but we’re only going to look at two possibilities. The most *likely* is that we should have, but simply forgot, or didn’t *bother,* to test with a big enough message. For instance, . . .

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

```
msg = "foo"
size = length(msg)
# other setup, like stubbing send_bytes
assert send_message(msg, size) == size
```

Codosaur.us @davearonson

. . . suppose our maximum chunk size, what send_*bytes* can handle in one chunk, is 10,000 bytes. But . . .

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

```
msg = "foo"
size = length(msg)
# other setup, like stubbing send_bytes
assert send_message(msg, size) == size
```

Codosaur.us @davearonson

. . . we're only testing with an itty-bitty *three* byte message. (PAUSE!)

The obvious fix is to deliberately use a message larger than our maximum chunk size. With this kind of message, we can easily construct one, as shown . . .

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

```
size = socket.max_chunk_size + 1
msg = "x" * size
# other setup, like stubbing send_bytes
assert send_message(msg, size) == size
```

Codosaur.us @davearonson

. . . here. (PAUSE!) We just take the maximum size, add some, and construct that big a message.

But now let's look at another possible cause and solution. Maybe we *did* test with the *largest* permissible message, out of a set of predefined messages, or at least message *sizes*. For instance, . . .

in module Message:

```
SmallMsgSize = 1_000
LargeMsgSize = 5_000  # the largest
```

in test_send_message:

```
size = Message.LargeMsgSize
msg = Message.make_msg("a" * size)
# other setup, like stubbing send_bytes
assert send_message(msg, size) == size
```

Codosaur.us @davearonson

. . . here we have Small and Large message sizes, we test with a Large, and yet, this mutant survives! In other words, we're still sending the whole message in one chunk. What could possibly be wrong with that? It sounds like a *good* thing to me! What is this mutant trying to tell us in this case? (PAUSE!)

In *this* scenario, it's trying to tell us that a version of send_message with the looping removed will do the job just fine. If we remove the looping, we wind up with . . .

```
def send_message(buf, len):
  sent = 0
  sent += send_bytes(buf + sent,
                     len - sent)
  return sent
```

Codosaur.us @davearonson

. . . this code I showed you earlier. If we run our mutation testing tool on *this*, it will show some other stuff as now being redundant, because we only needed it to support the looping. If we *also* remove that, then it boils down to . . .

```
def send_message(buf, len):
  return send_bytes(buf + sent,
                    len - sent)
```

Codosaur.us @davearonson

. . . this. (PAUSE!) Now the message is clear: the *entire* send_message *function* may well be *redundant*, so we can just use send_*bytes directly!* In real-world code, though, it might not be, because there may be some logging, error handling, and so on, needed in send_message, but at the very least, the *looping* was redundant. Fortunately, when it's this kind of problem, the usual solution is clear and easy, just rip out the extra junk that the mutant doesn't have. This will also make our code more *maintainable*, by getting rid of useless cruft that just gets in the way of understanding it.

Now that we've seen a few different examples, of spotting bad tests and redundant code, I'd like to address some . . .

. . . common questions. First, this all sounds pretty weird, deliberately making tests fail, to prove that the code succeeds! Where did this whole bizarro idea come from anyway? Mutation testing has a surprisingly . . .

Codosaur.us

Image: https://pixabay.com/photos/egypt-education-history-egyptian-1826822/

@davearonson

. . . long history -- at least in the context of computers. It was first proposed in 1971, in Richard Lipton's term paper titled “Fault Diagnosis of Computer Programs”, at Carnegie-Mellon University. The first *tool* didn't appear until nine years *later*, in 1980, as part of Timothy Budd's PhD work at Yale. Even then, it was not *practical* on typical developer-grade computers, until the early 2000s, with advances in CPU *speed*, multi-*core* CPUs, larger and cheaper memory, and so on.

That leads us to the next question: *why* is it so CPU- and memory-intensive? To answer that, we need do some math, but don't worry, it's pretty basic. Suppose our functions have, on average, . . .

10 lines

Codosaur.us

@davearonson

. . . about ten lines each. And each line has about . . .

10 lines

x 5 mutation points

Codosaur.us

@davearonson

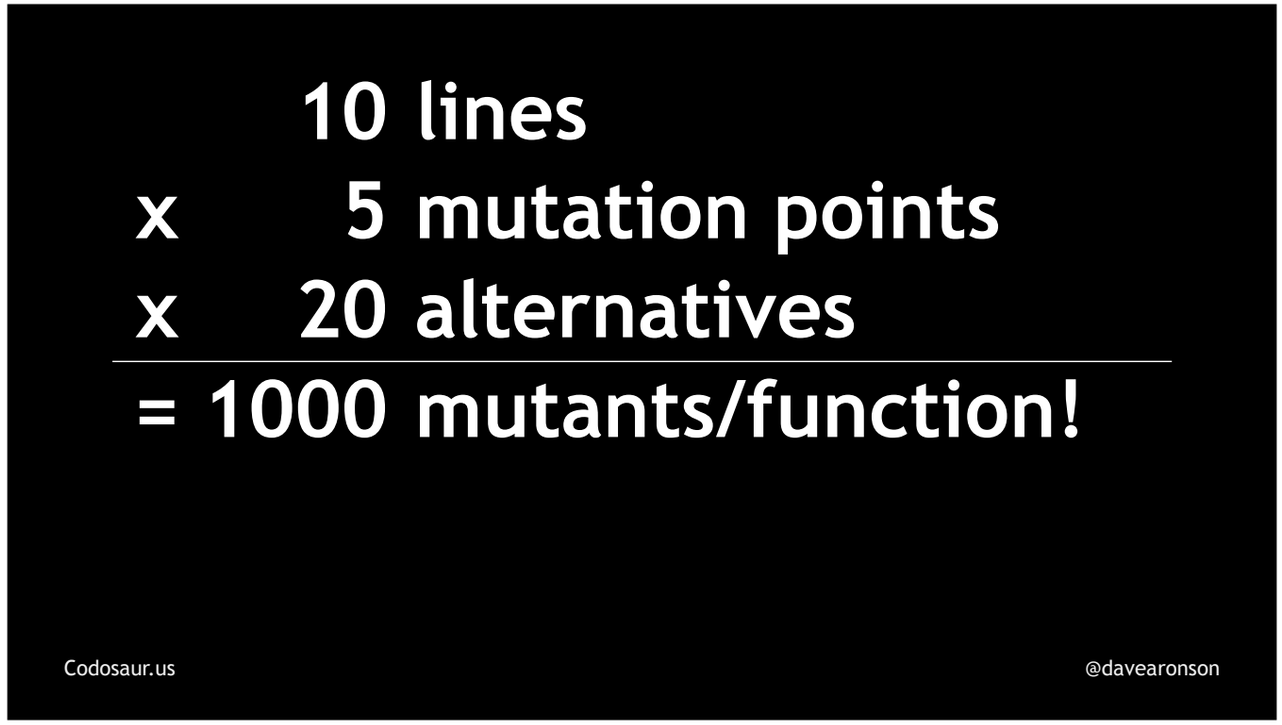
. . . five places where it can be mutated, to any of about . . .

```
      10 lines
x      5 mutation points
x     20 alternatives
```

Codosaur.us

@davearonson

. . . twenty alternatives. That works out to about . . .



. . . a thousand mutants for each function! And for each one, we'll have to run somewhere between one test, if we're lucky and kill it on the first try, all the way up to *all* of that function's tests, if we kill it on the last try, or worse yet, it survives.

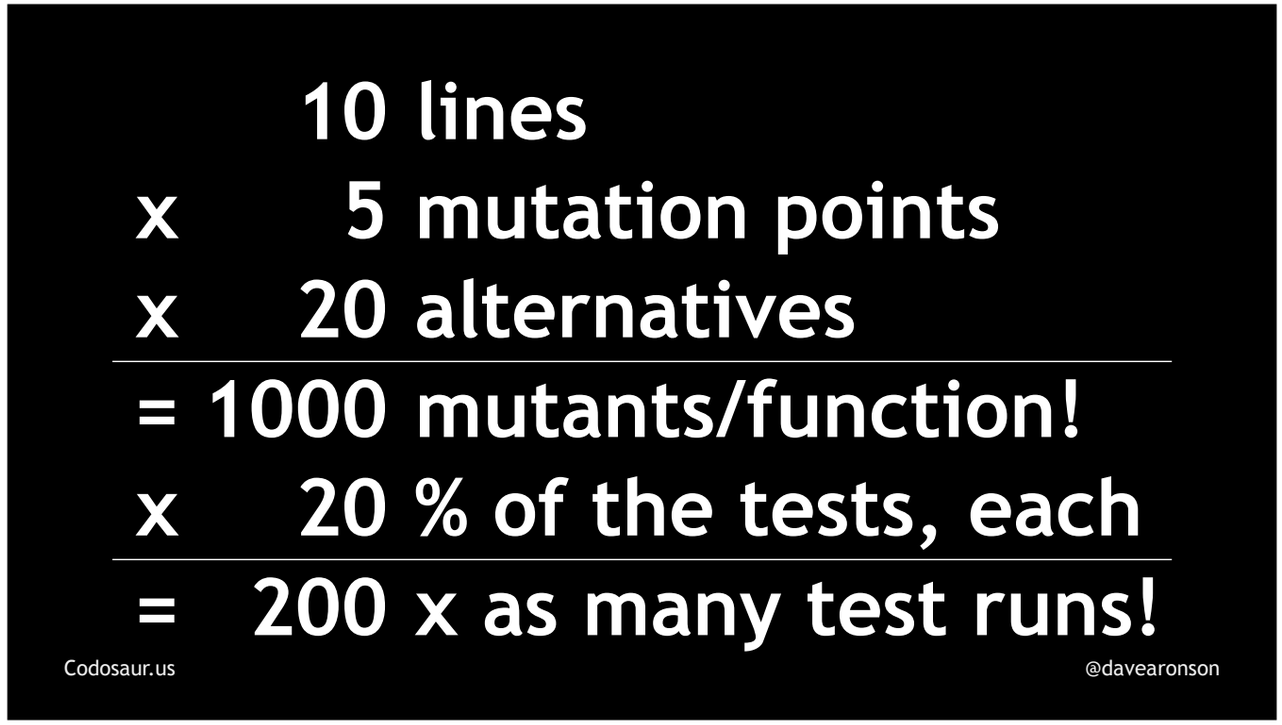
Suppose we wind up running just . . .

```
      10 lines
x      5 mutation points
x     20 alternatives
-----------------------
= 1000 mutants/function!
x     20 % of the tests, each
-----------------------
```

Codosaur.us

@davearonson

. . . one *fifth* of the tests for each mutant. Since we start with a thousand mutants, that's still . . .



. . . *two hundred times* the test runs for that function, compared to regular testing. If our test suite normally takes a zippy ten seconds, then with these assumptions, mutation testing will take about *two thousand* seconds. That might not sound like much, because I'm saying "seconds", but it's over *half an hour!* I don't want to sit and wait for that! So remember to use the filtering I spoke of at the start.

To summarize at last, mutation testing is a powerful technique to . . .

😀 Checks that code is meaningful

. . . help ensure that our code is meaningful and . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

. . . our tests are strict. It's . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

😀 Easy to get started with

Codosaur.us

@davearonson

easy to get started with, in terms of setting up most of the tools and annotating our tests if needed
(which may be *tedious* and *time-consuming* but at least it's *easy*),
but it's . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

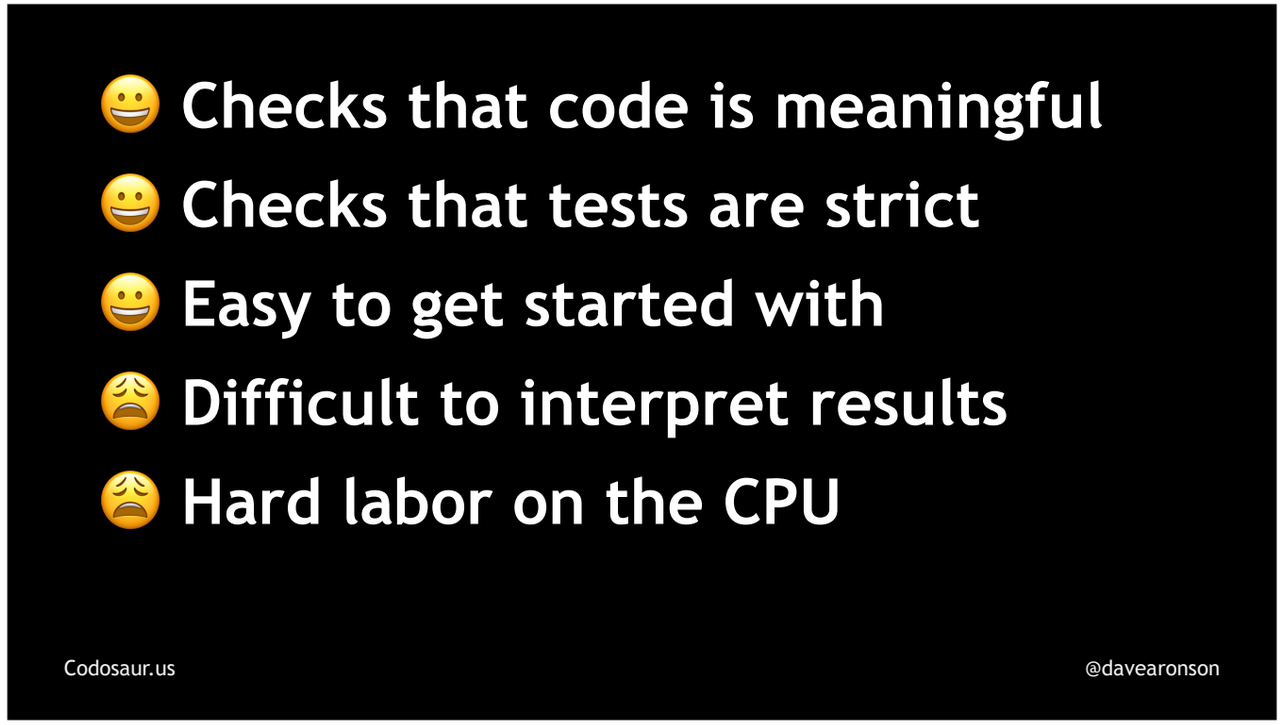
😀 Easy to get started with

😩 Difficult to interpret results

Codosaur.us

@davearonson

. . . not so easy to interpret the results, nor is it . . .



. . . easy on the CPU.
Even if these drawbacks mean it might not be a good fit for our particular current projects right now, I still think it's just . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

😀 Easy to get started with

😩 Difficult to interpret results

😩 Hard labor on the CPU

😎 Fascinating concept! 🤓

Codosaur.us

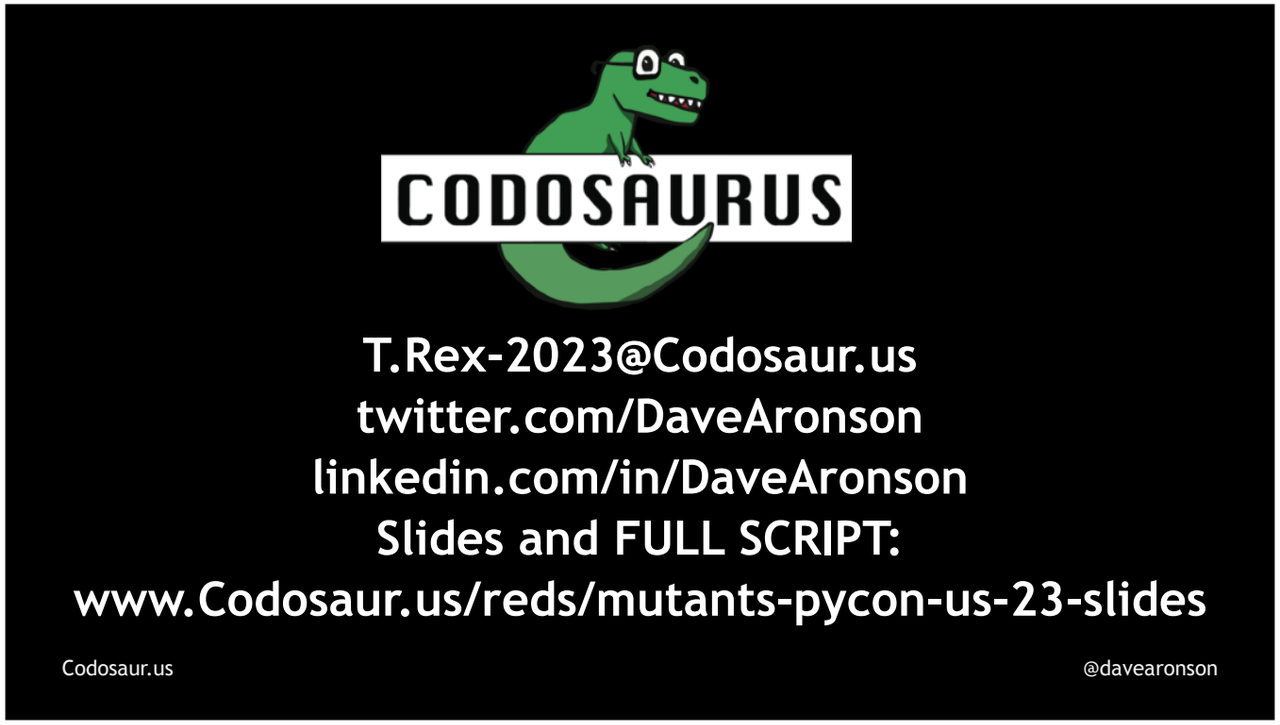
@davearonson

. . . a really cool idea . . . in a geeky kind of way.

If you'd like to try mutation testing for yourself . . .

. . . here is a list of tools for some popular languages and platforms . . . and some others; I doubt many of you are doing FORTRAN-77 these days. The tools I *know* are outdated, are crossed out.

And now . . .



. . . it's your turn! Any questions?