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

CURRENT TIME, at medium speed: was 39 minutes, want max 40 of content (plus 5 of Q&A), so aim for 35-40, so SKIP SOME MORE and WATCH AD-LIBS!

Kill All Mutants!

(Intro to Mutation Testing)

by Dave Aronson

CODOSAURUS

GOOten Tahg, Veen!

Mein Name ist Dave Aronson,

(I'm Dave Aronson,)

Codosaur.us

Image: me speaking at JSConf Hawai'i 2020

@davearonson

Mein Nah-muh ist Dave Aronson,

der T-Rex von Codosaurus,

CODOSAURUS

(the T. Rex of Codosaurus,)

Image: my company logo!

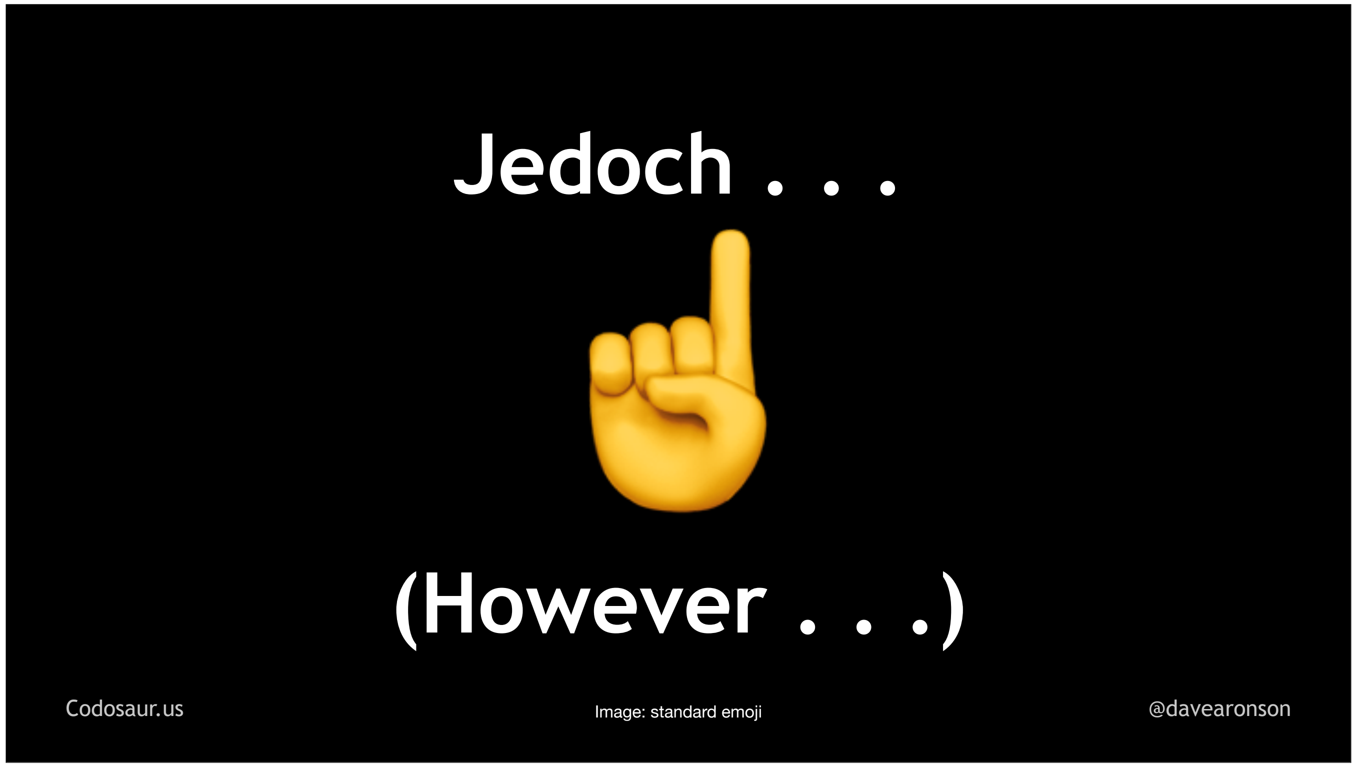
ikh bin deahr T. Rex fon Codozowrus,

und ikh flog hair

auf meinem P’teroDAKteelus

um oykh zoo TSEIgen

vee mahn moo-TAHN-ten TEU-tet!

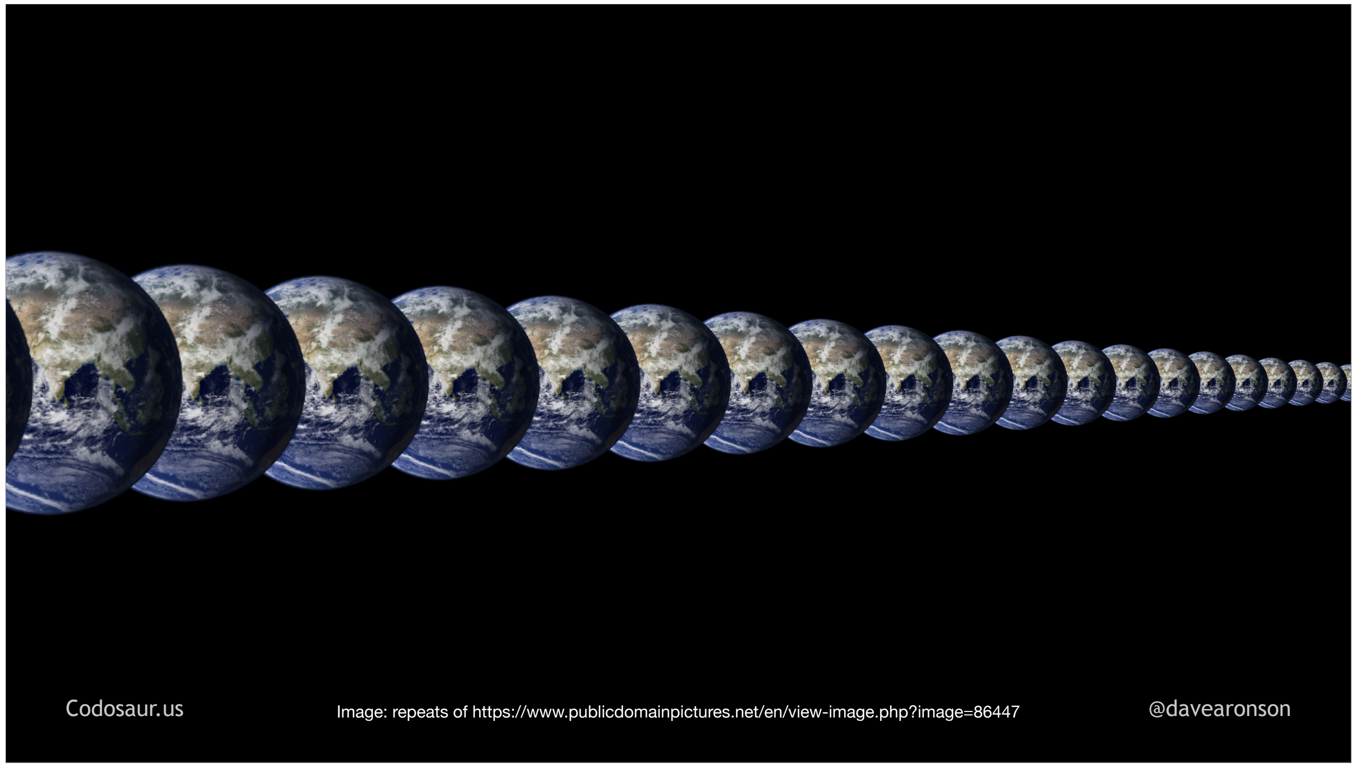

Jedoch . . .
(However . . .)
Codosaur.us
Image: standard emoji
@davearonson

yeh-DOKH . . .

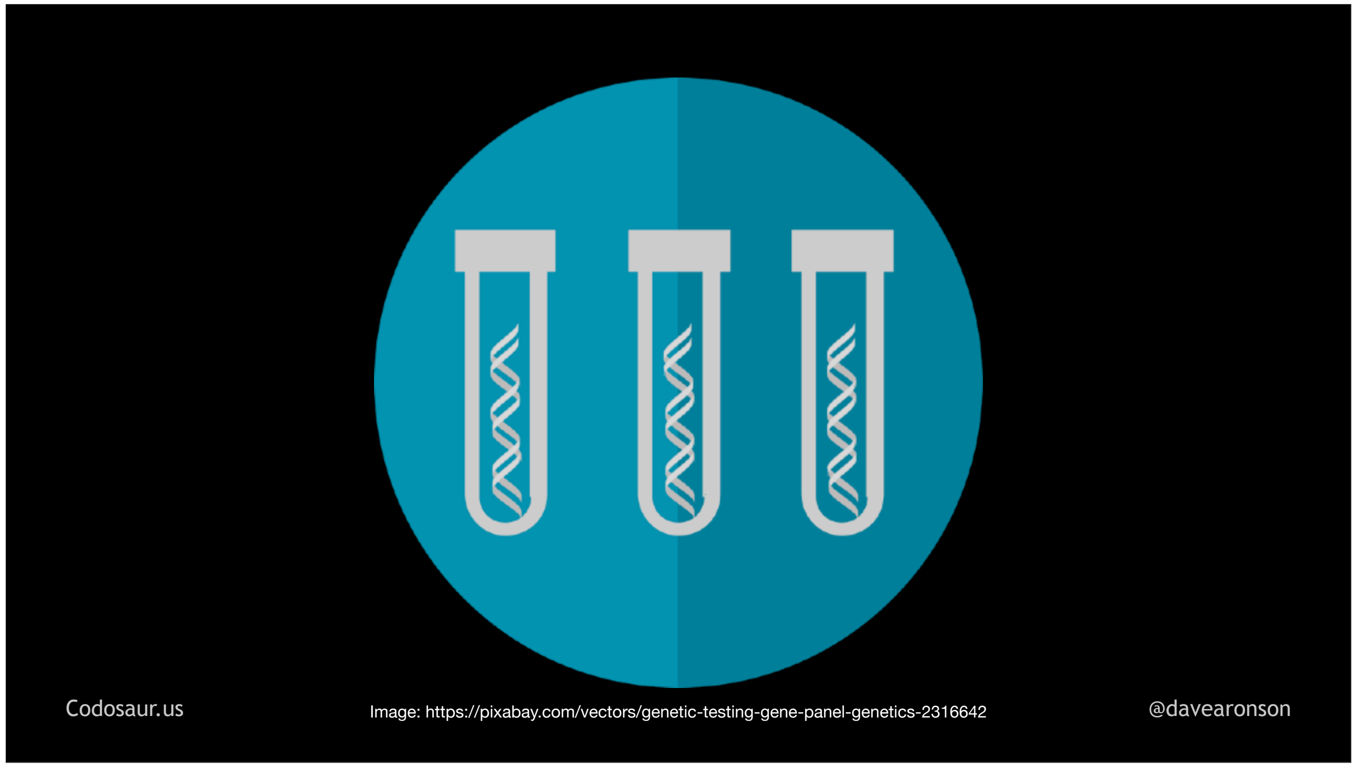
auf EN-glish. (PAUSE!)

Mainly because you've just heard about half the German I speak! (PAUSE!)

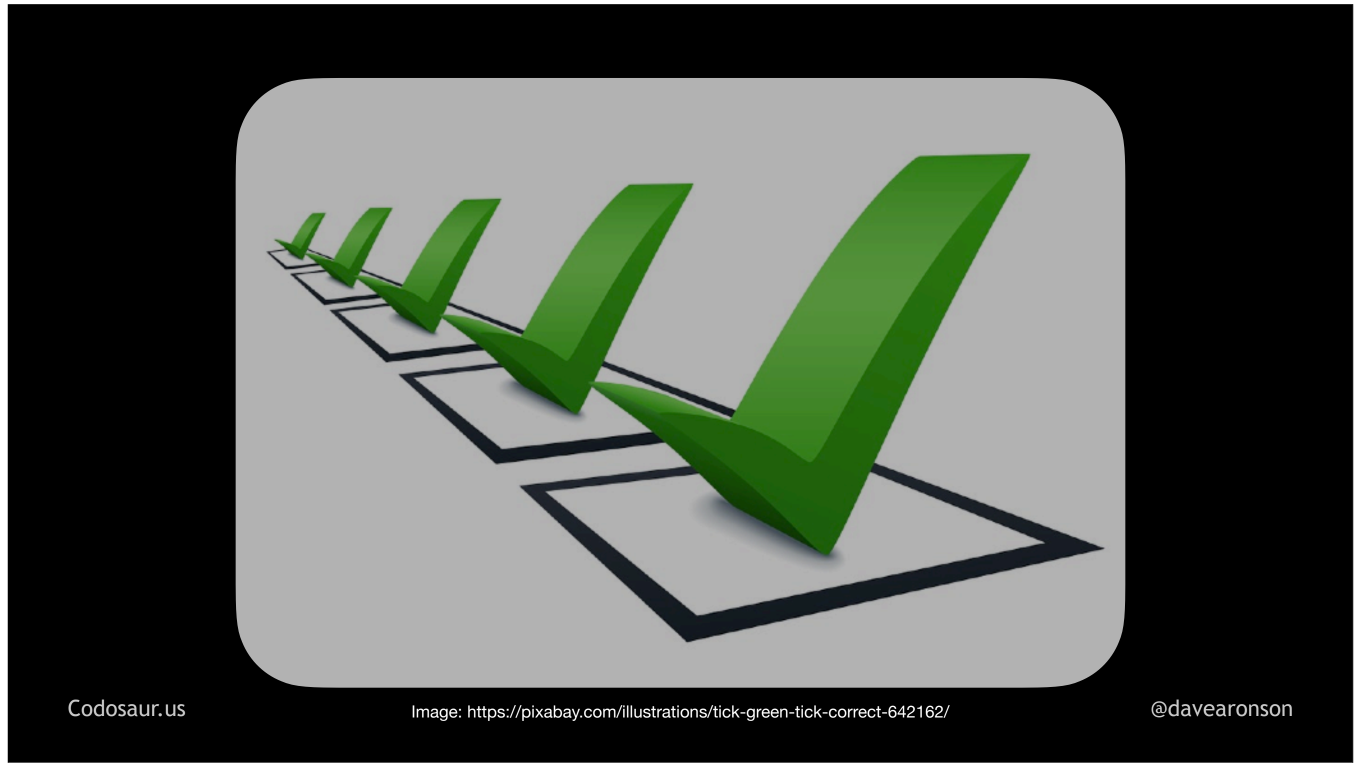
Let's start with the basics. What on . . .



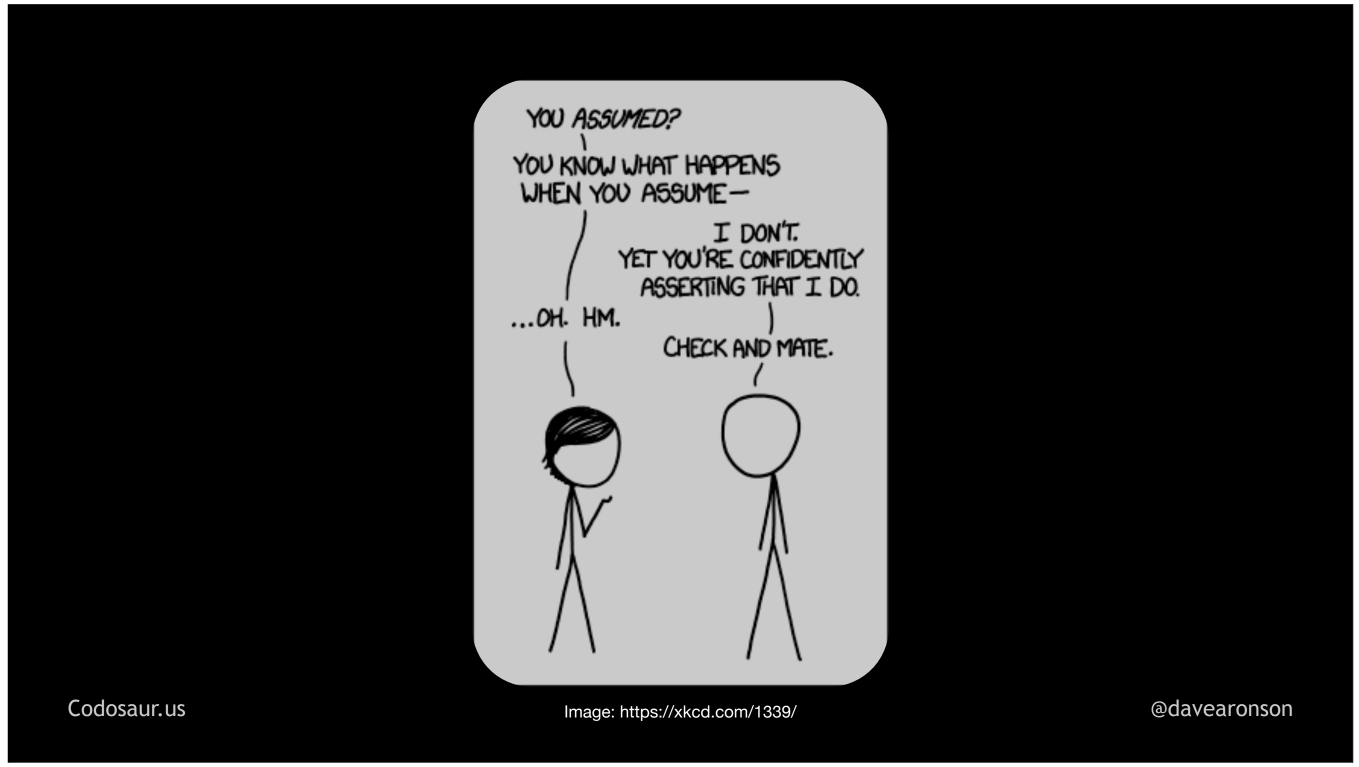
. . . Infinite Earths, makes . . .



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



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

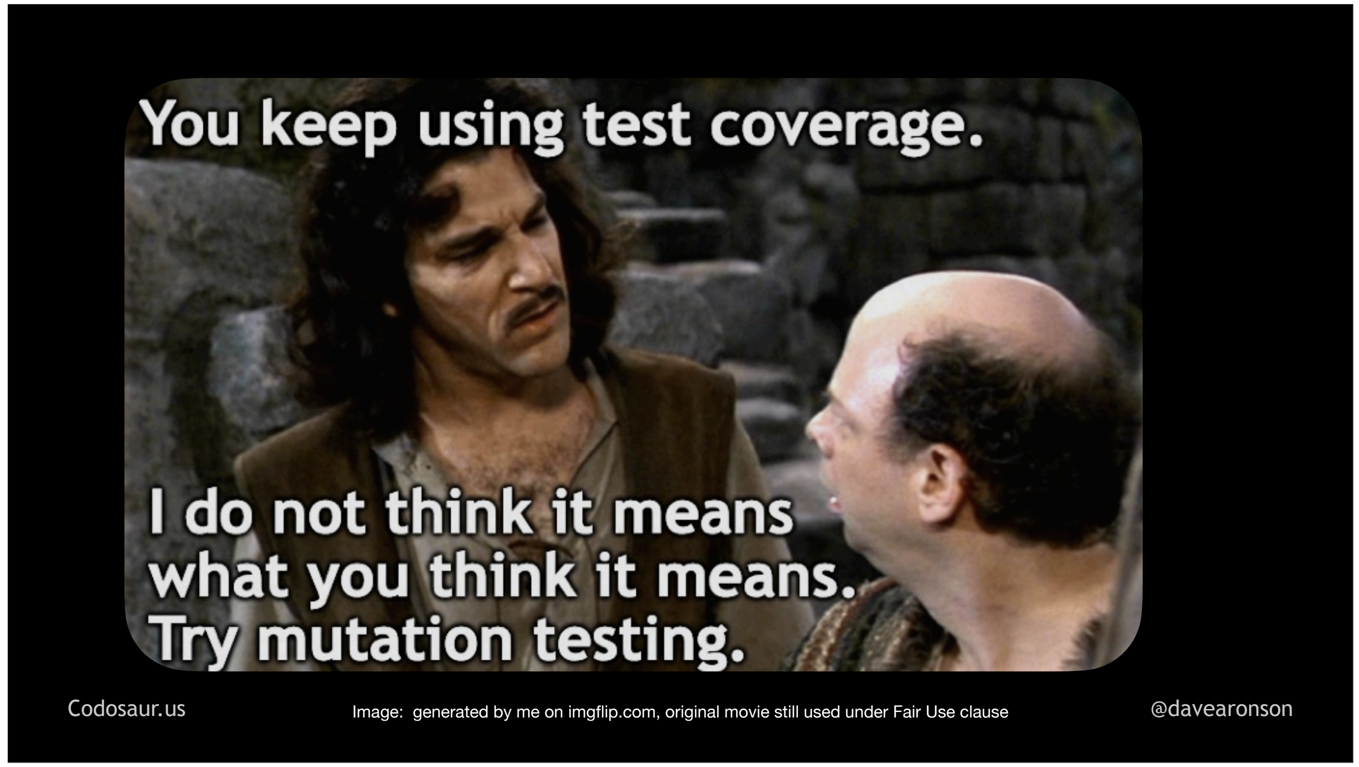


. . . *assumes* that our code is correct, at least in the sense of passing its tests. Instead, mutation testing checks *two other* qualities. In a typical codebase, I think the more *important* one is that our test suite is . . .

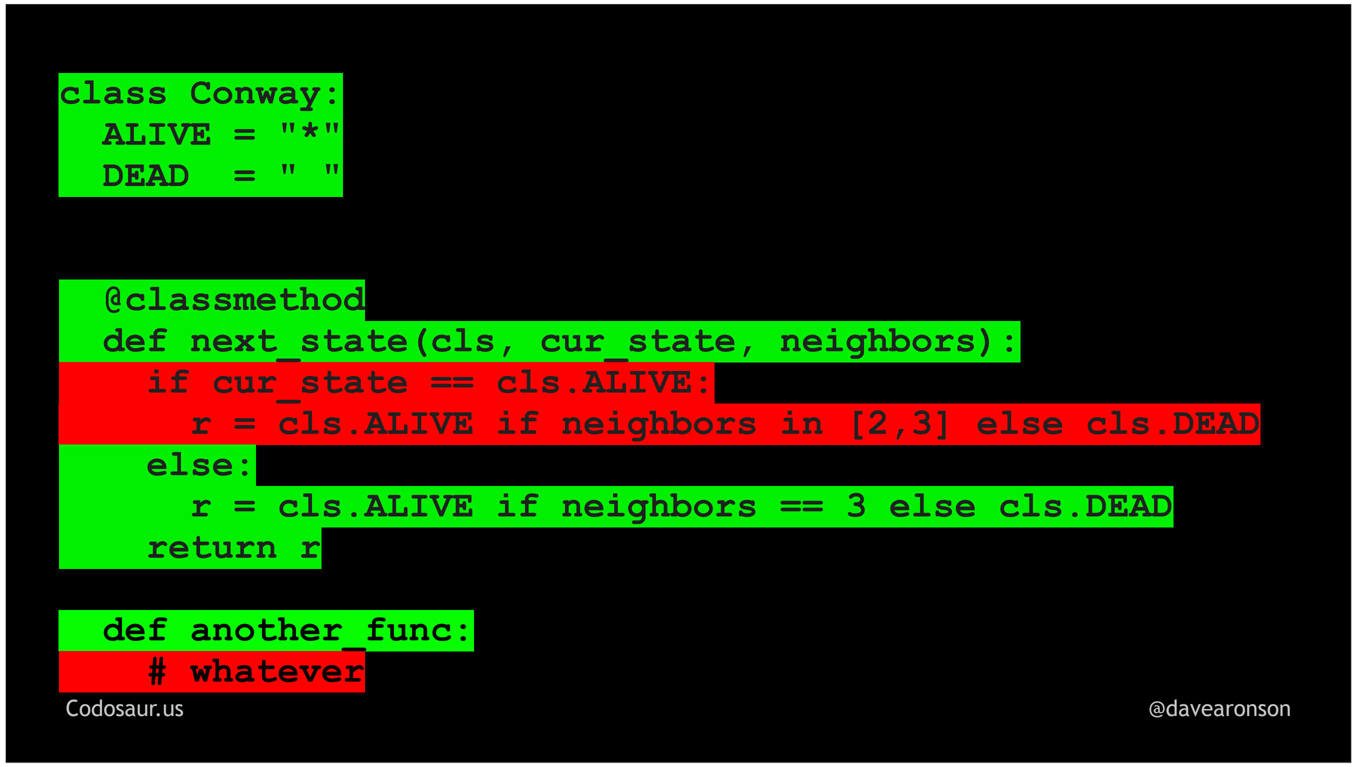
```
"use strict";
```

Codosaur.us @davearonson

. . . *strict*. Now you may be thinking, “Isn’t that what test coverage is for? If we have 100% test coverage, doesn’t that mean all our code is fully tested?”



No. (PAUSE!) The *only* thing that test coverage tells us is that at least one test *ran* . . .



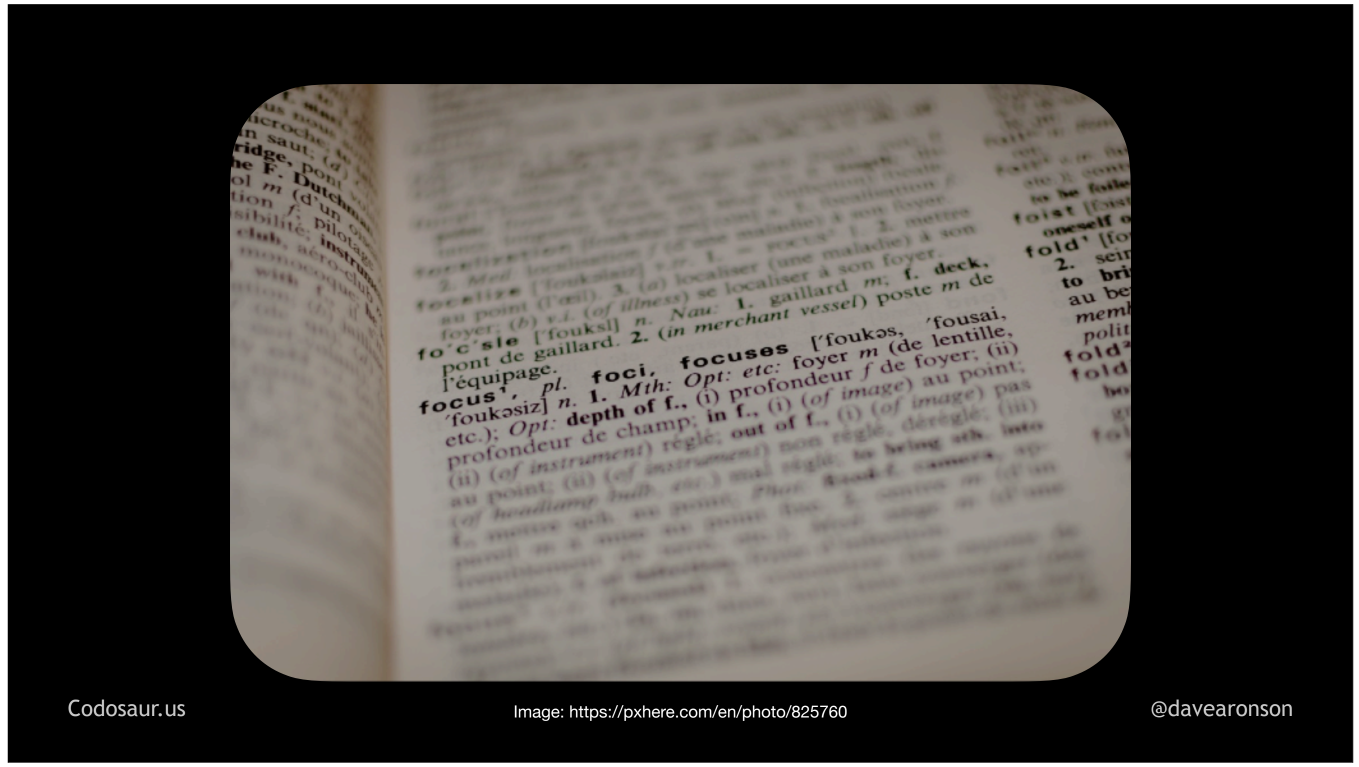
. . . the code it claims is “covered”. It tells us NOTHING about whether the *correctness* of that code *made any difference* to whether any test passed. And isn’t that what we really *mean* by “tested”?

So how can we *tell* if the code really is “tested”? As you may have guessed, that . . . is where mutation testing comes in.

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

. . . 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 . . .



. . . *meaningful*, so that any tiny little semantic change to the code (versus structural or syntactic changes), 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 . . .

. . . 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 indeed notice that change, and fail.

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

Fred Brooks, author of "No Silver Bullet — Essence and Accident in Software Engineering" (1986 paper)

Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:Frederick_Brooks_IMG_2279.jpg

@davearonson

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

Codosaur.us

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

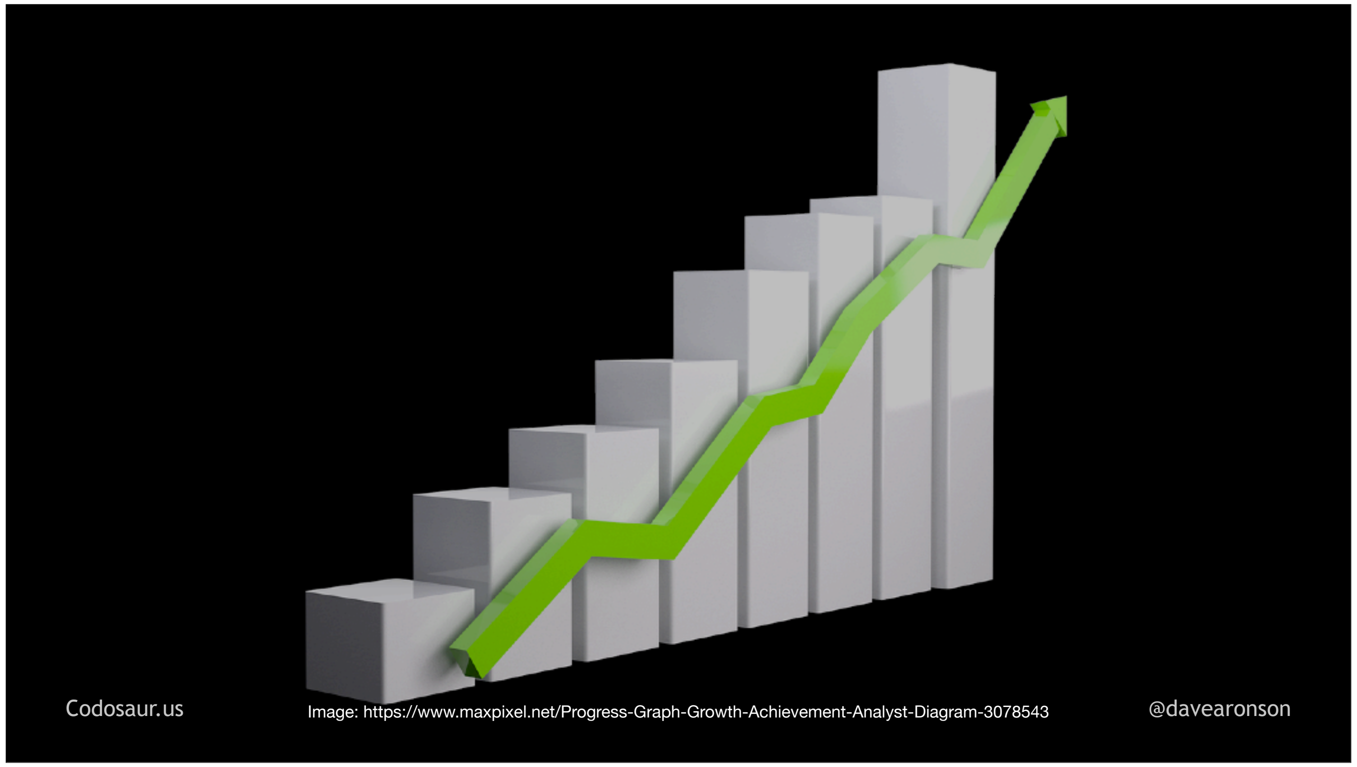
@davearonson

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

. . . werewolves, not mutants!

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

. . . hard labor on the CPU, and therefore usually ra-ther sloooow. We certainly won’t want to mutation-test our entire 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 . . .

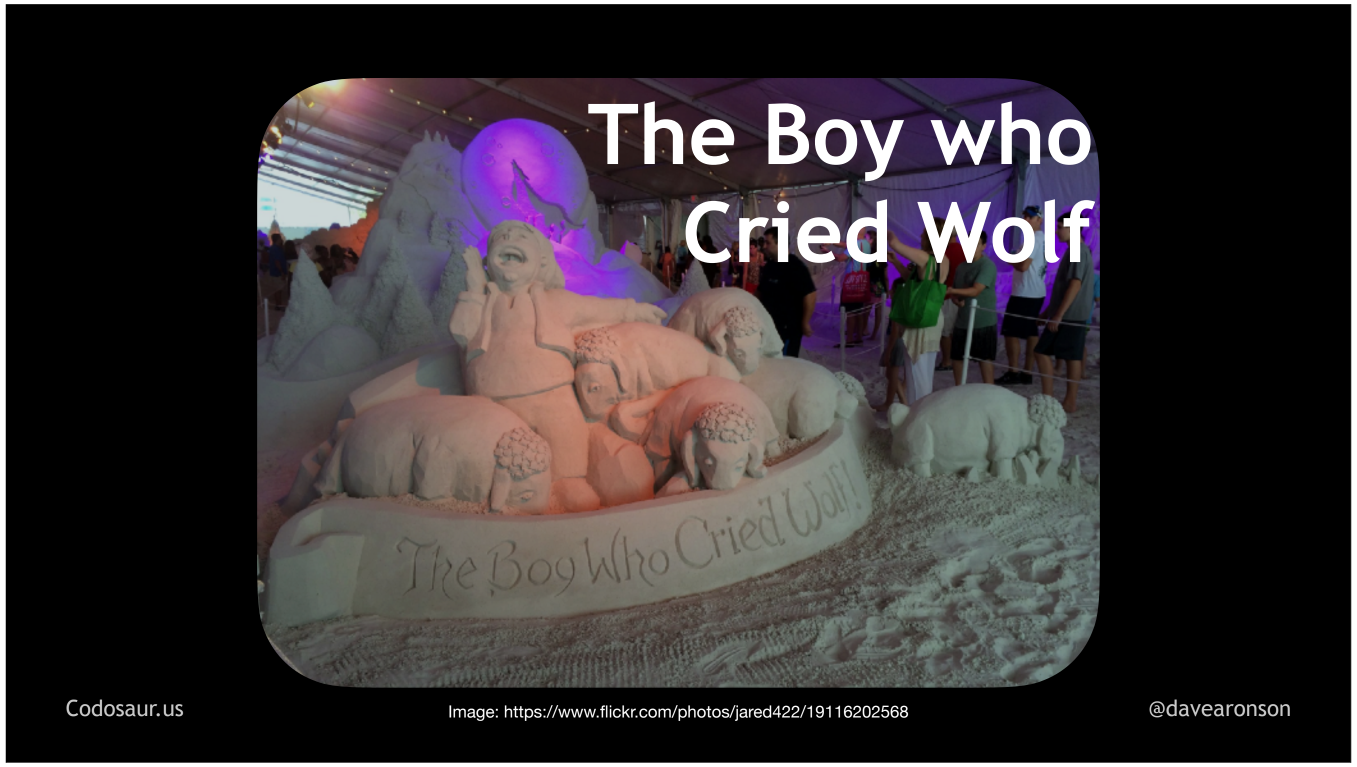


. . . incremental mode, so that we can test only the changes since the last mutation test, or the last git commit, or the main branch, or some such difference. With such filtering, maybe we can test just the relevant changes on each save, or at least over a much shorter break.

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. 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 . . .

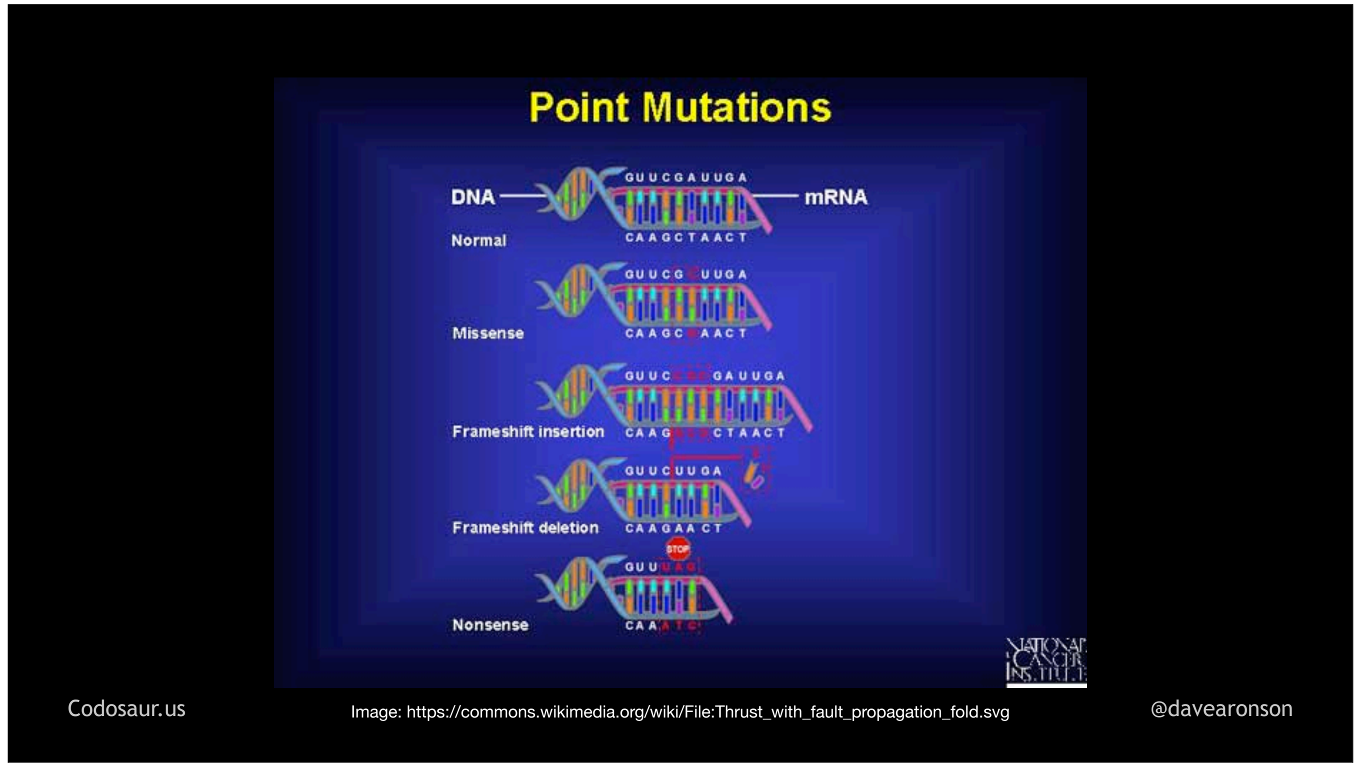


. . . false alarm, because the mutation didn't make a test fail, but it didn't make any behavioral difference in the first place. It can still take quite a lot of time and effort to figure *that* out.

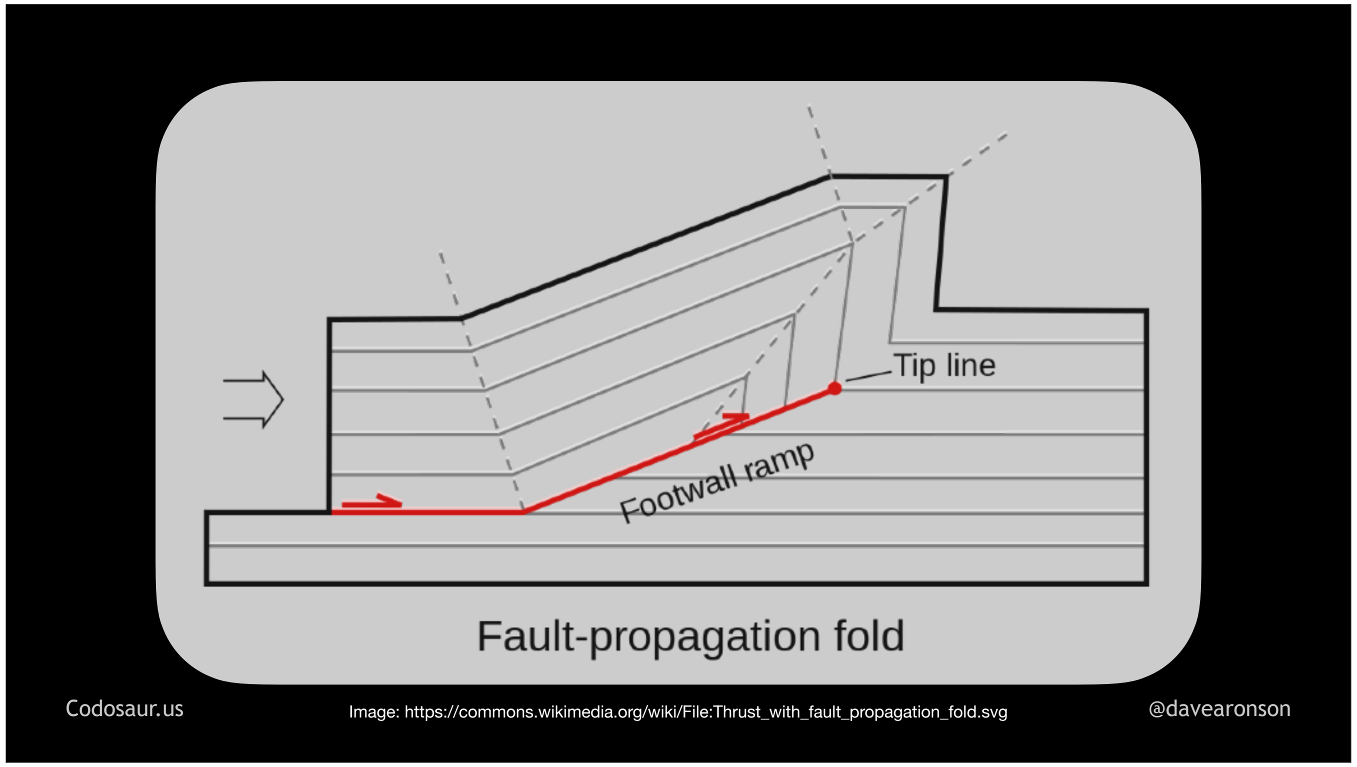
And even if a mutation *does* make a difference, most programs have quite a lot of code that we just . . .

. . . *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 some of the pros and cons, how does mutation testing work, unlike this guy? It . . .



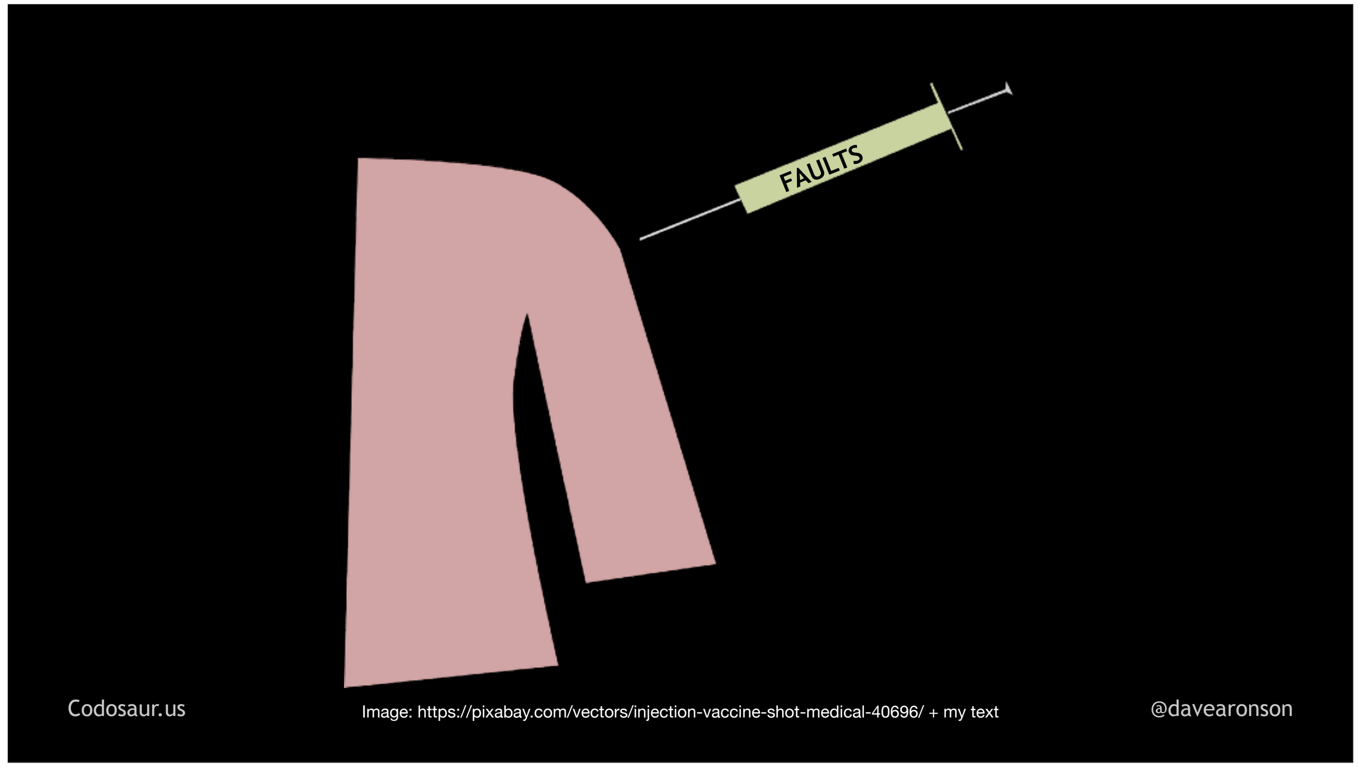
. . . *mutates* copies of our code, hence the name. It does this with the intent to create test failures, also known as . . .



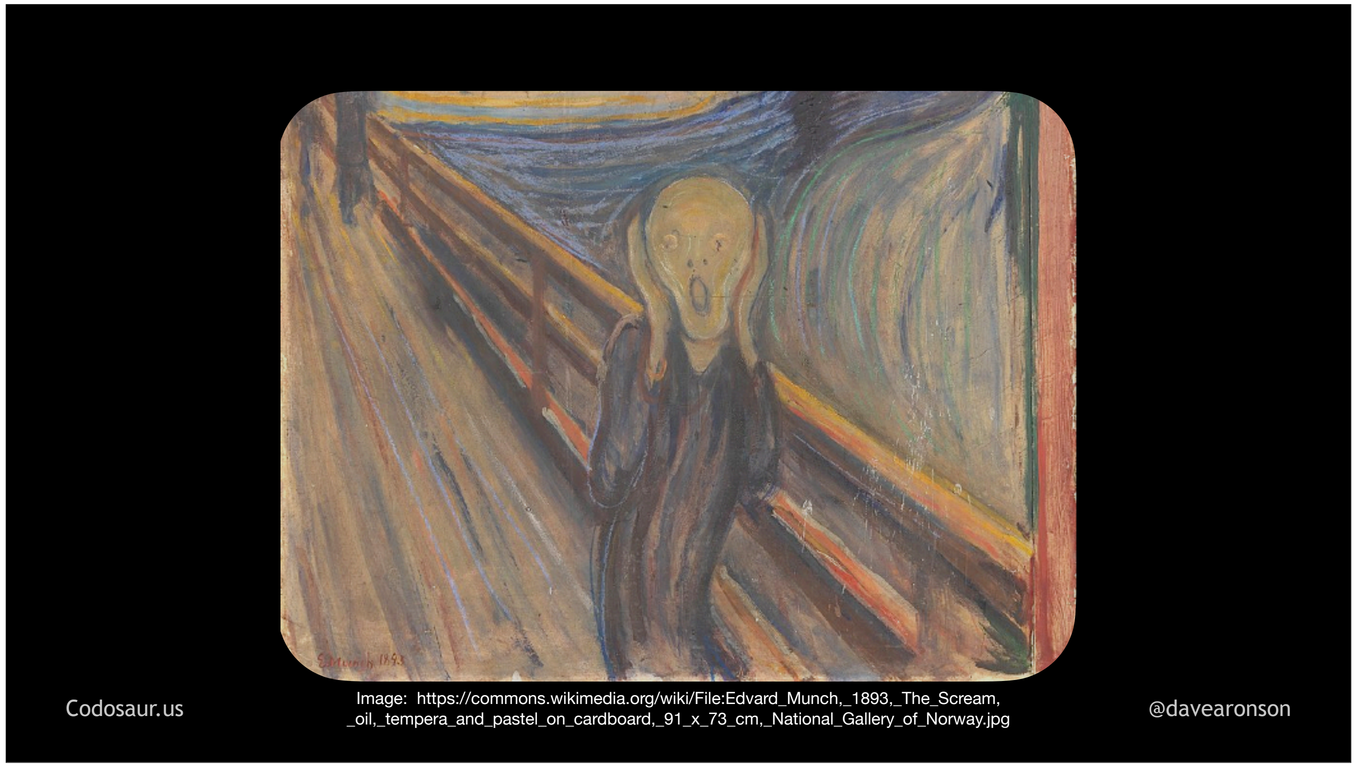
. . . faults. So, mutation testing can be categorized as a *"fault-based"* testing technique. This means that it is related to something you might already be familiar with:

. . . 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 . . .

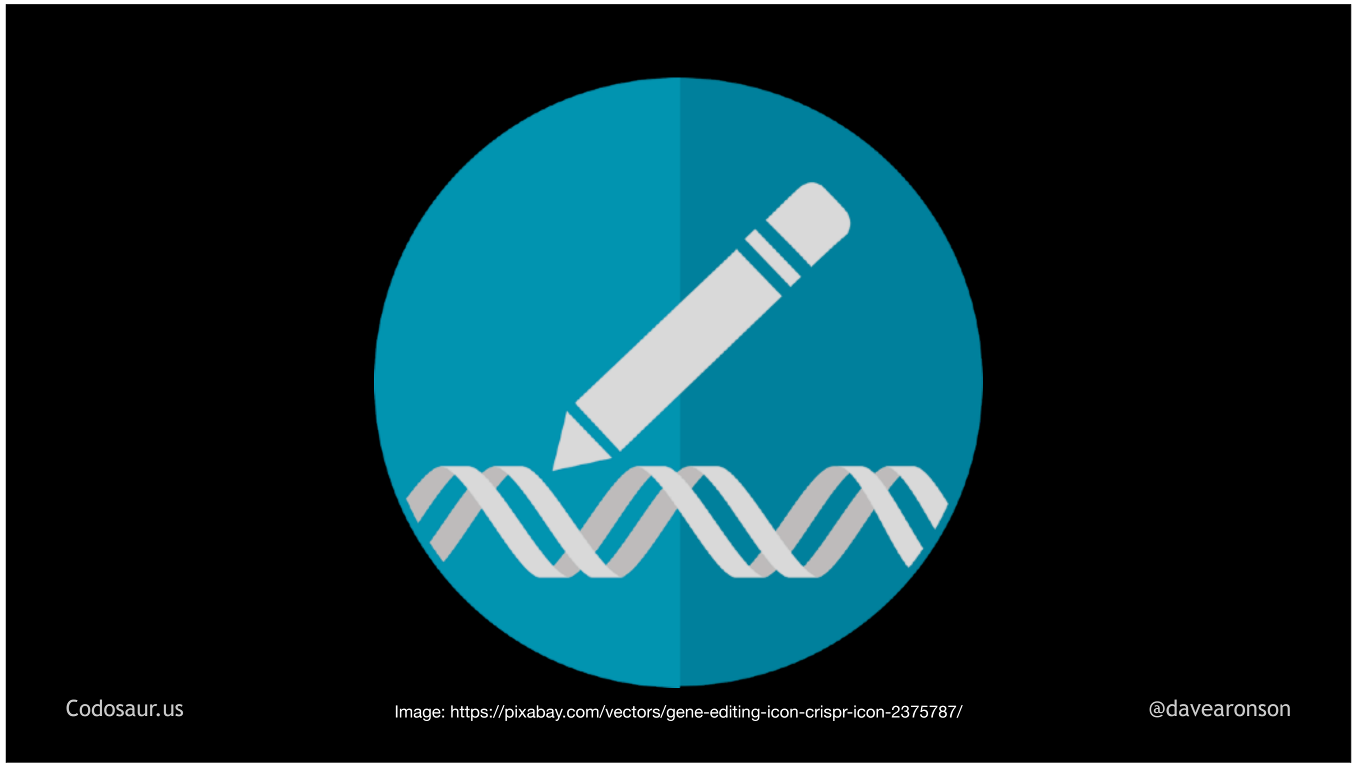


. . . injecting faults, such as latency, jitter, and dropped connections, into Netflix's production network. (QUICK-CUT TO NEXT SLIDE!)



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 . . .

. . . *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, and does this in our . . .

. . . *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
```


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

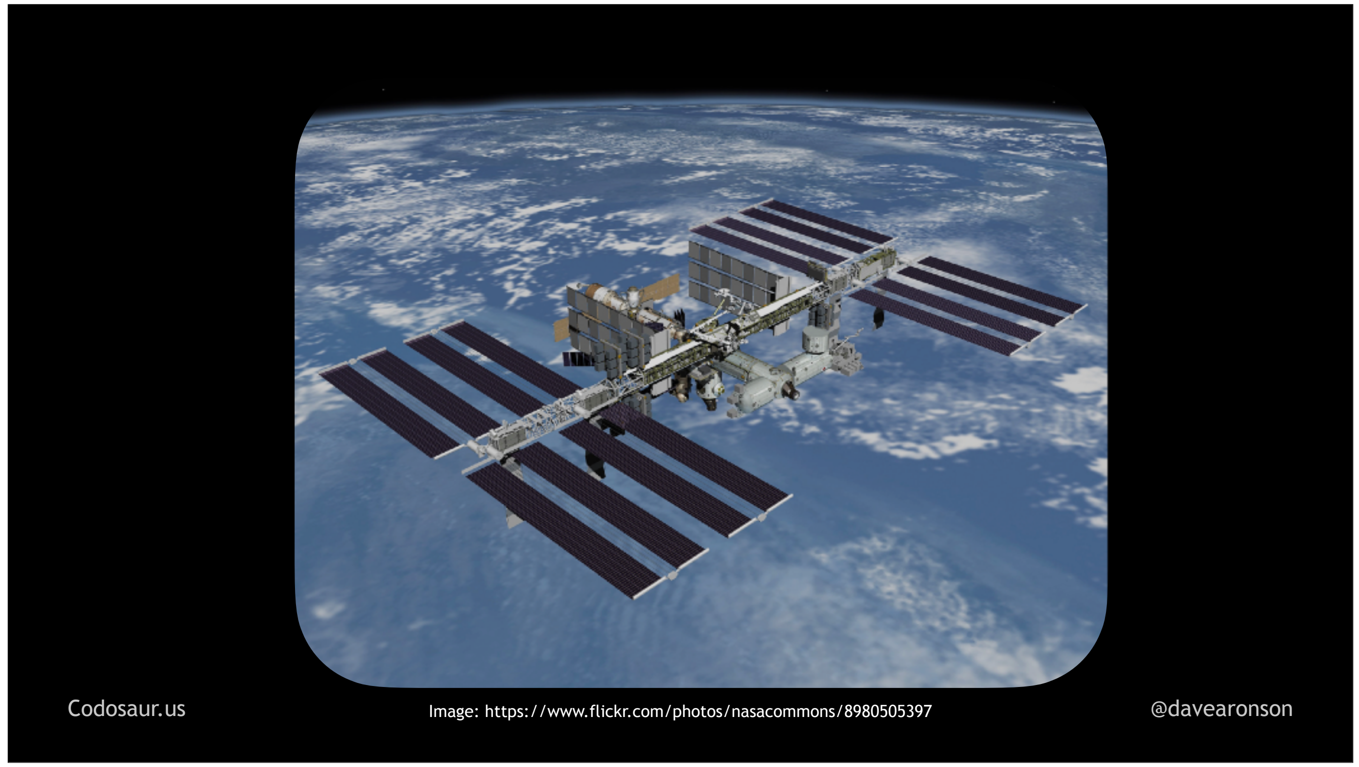
. . . there *is* a problem! Remember, each change to our code should make *at least* one test *fail*.

Mutation testing has also been compared to . . .

. . . fuzzing, a security penetration technique involving throwing random data at an application. Mutation testing is somewhat like fuzzing our *code* rather than fuzzing the *data*, but it's . . .

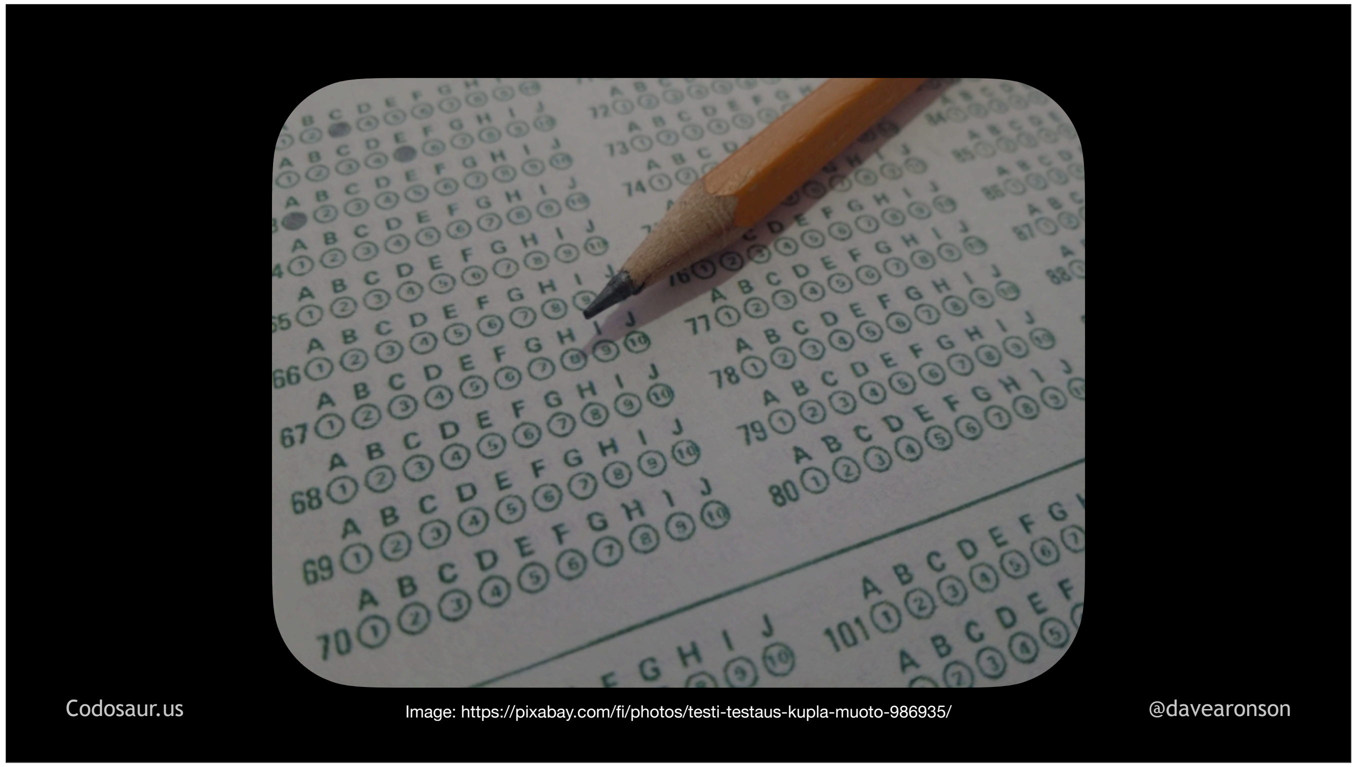
. . . not random. These tools have a set of mutations they know how to do. The smarter ones can use the results of simpler mutations, to know they don't need to bother with more complex ones, so it may sometimes do different things and therefore *look* random, but it's not.

But enough about differences. What exactly does mutation testing *do*, and how? Let's start with . . .



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

. . . breaks our code apart into pieces to test. Usually, these are our functions -- or methods if we're doing *object-oriented* programming, but I'm just going to say functions. Then, for each function, it tries to find . . .



. . . the *tests* that cover that function. If the tool can't find any applicable tests, most will simply skip this function. Better yet, most those of them will warn us, so we know we need to add or maybe annotate some tests. Some, though, will use the whole test suite, which is horribly inefficient, because the vast majority of the tests are almost certainly not relevant to this particular function.

Assuming we aren't skipping this function, next the tool . . .

. . . makes 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 . . .

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

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

. . . 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

. . . 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

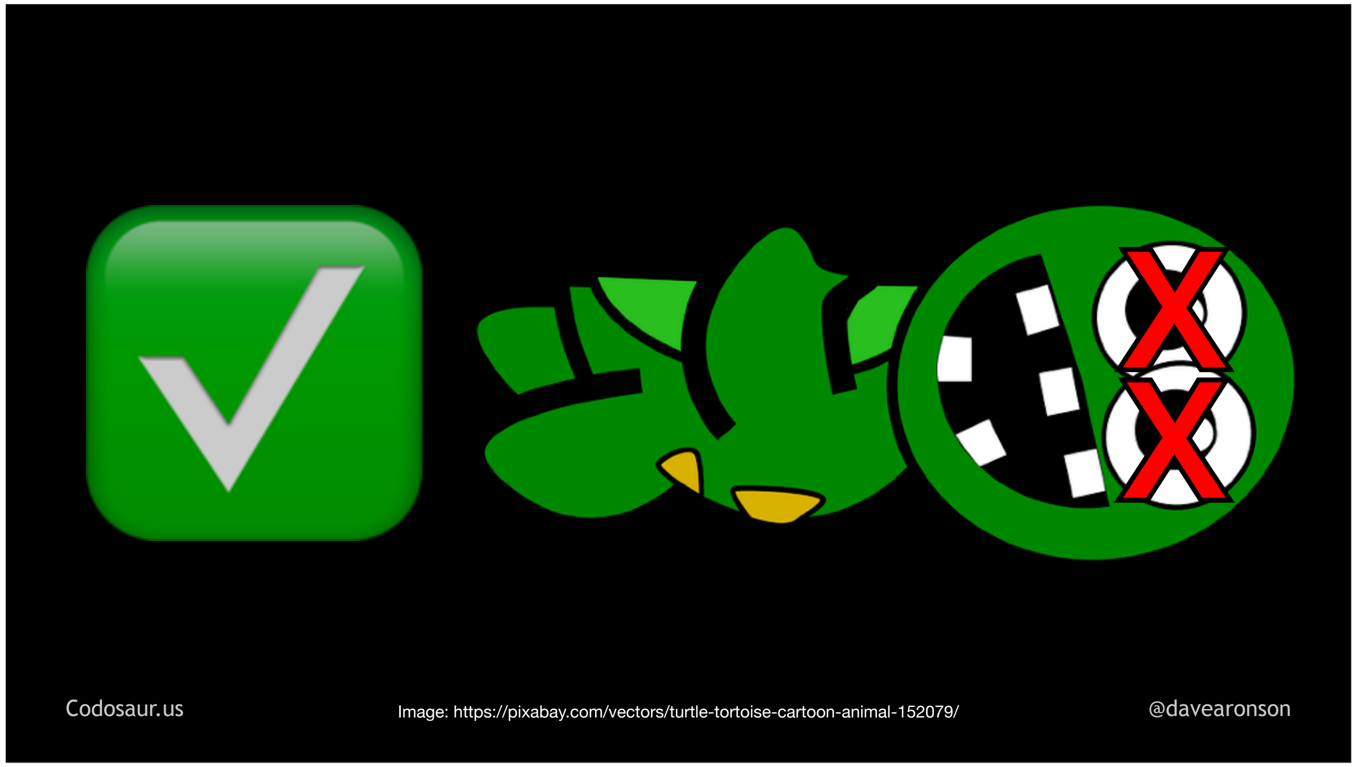
. . . using the *current mutant* in place of the original function.

(PAUSE) If any test . . .

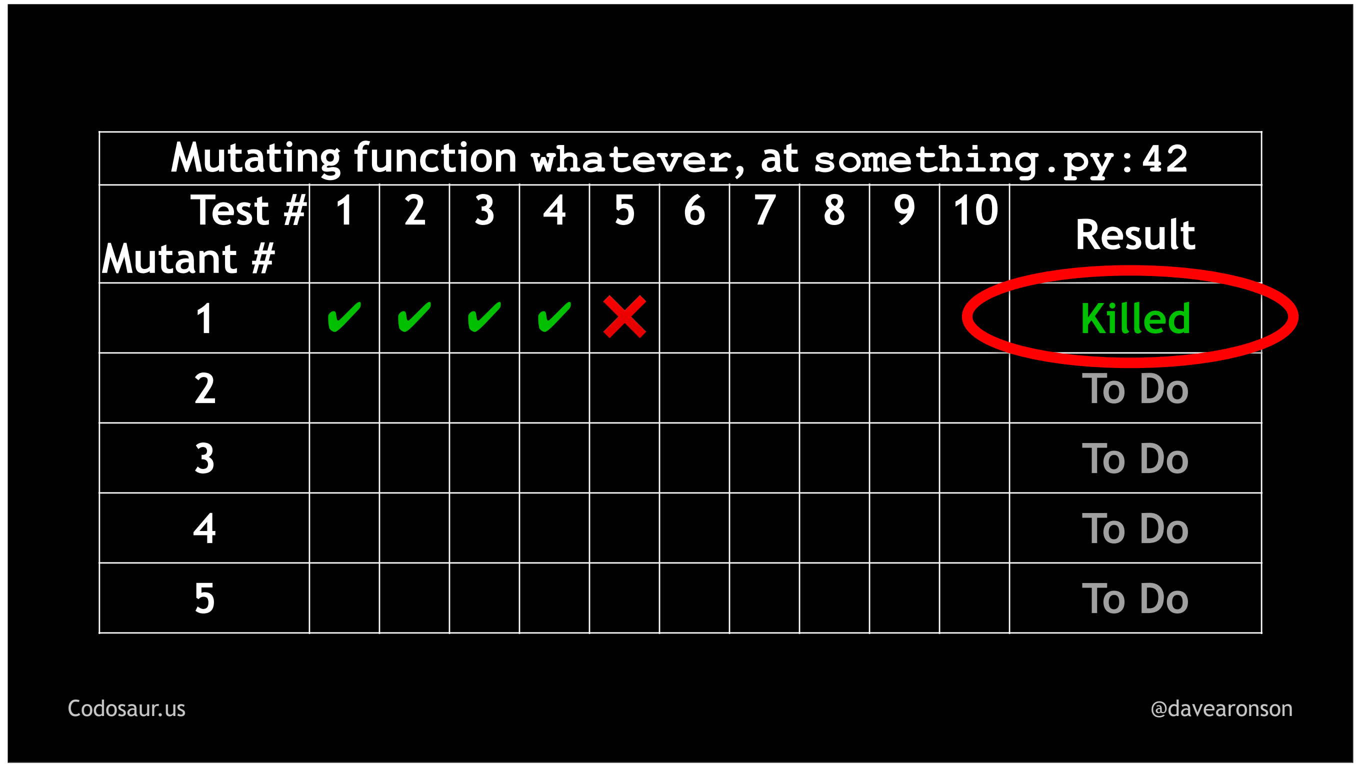
Mutating function **whatever**, at **something.py:42**											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

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

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



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

. . . 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 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 . . .

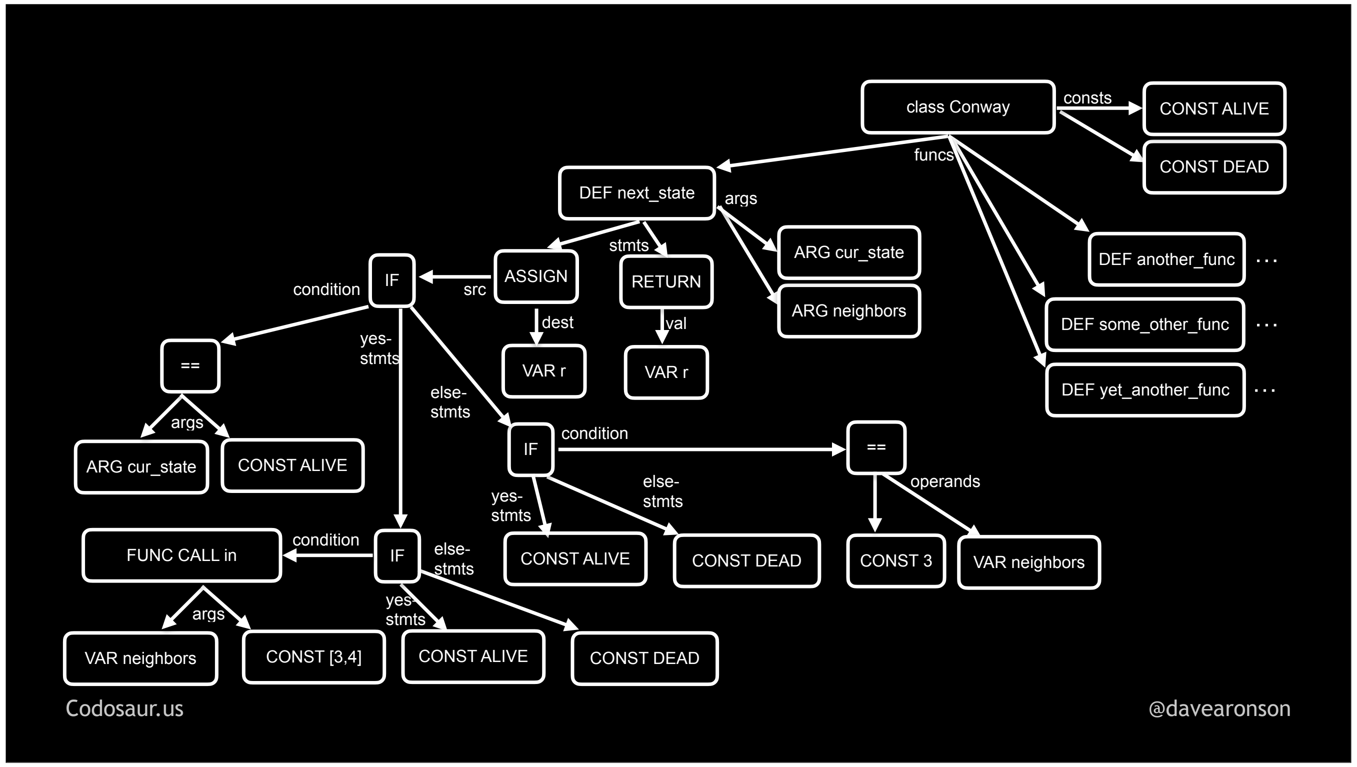
. . . superpower of mimicry, skilled enough to *fool our tests!* This usually means that our code is meaningless, or our tests are lax, or both — and now it's up to us to figure out how.

Now let's peel back one . . .

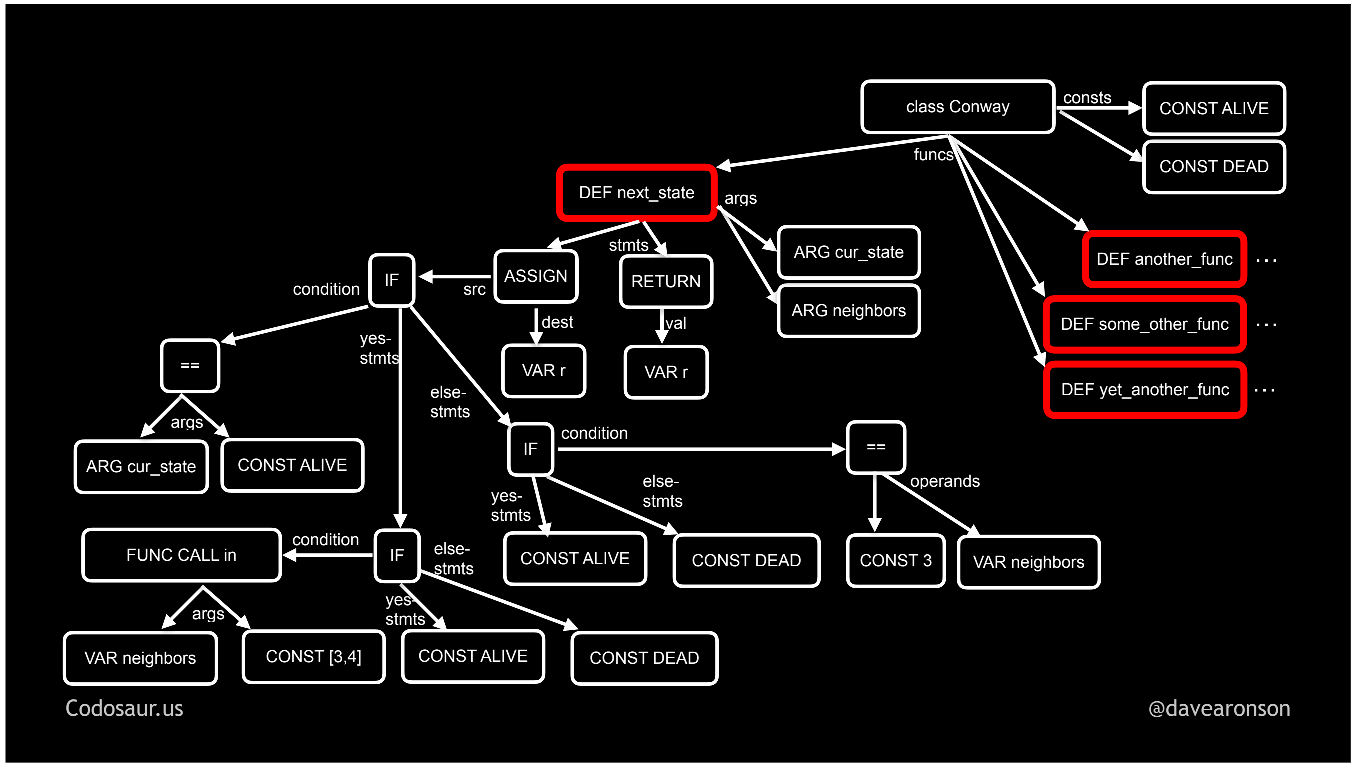
. . . layer of the onion, and look at some *technical details* of how this works. First, our tool parses . . .

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

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

  def another_func:
    # whatever
  def some_other_func:
    # whatever
  def yet_another_func:
    # whatever
```


. . . our code, usually into an Abstract Syntax Tree. So, this code, which you don't need to understand in detail, becomes . . .

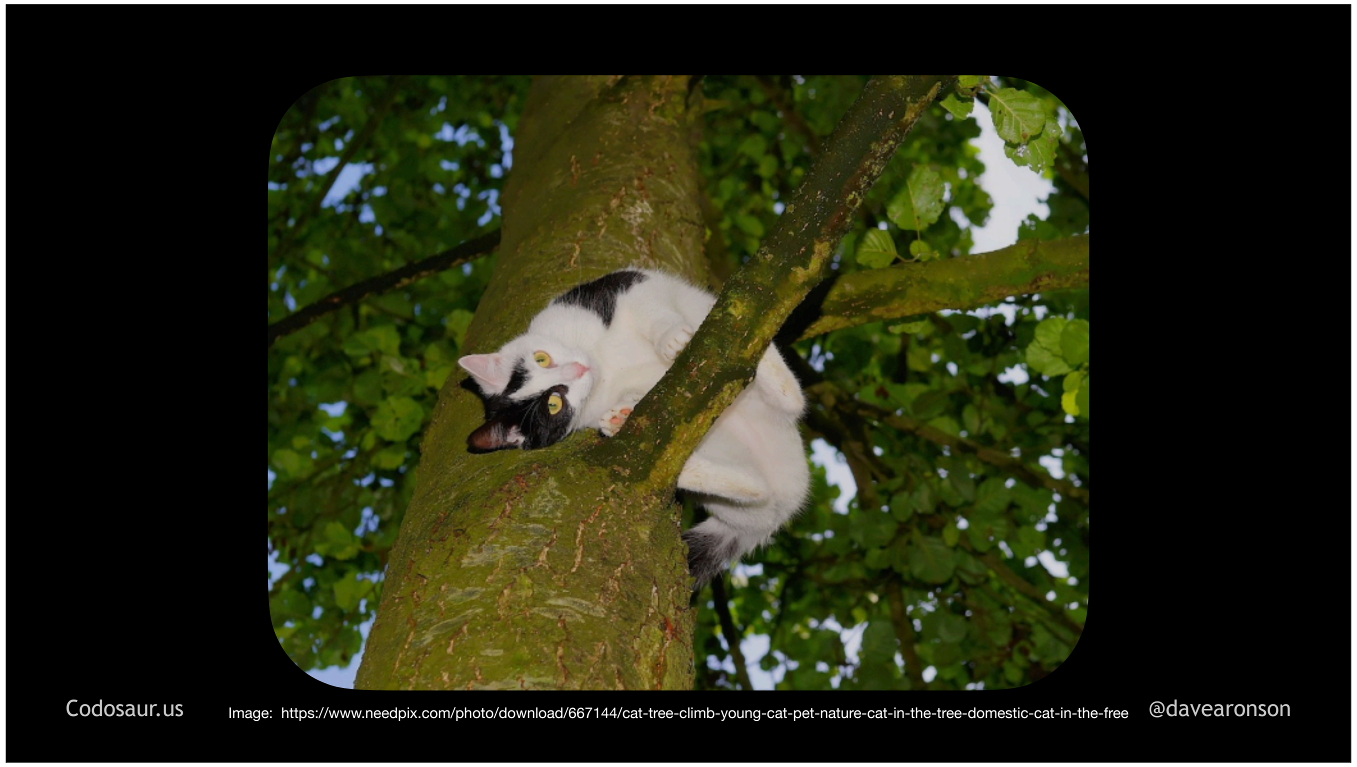


. . . this AST, which you also don’t need to understand. Just notice that there are several functions in there, rooted at . . .

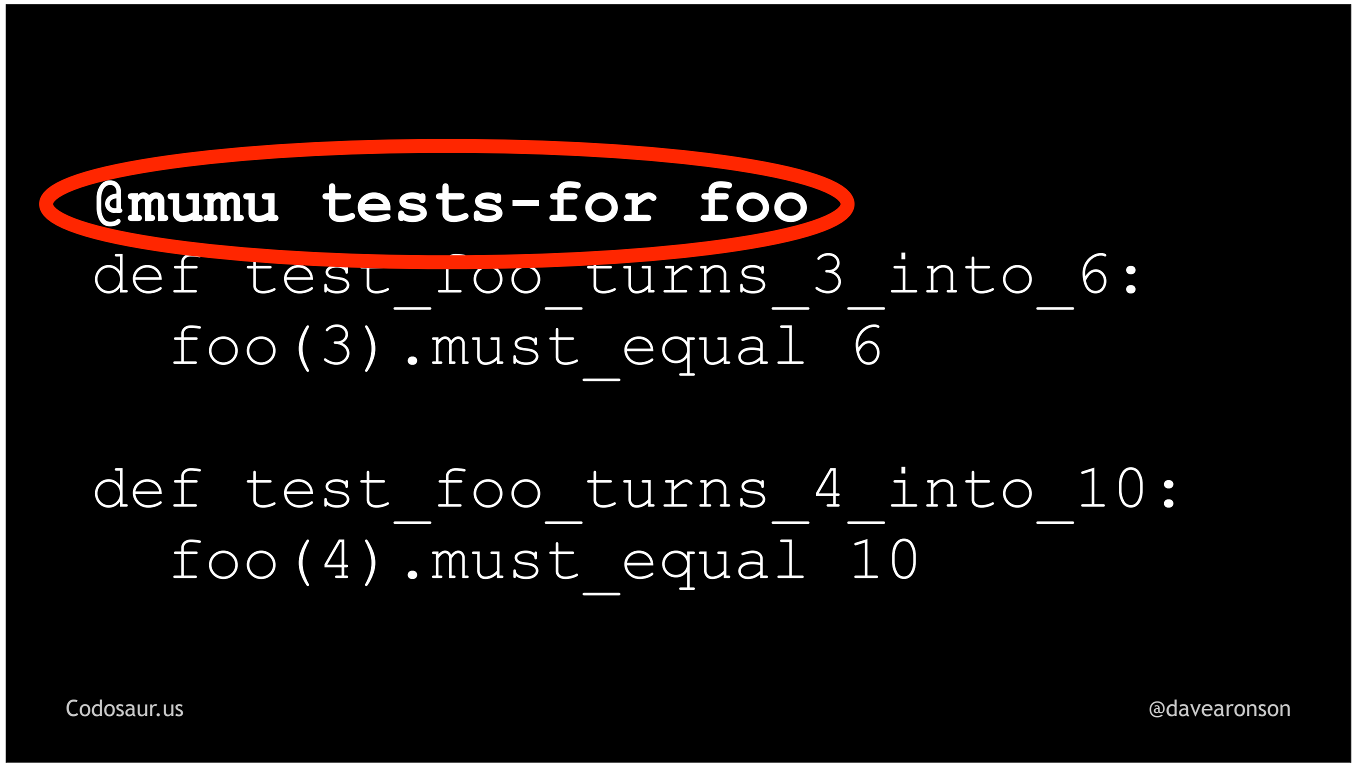


. . . the DEF nodes, and that I've fleshed out the AST subtree for one of them.

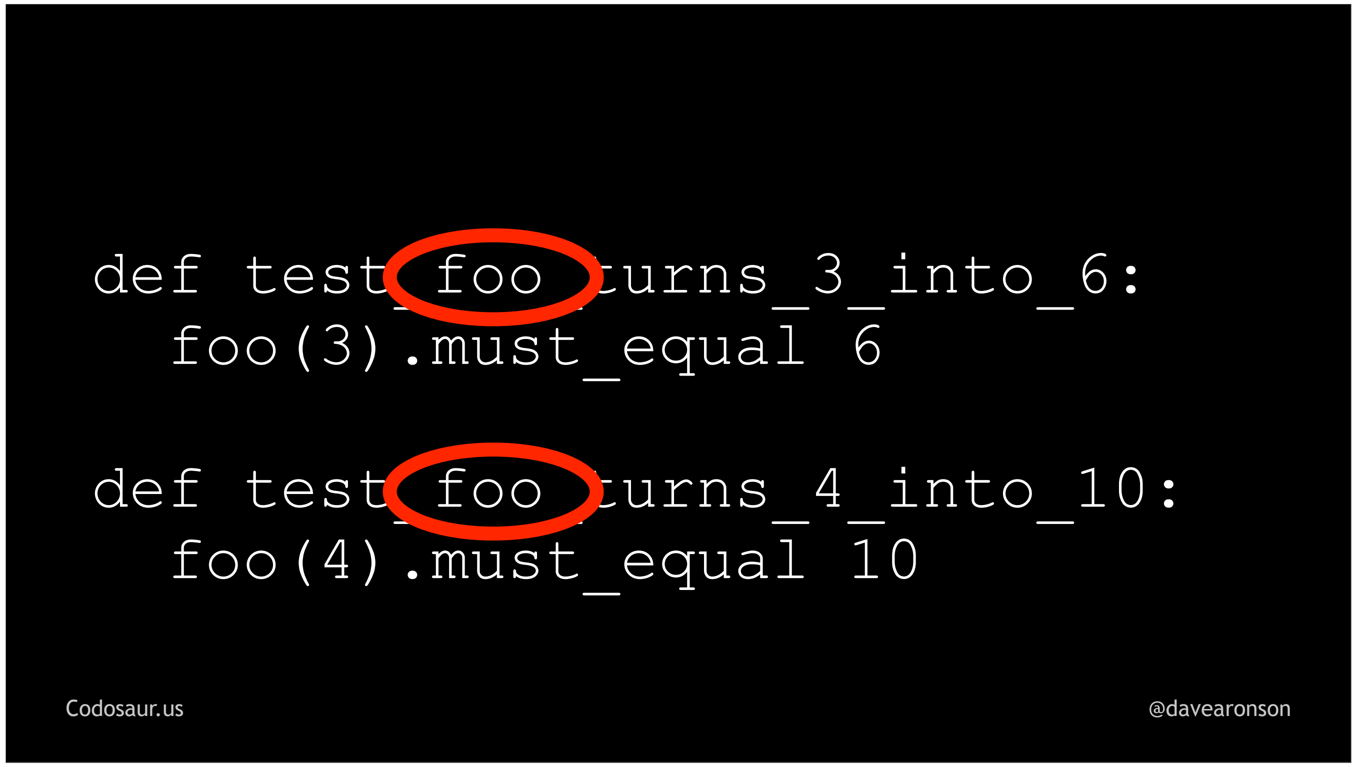
After the tool makes an AST out of our code, then it . . .



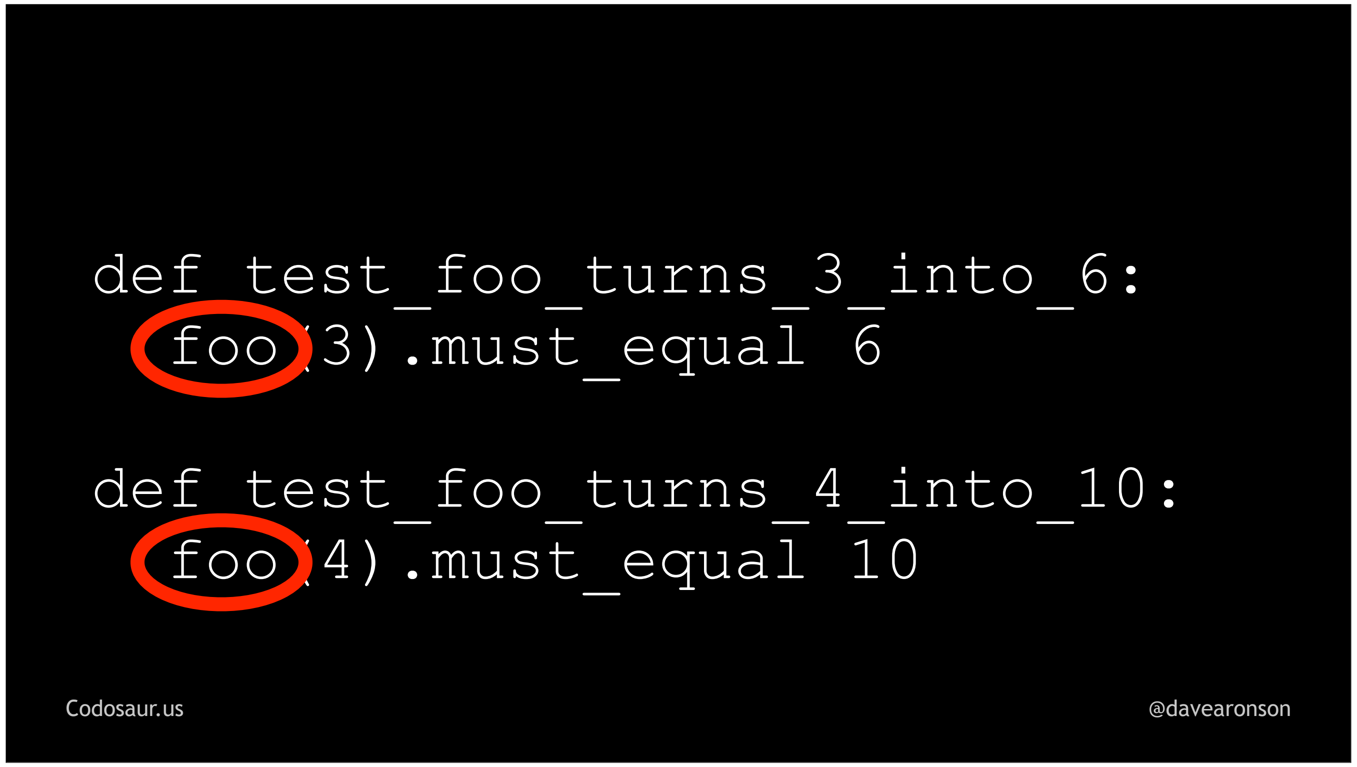
. . . traverses the tree, looking for sub-trees, or branches if you will, that represent each function. After finding *them*, it handles each one as I described before, starting with looking for each one's *tests* . . . so how does it do *that?* That usually relies mainly on us developers, either . .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



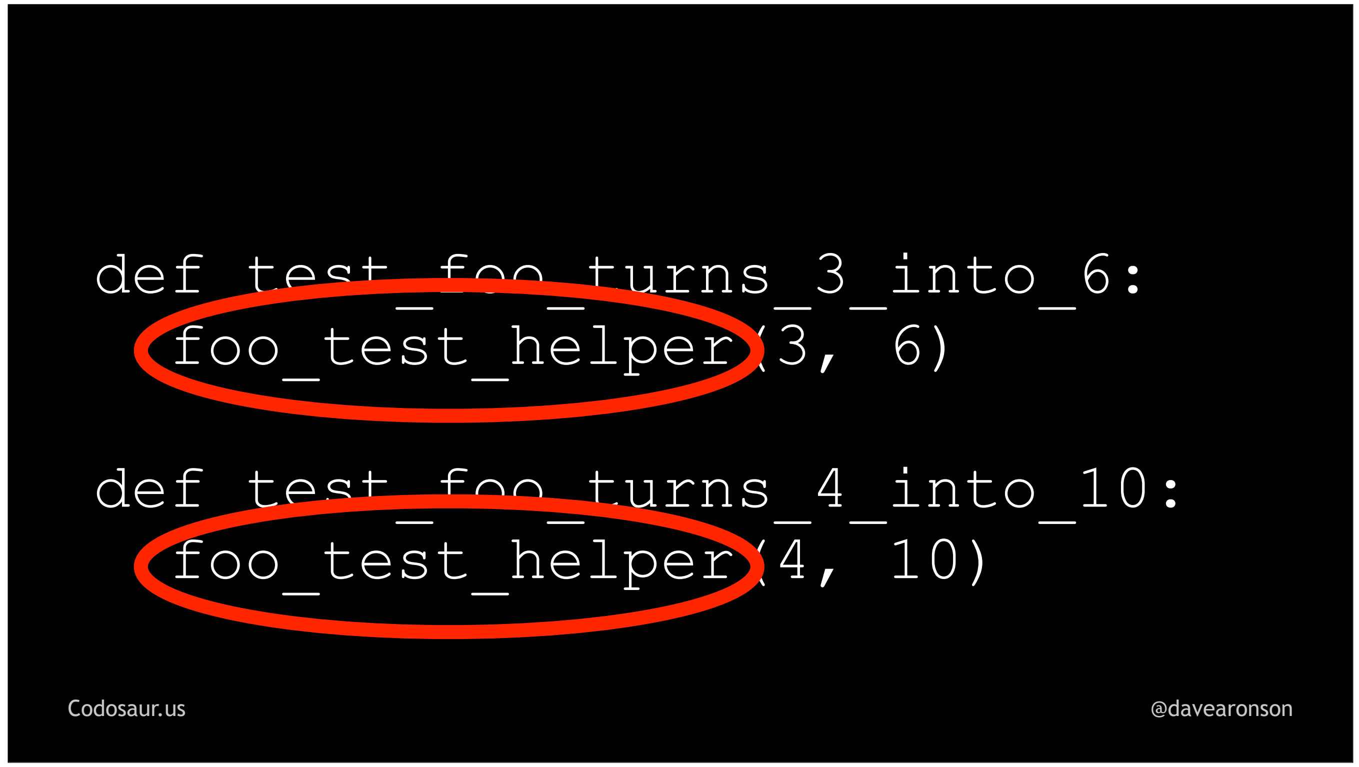
. . . annotating our tests, or following some kind of . . .



. . . convention in naming the tests, the files, or perhaps both. These manual techniques are often supplemented and sometimes even replaced by . . .

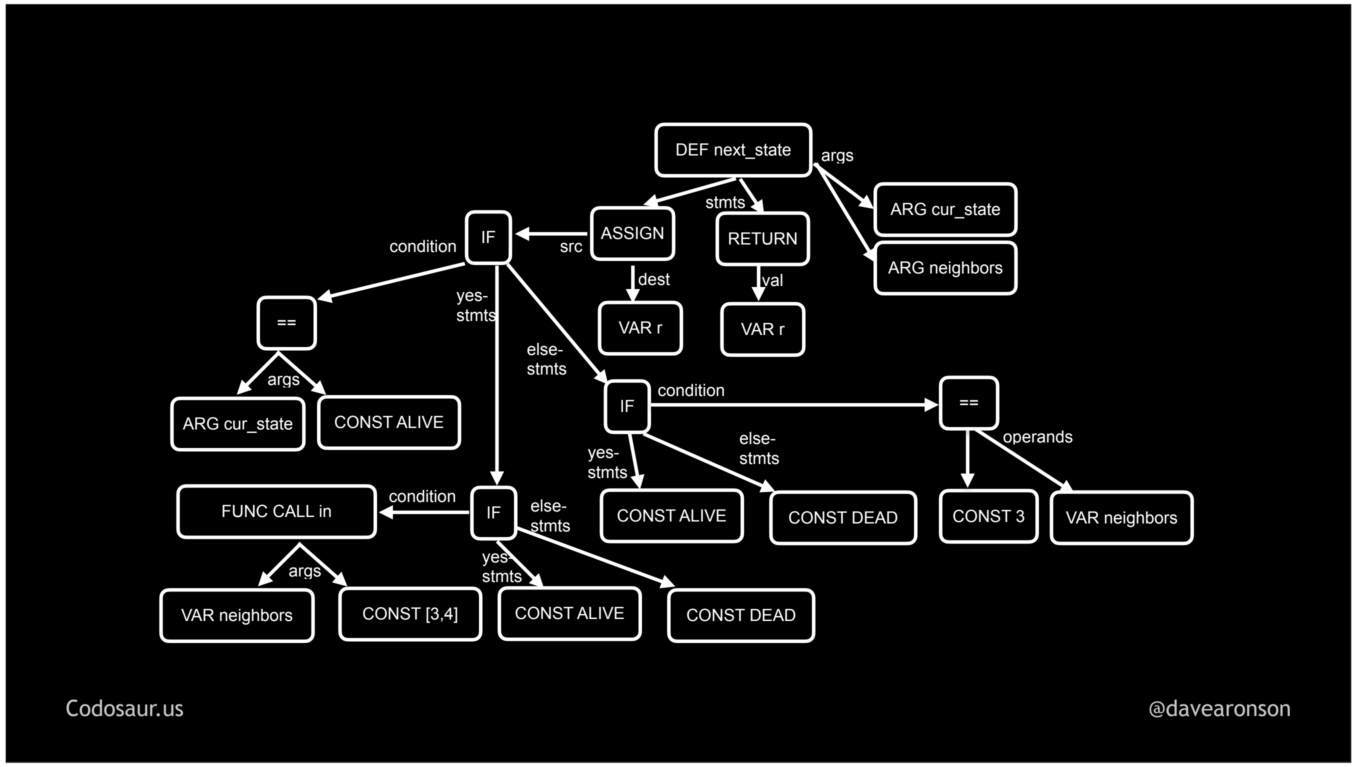


. . . the tool looking at what tests call what functions, though that can get tricky and unreliable when . . .

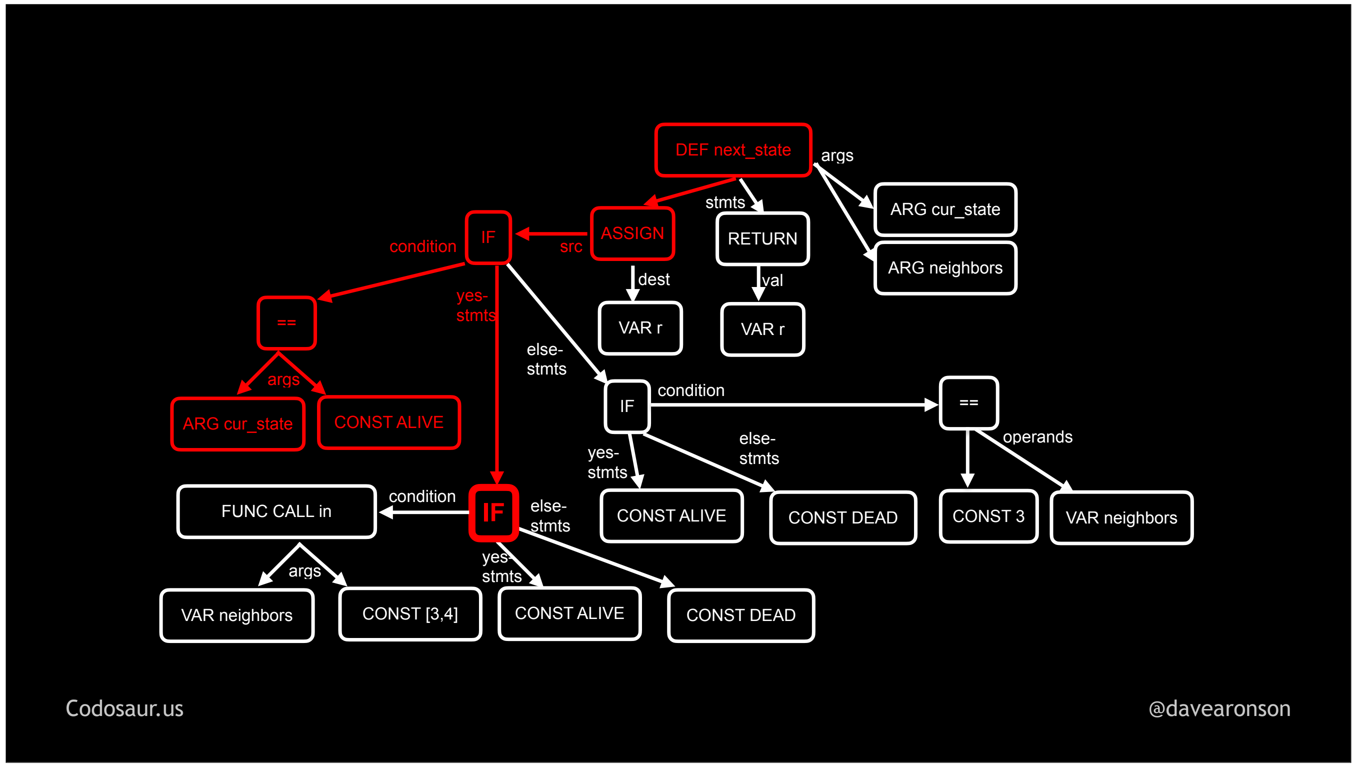


. . . the function isn’t called directly. Anyway, after the tool has found the function's tests, then, assuming it won't skip this function because it *didn't* find any tests, it makes the mutants. To make mutants *from* an AST subtree, it . . .

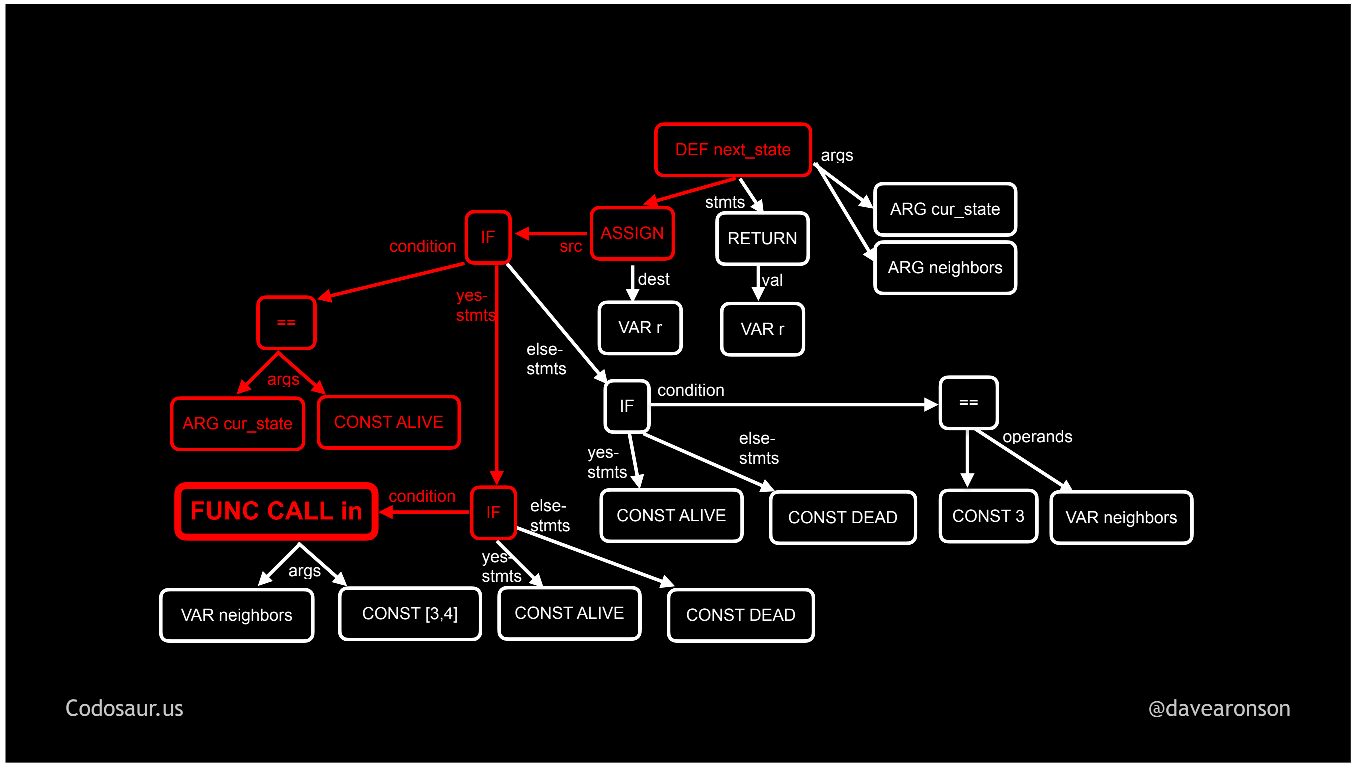
. . . traverses that subtree, just like it did to the whole thing. But now, instead of looking for even *smaller* subtrees it can *extract*, like twigs or something, it's looking for *nodes* where it can *change* something. Each time it finds one, then for each way it can change that node, it makes one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



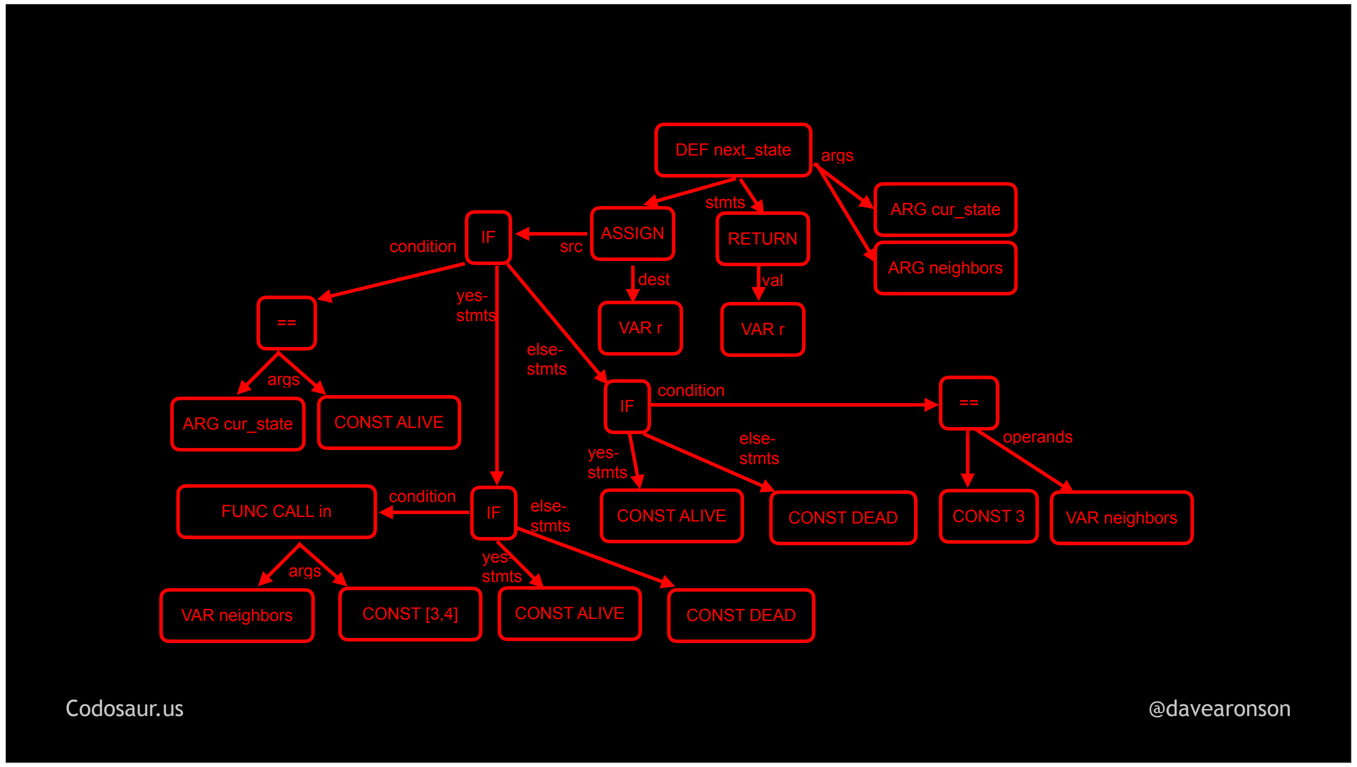
. . . the function subtree from that AST I showed earlier, and has gotten down to . . .



. . . 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 . . .

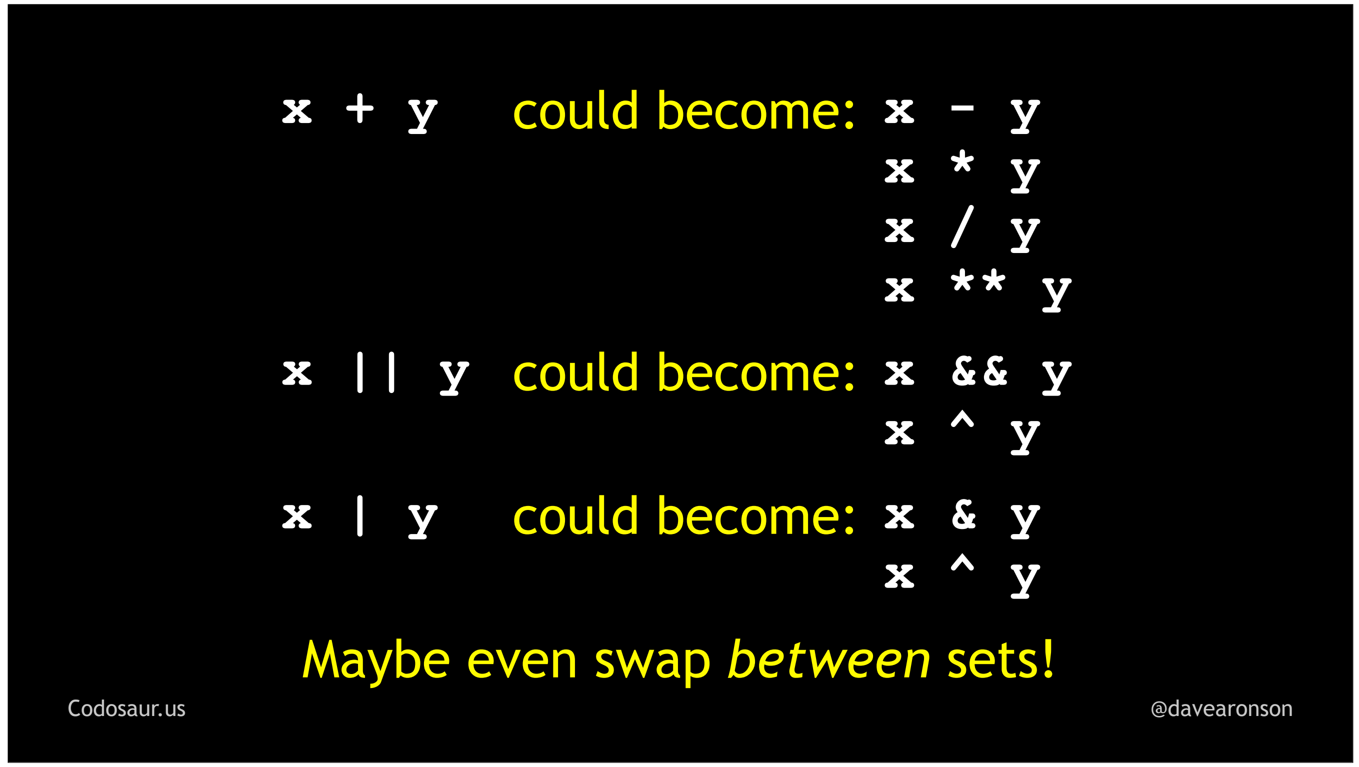


. . . the *next* node. Again, for each way it could change *that* node, it would make a copy of this whole subtree, with only *that* mutation. And so on, until it has . . .

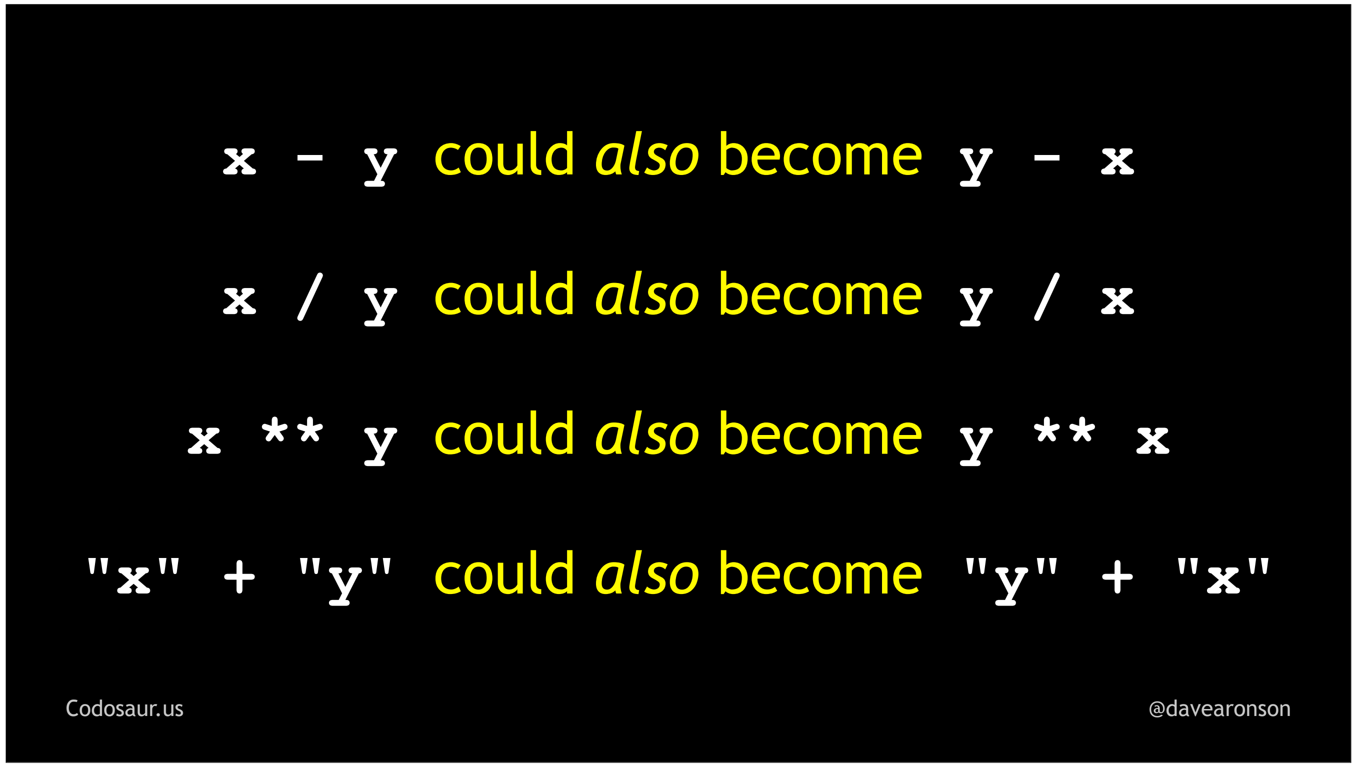


. . . traversed the entire subtree.

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



It could change a mathematical, logical, or bitwise operator from one to another. When the language allows, it could even cross these categories. For instance, in many languages, we can treat *anything* as a boolean, and all kinds of integers as bitfields, so x TIMES y could become x AND y, or x BITWISE-EXCLUSIVE-OR y, and so on.



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

```
x < y
```

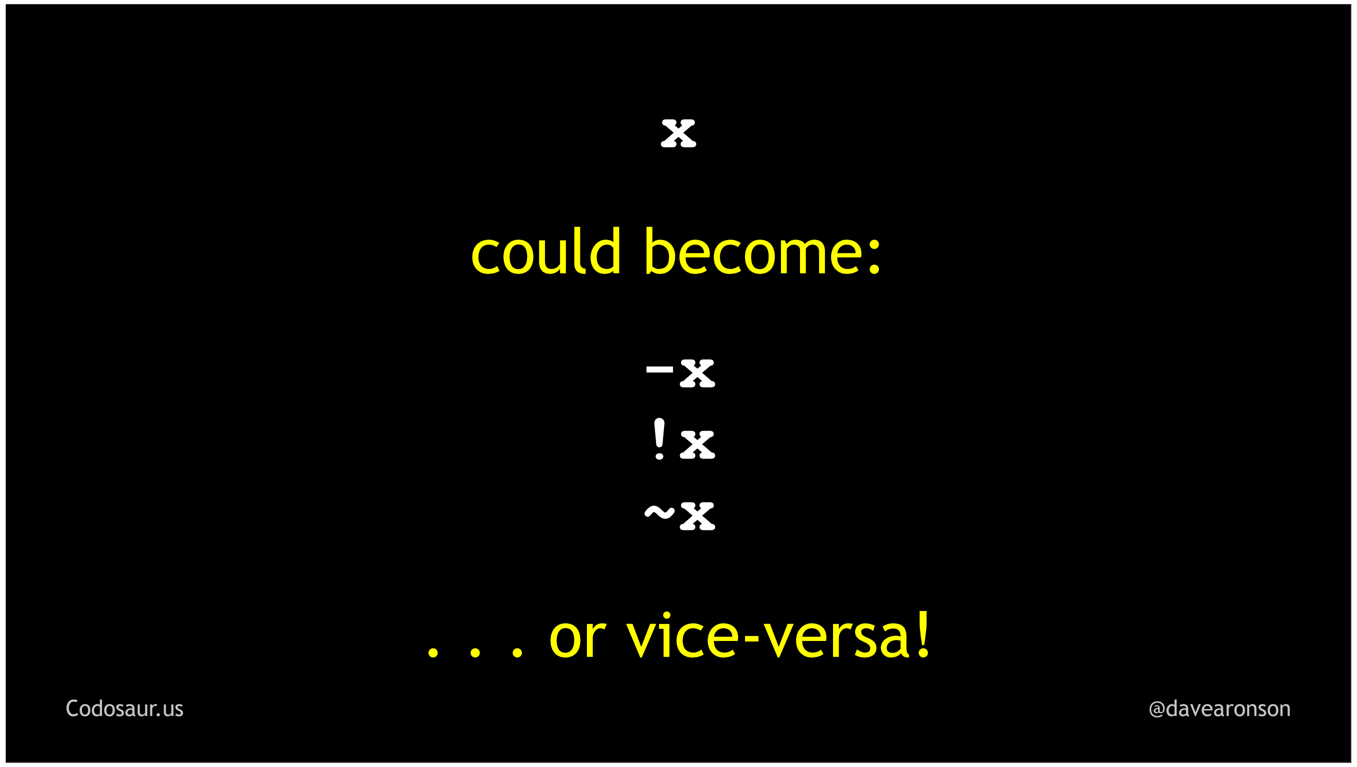
could become:

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

Codosaur.us

@davearonson

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



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

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

could become:

```
foo(z)
```


It can remove an if-condition, so that something that might be done or not, is always done.

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

could become:

```
foo(z)
```

Codosaur.us @davearonson

It can remove a looping condition, so something that might might be done, skipped, or done multiple times, is always done once.

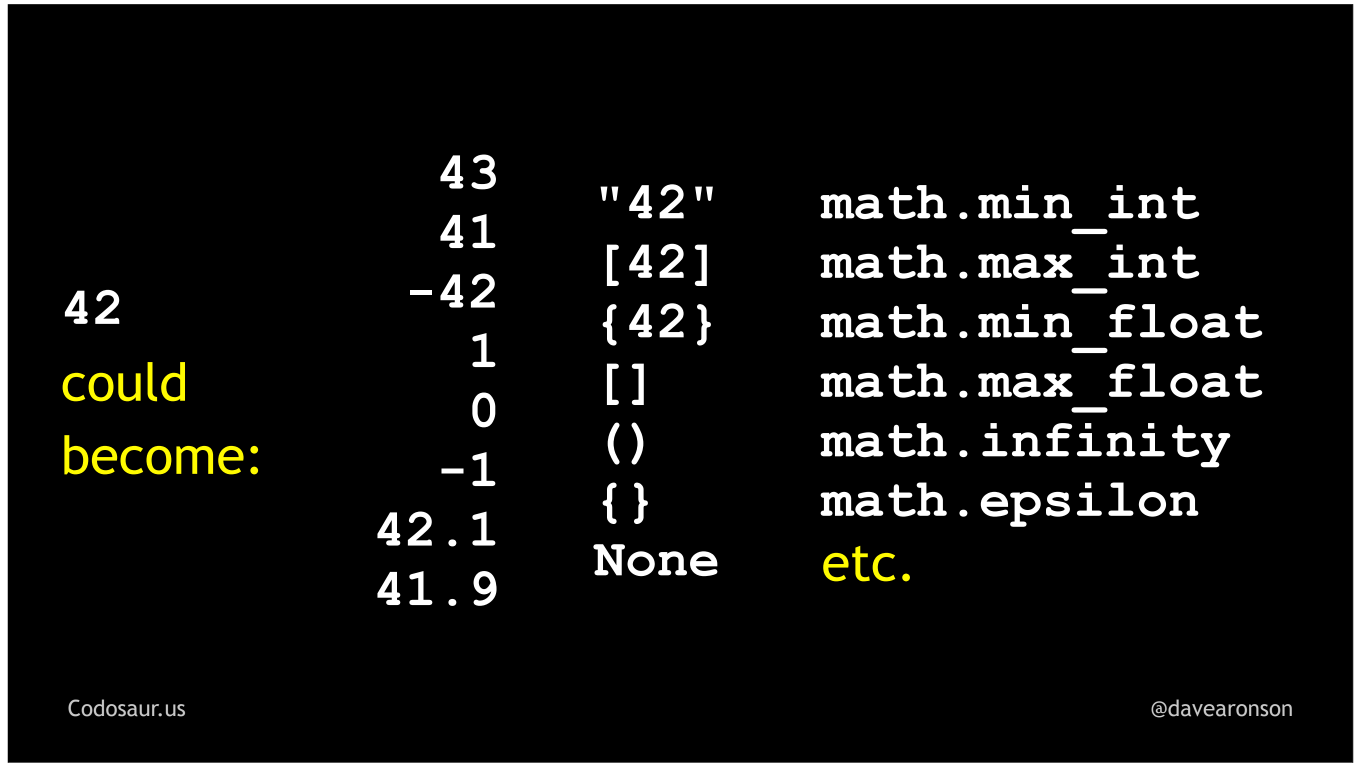
```
def f(x, y):
  # lots of code here
```

could become:

```
def f(x, y): return 0
def f(x, y): return :math.max_int
def f(x, y): return "a string"
def f(x, y): return nil
def f(x, y): return x
def f(x, y): return fail("boom")
def f(x, y): # nothing
```

etc.

It could replace a function's *entire contents* with returning a constant, or any of the arguments, or raising an error, or nothing at all, if the language permits.

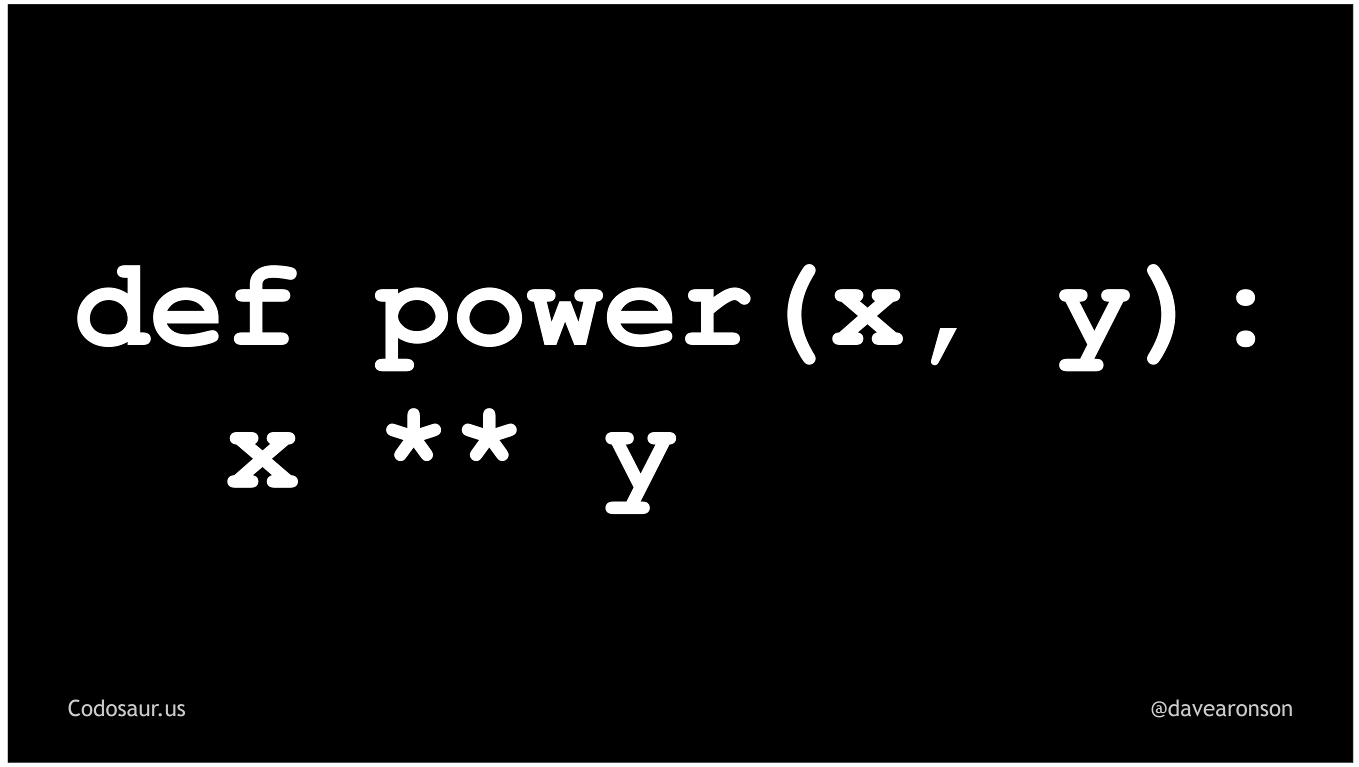


It could change a value to some other value, such as changing 42 to any of these, and many more but I had to stop somewhere. It could even change it to something of a different and possibly incompatible type, such as changing a number into a, if I may quote . . .

. . . 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 . . .



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

Think about what a mutant made from this might *return*.

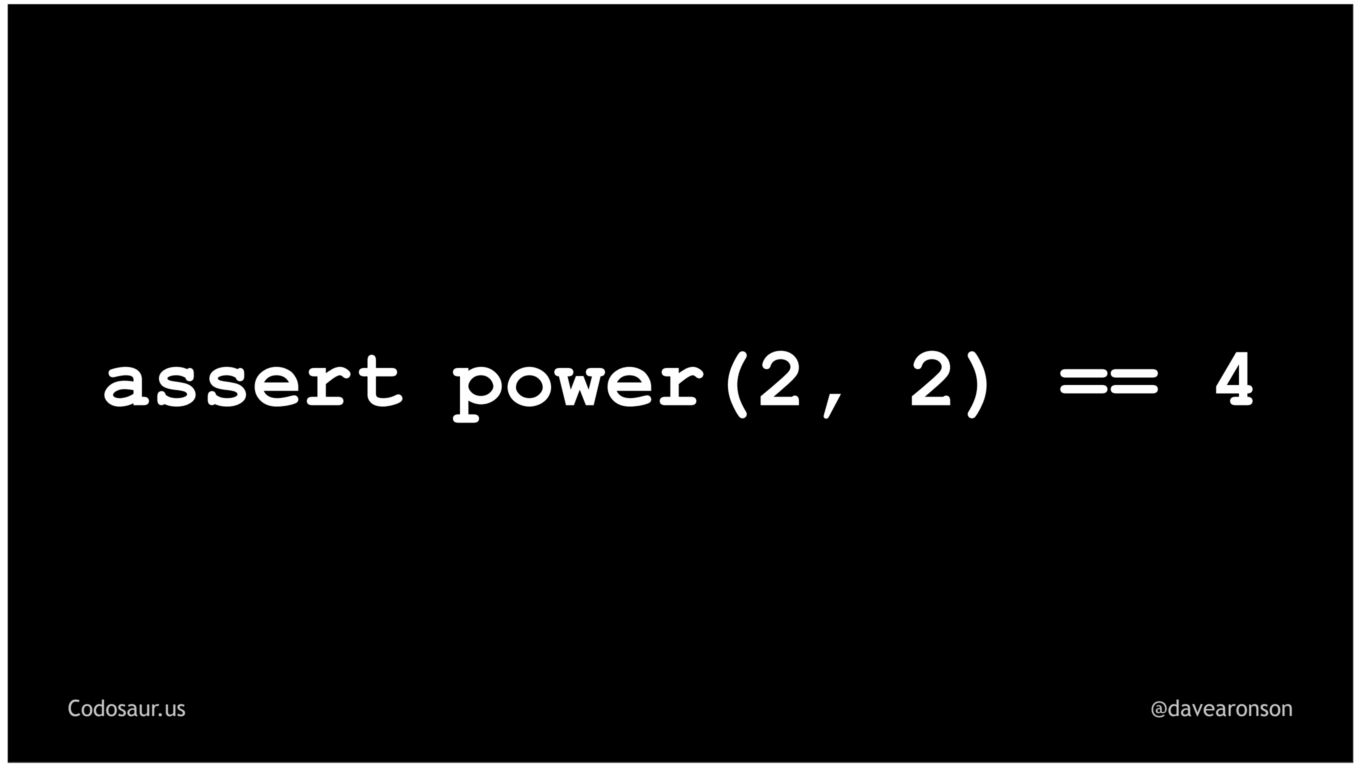
Mainly, it could return results such as . . .

```
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
```

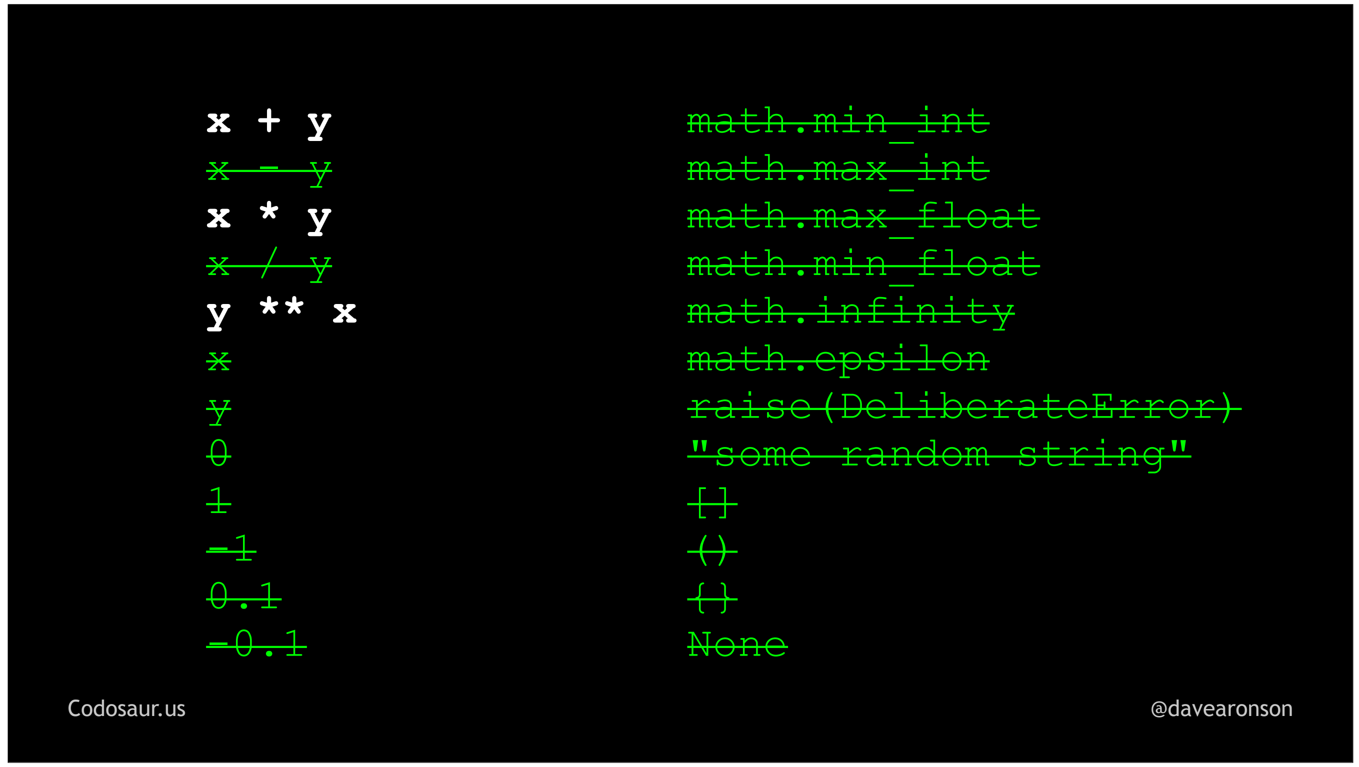
Codosaur.us @davearonson

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

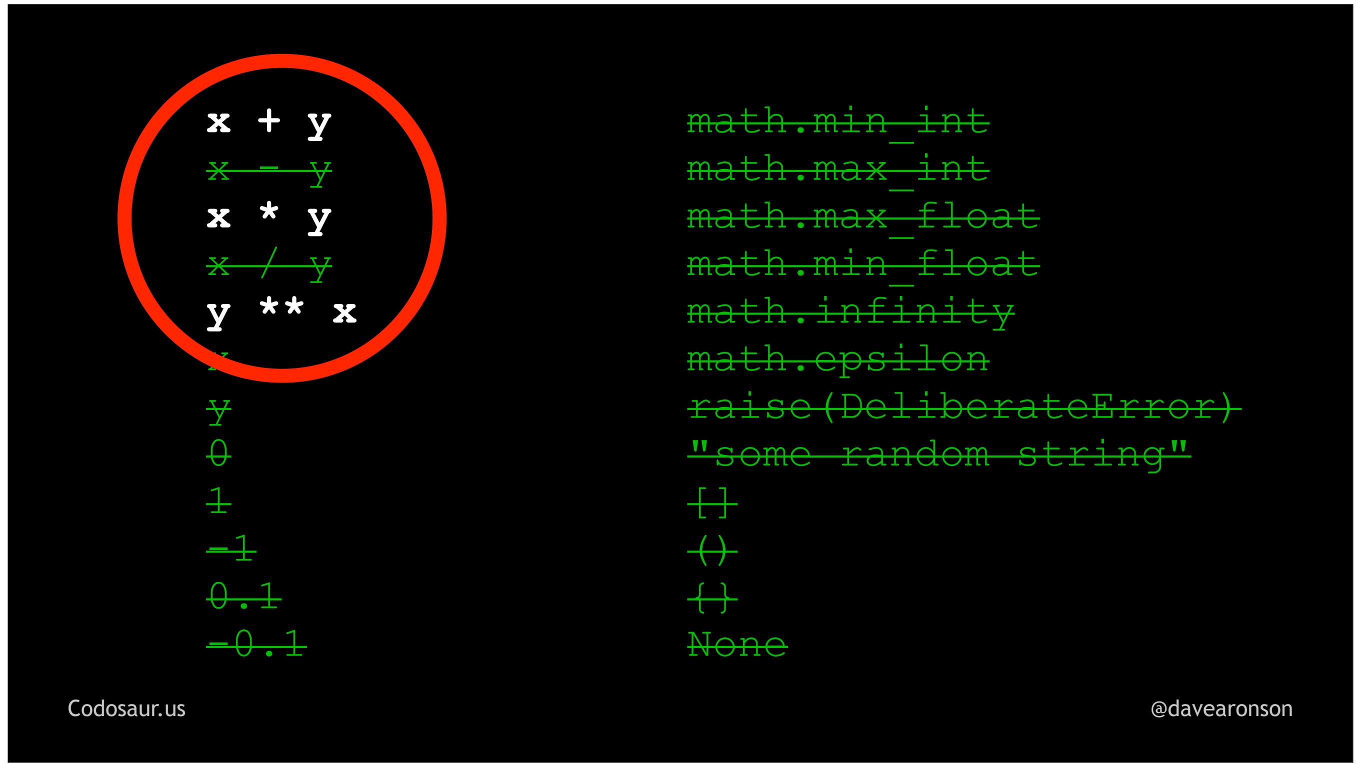
Now suppose we had only one test . . .



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

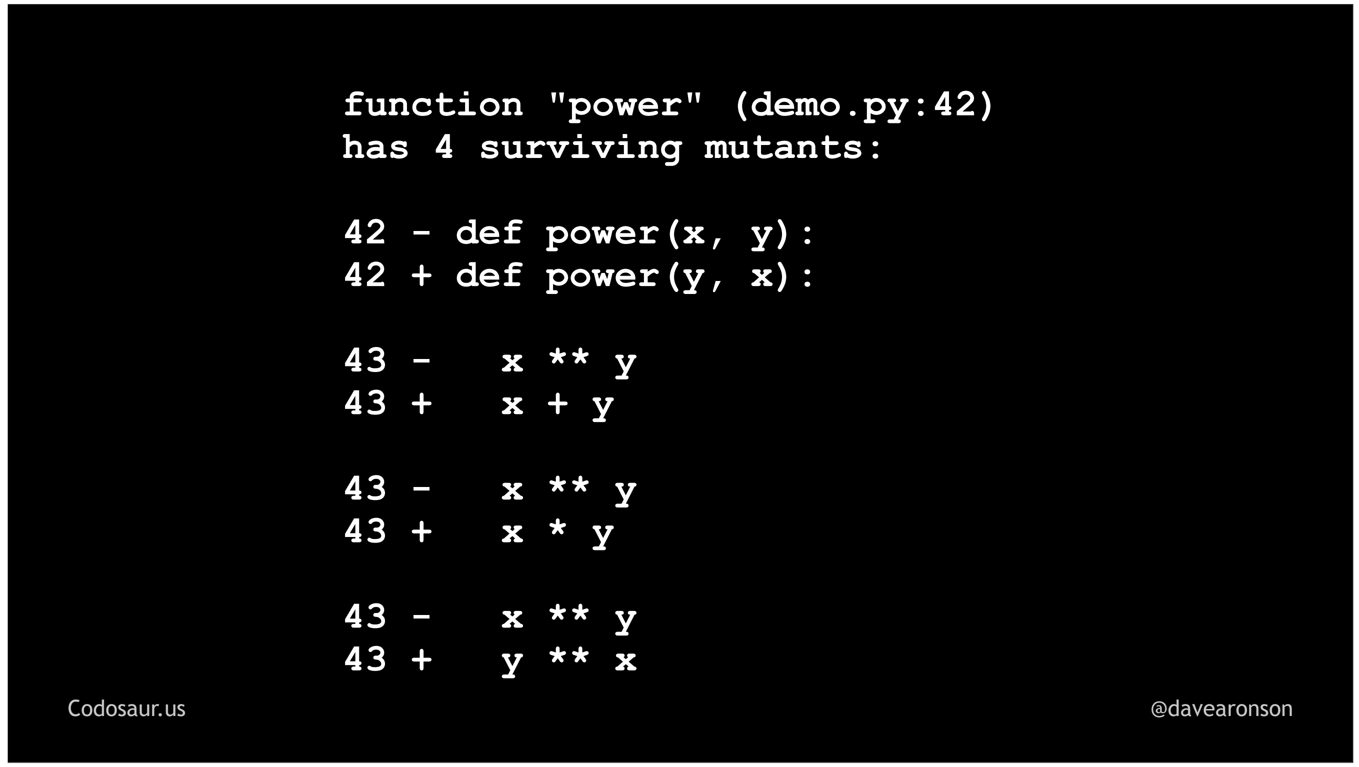


. . . here in crossed-out green. The ones returning constants, are very unlikely to match. There's no particular reason a tool would put a 4 there, as opposed to zero, 1, -1, and other such significant numbers. Changing the exponentiation into subtracting one argument from the other gets us zero, dividing 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 *"survive"* our test.

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



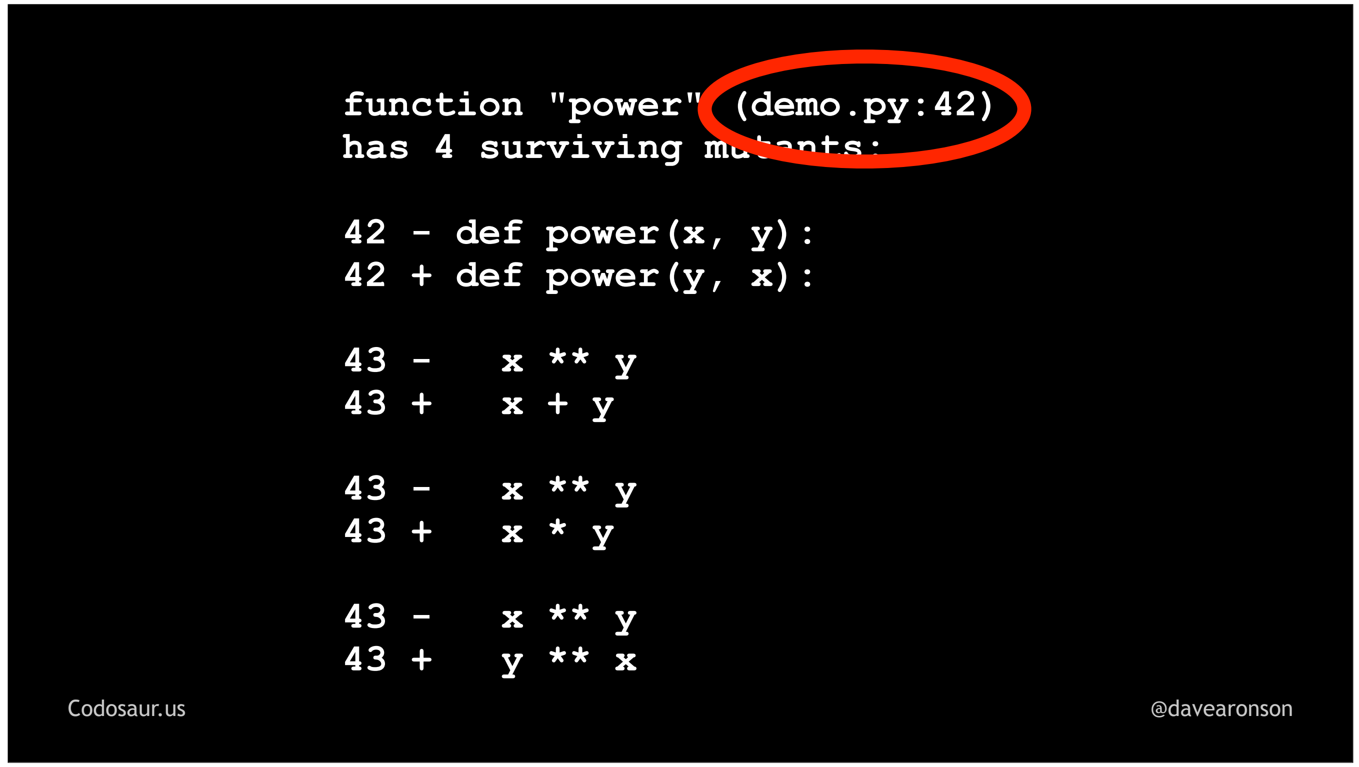
. . . this. The exact words, format, amount of context, and so on, will vary greatly depending on exactly which tool we use, but *semantically*, the information should be pretty much the same. And that is, that if we changed . . .

```
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
```

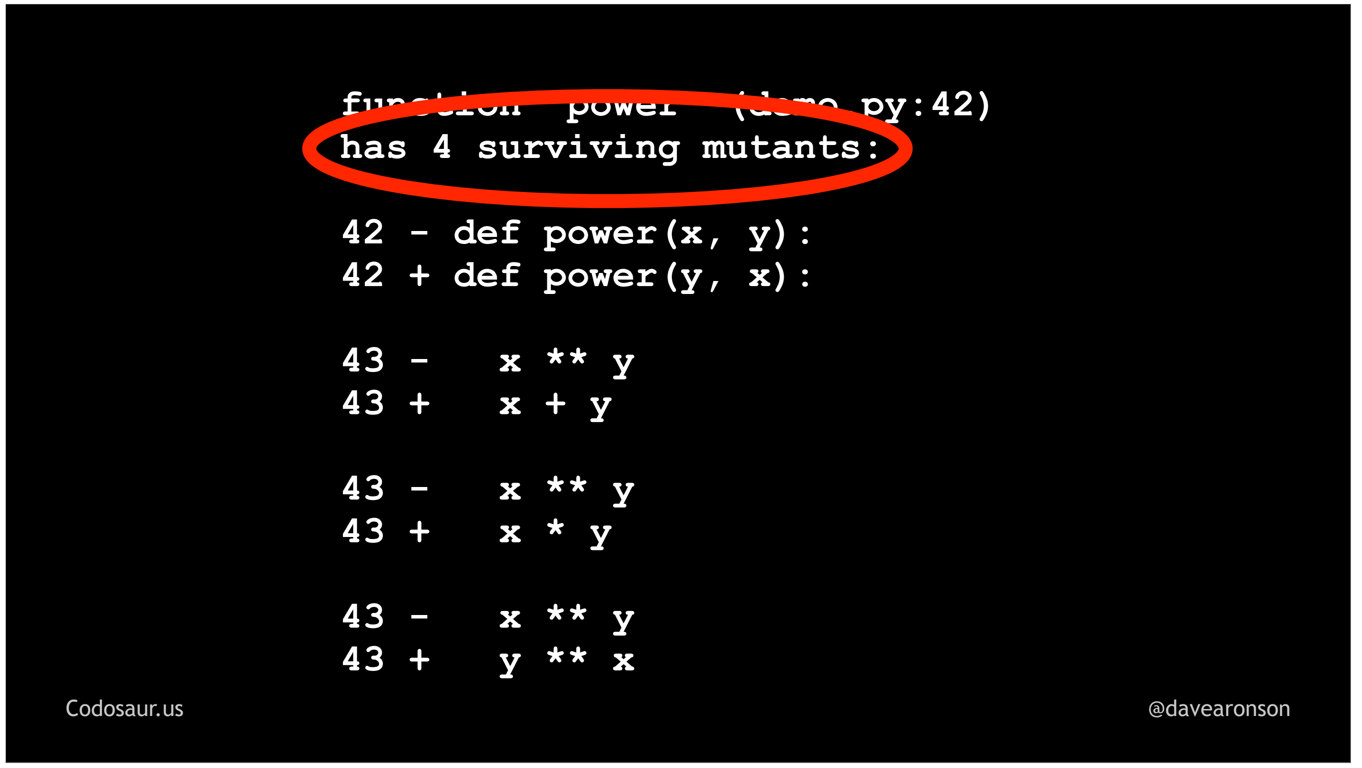
Codosaur.us

@davearonson

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

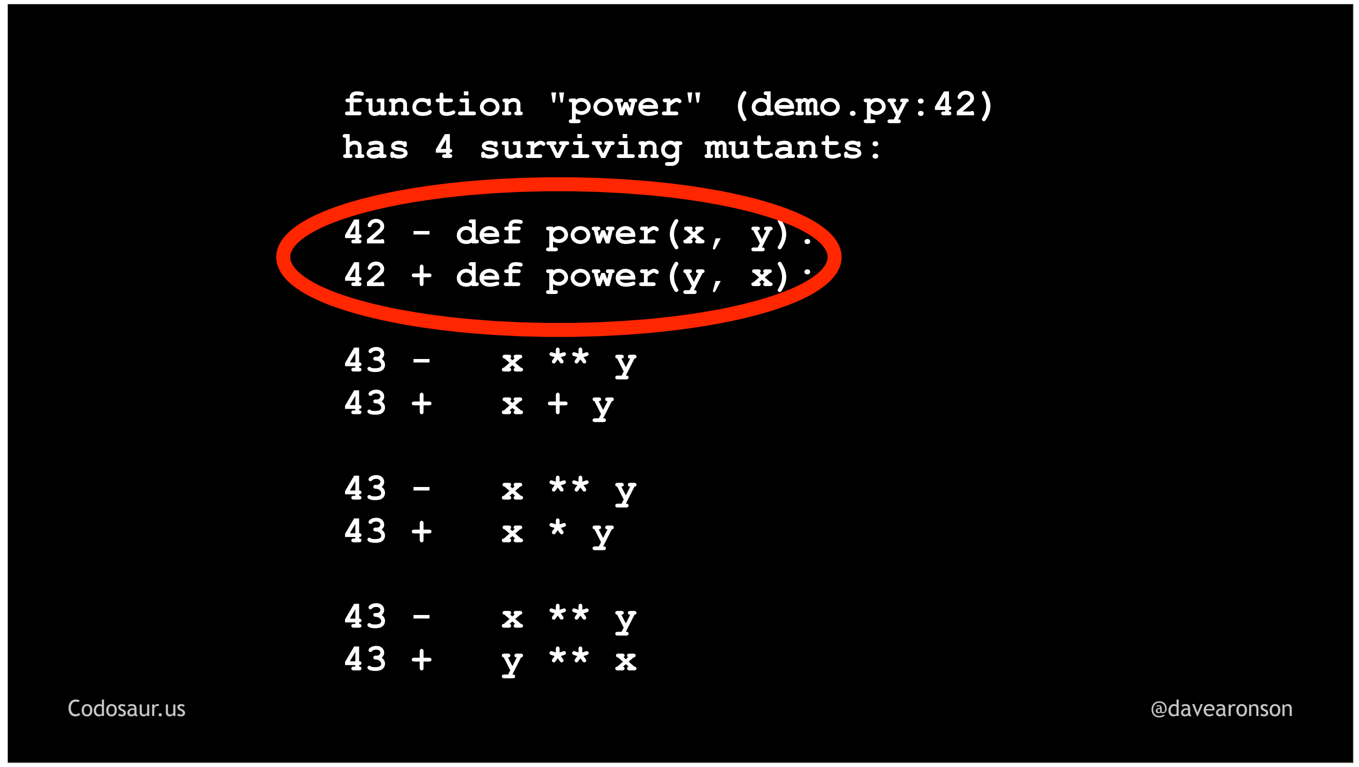


. . . file demo.py, at line 42 . . .



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

And, that those four ways are: . . .



. . . to swap the arguments, . . .

```
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 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

. . . 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

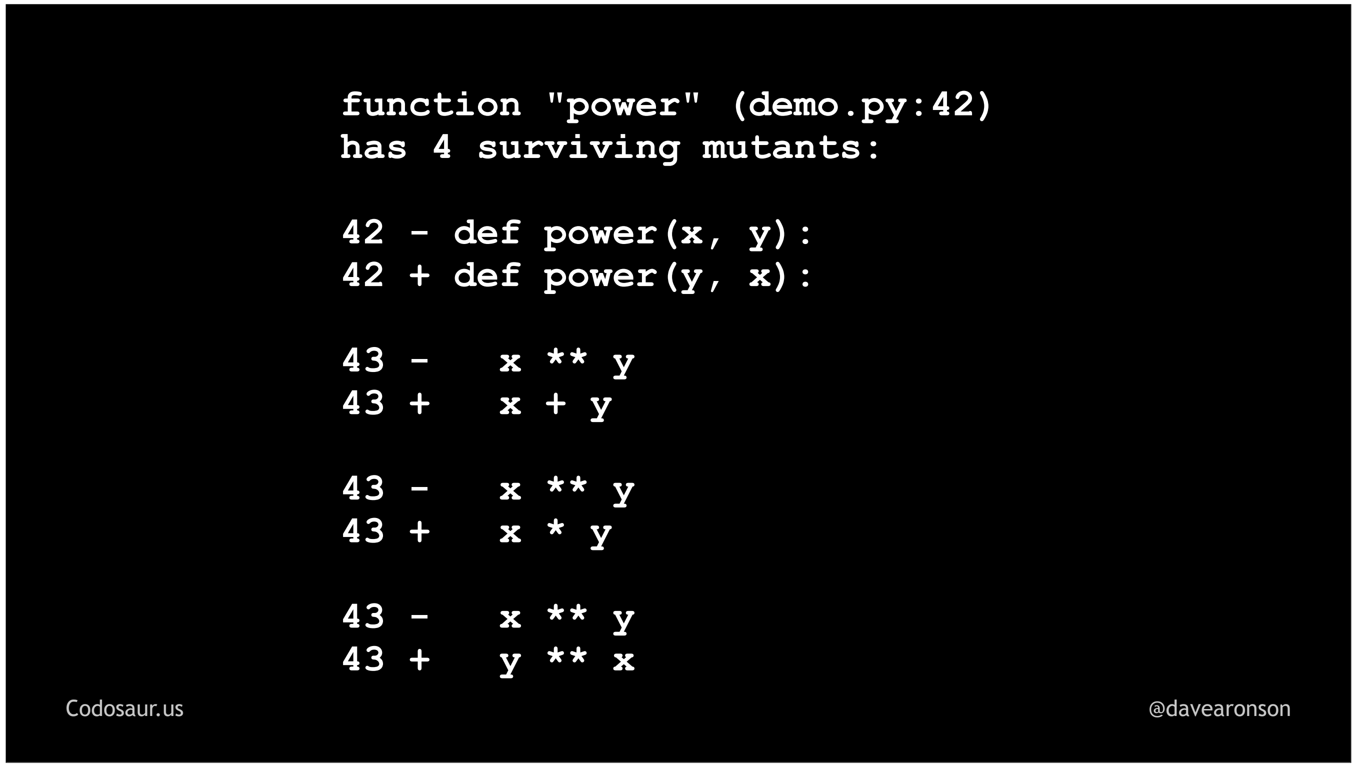
. . . this set of surviving mutants trying to tell us? We can tell from a glance at . . .

```
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 . . .

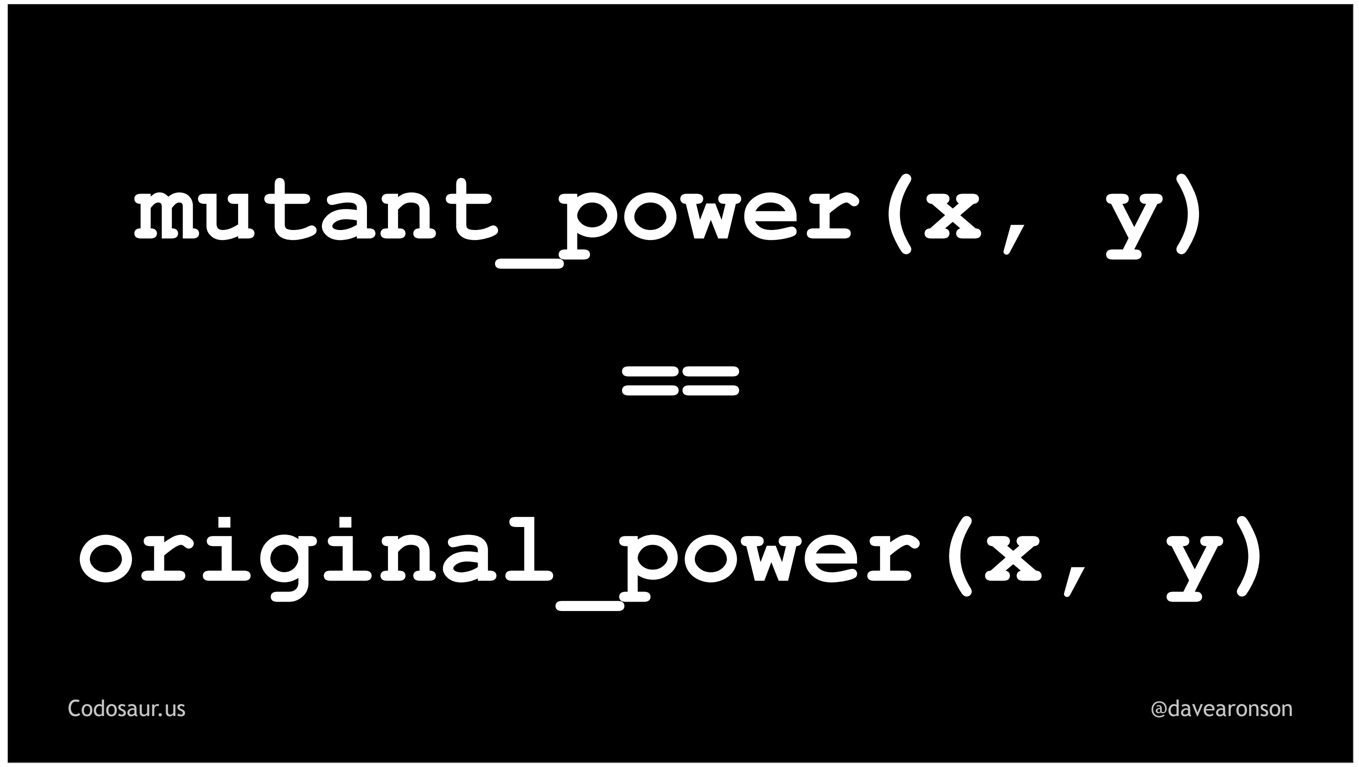


. . . these mutants surviving? Are they . . .

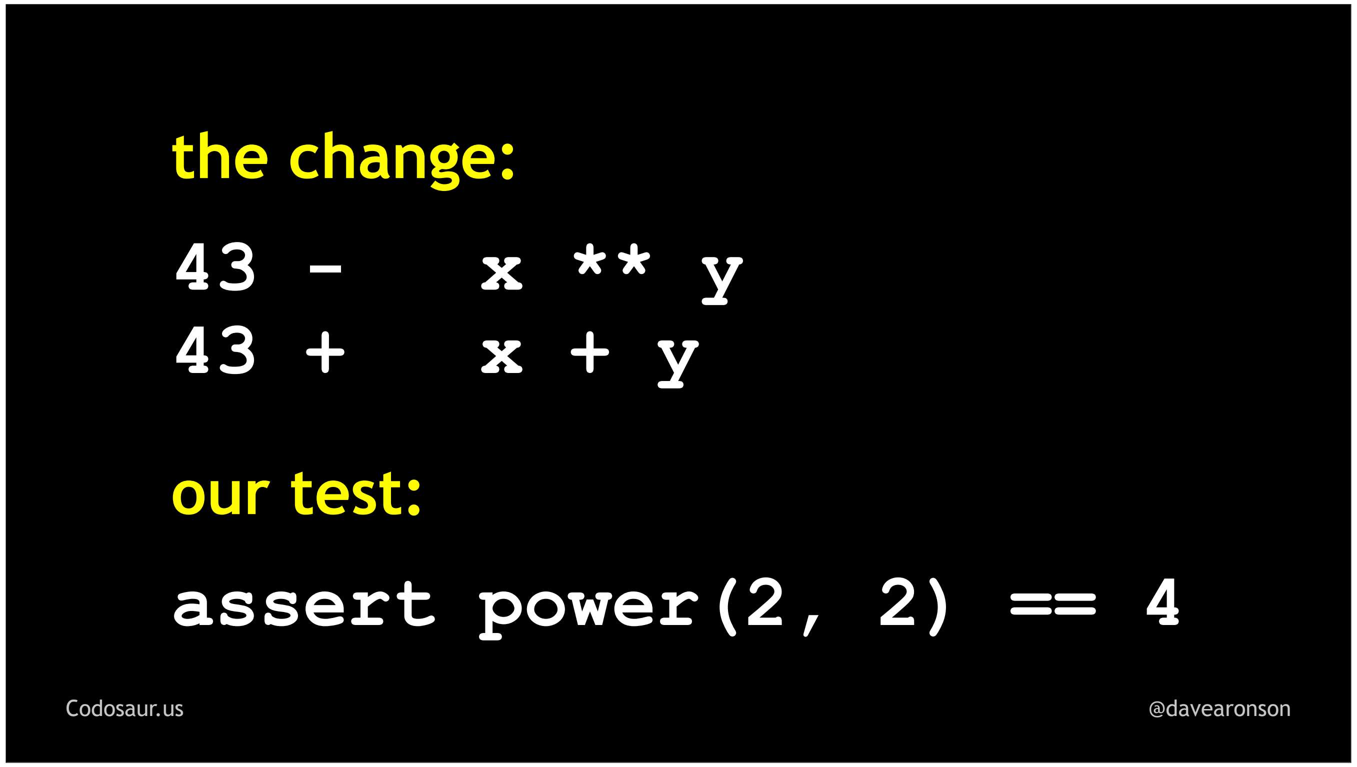
. . . pulling heists? Are they getting free room and board at the . . .

. . . Xavier Institute? Or what?

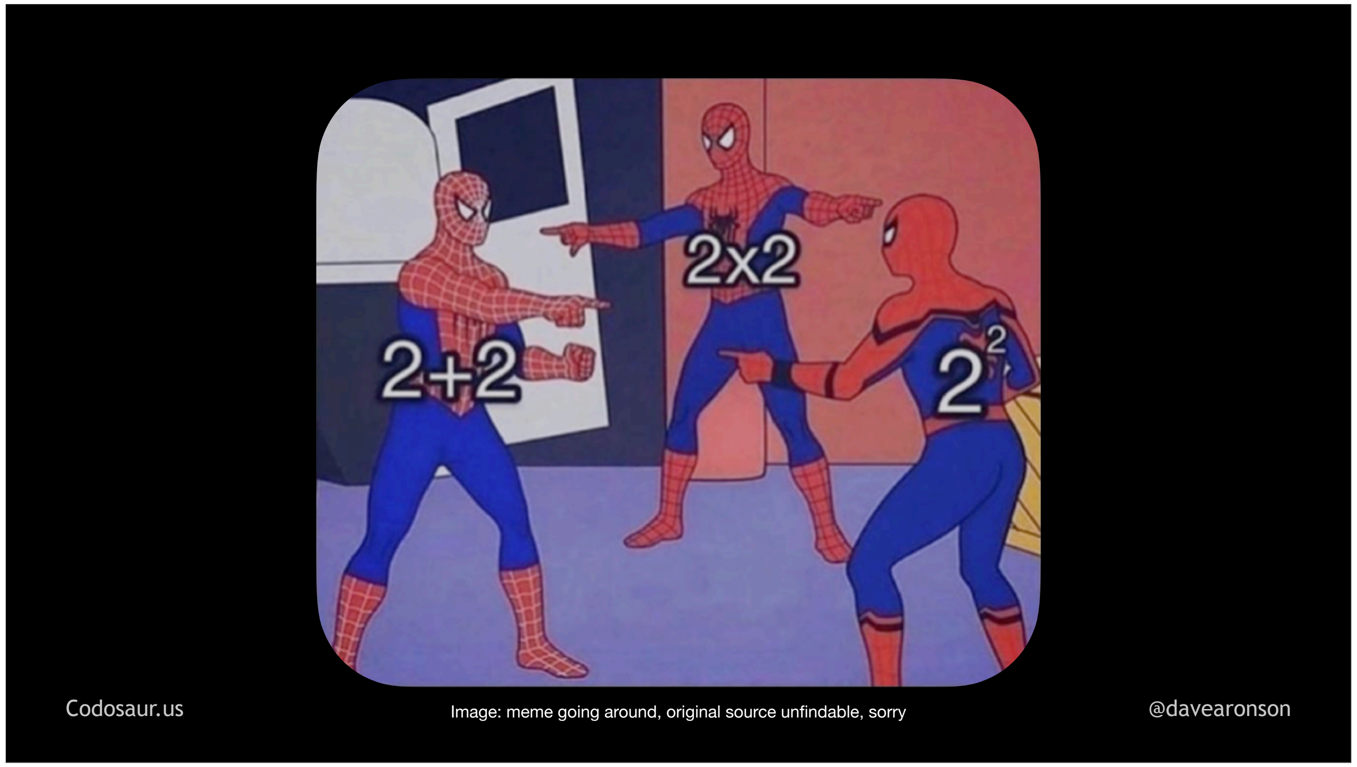
The usual answer is that . . .



. . . 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 . . .

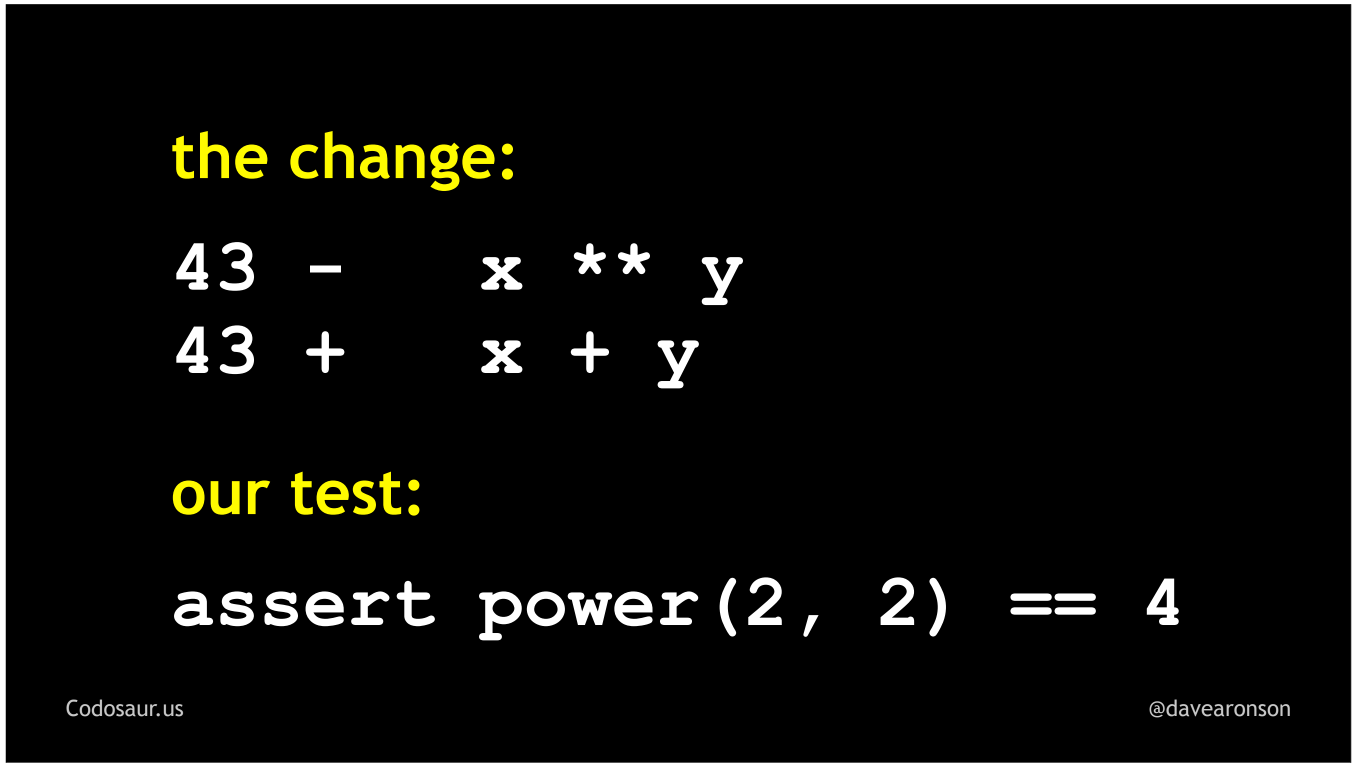


. . . the "plus" mutant. Looking at the change, together with our test, makes it much clearer that this one survives because . . .

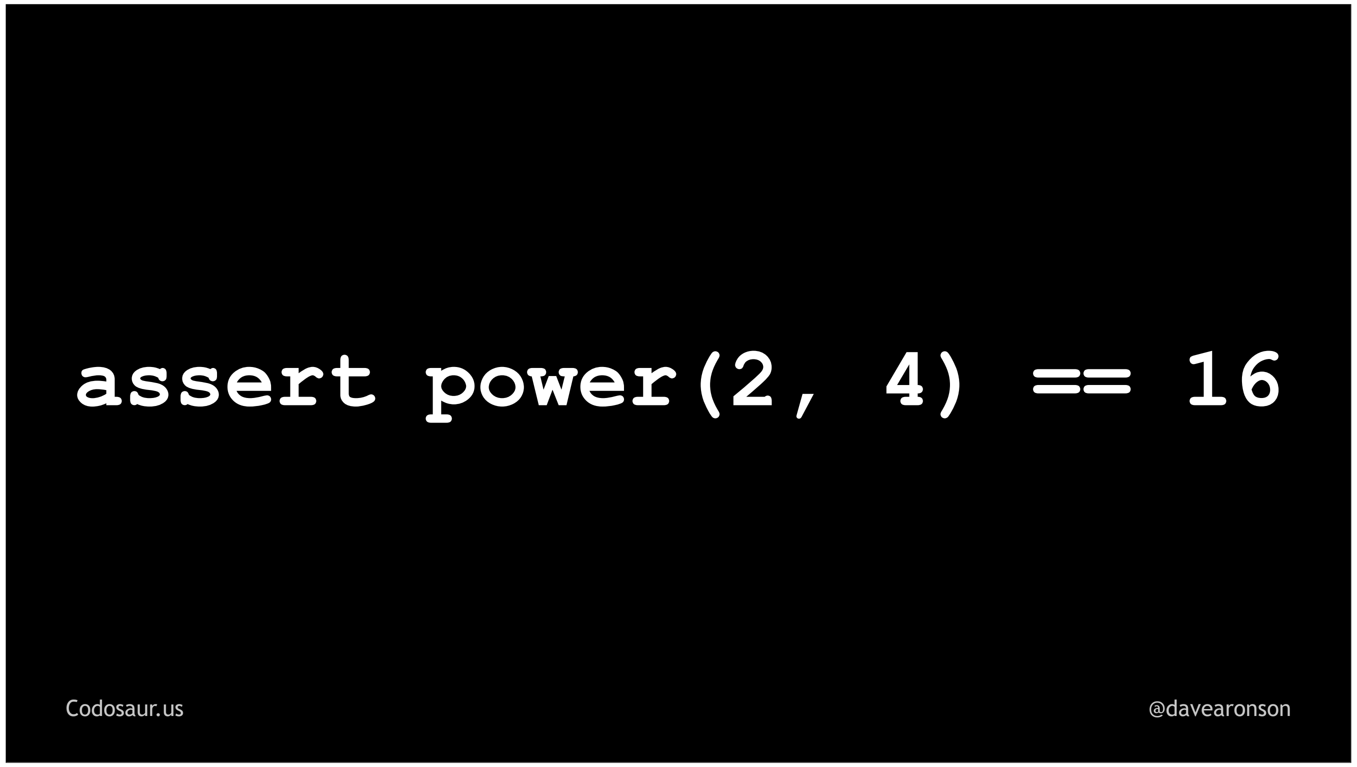


. . . two *plus* two equals two *to* the two (singing) tu tu, tu tu tu-tu tu . . . . (And so does two *times* two, but he's in the background, we can save him for later.)

So how can we *kill* . . .



. . . 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 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 . . .



. . . something like asserting 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! We devs should certainly know our powers of two at least *that* well! So, this kills the "times" mutant as well. Killing one mutant often kills other mutants of the same function, often a large fraction of them.

But . . .

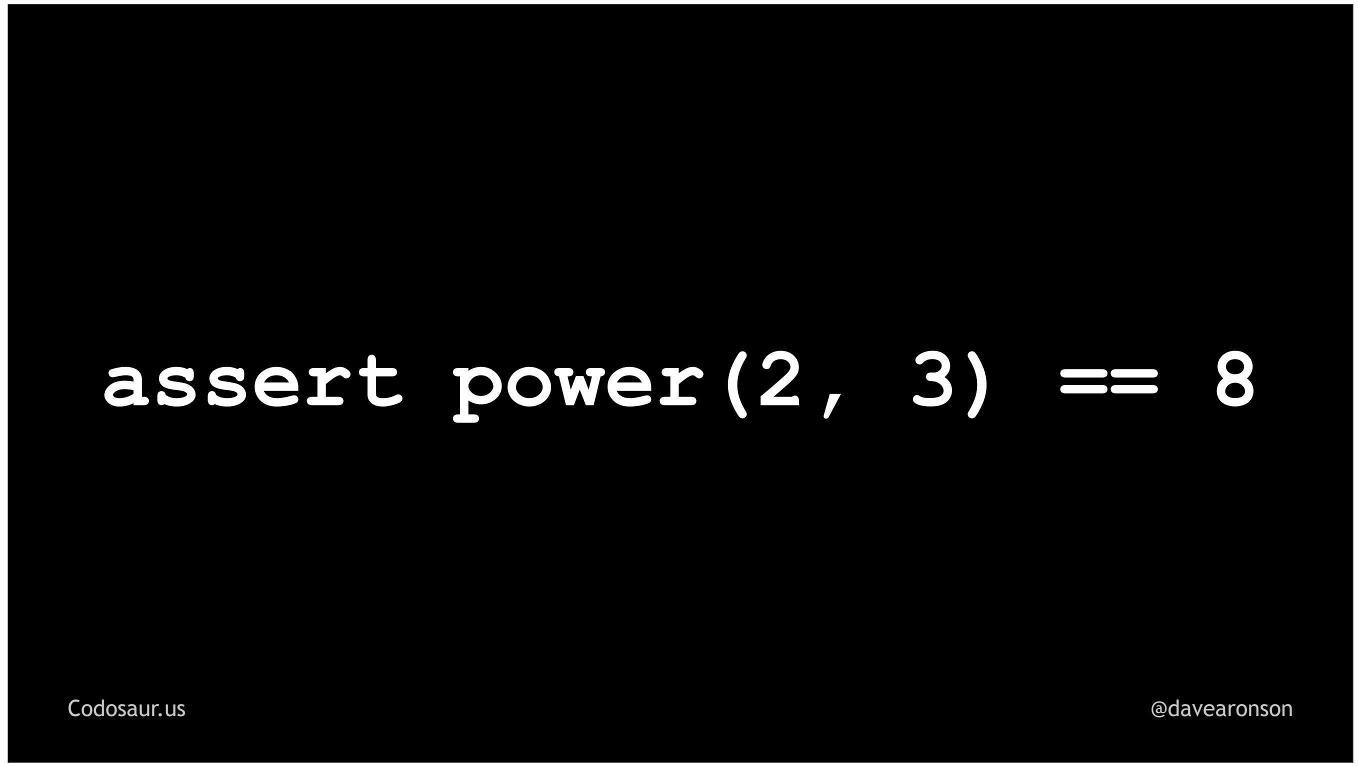
. . . the pair of argument-swapping mutants survive, because . . .

```
4 ** 2 == 16

2 ** 4 == 16
```

Codosaur.us @davearonson

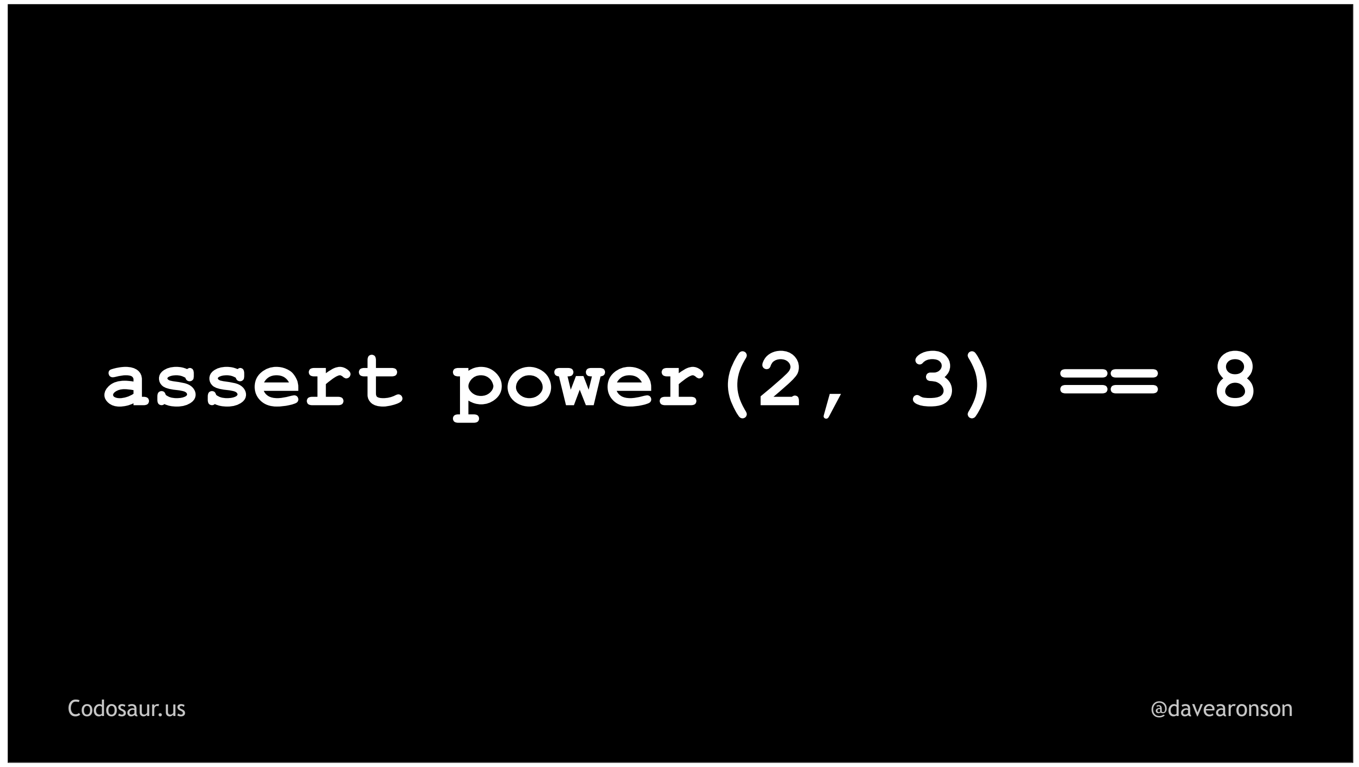
. . . four squared and two to the fourth, are both sixteen. But that’s not a big deal, we can . . .

. . . attack these mutants separately, no need to kill them all 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 . . .



. . . 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, guess what, not eight, so the "plus" and "times" mutants *stay* dead, and we don't get any . . .

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



. . . 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 . . .

. . . *that* flerken!

This may make mutation testing sound . . .

. . . simple, but this was a downright trivial example, so we could *easily* think up arguments to make *all* mutants, within reason, behave differently from the original code.

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, kind of like a woodchuck, looping to pick up where it left off, until the message is all sent. This is a very common pattern.

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
```


. . . 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
```

Codosaur.us @davearonson

. . . this.

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

```
assert send_message(msg, size) == size
```

Codosaur.us @davearonson

. . . *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!)

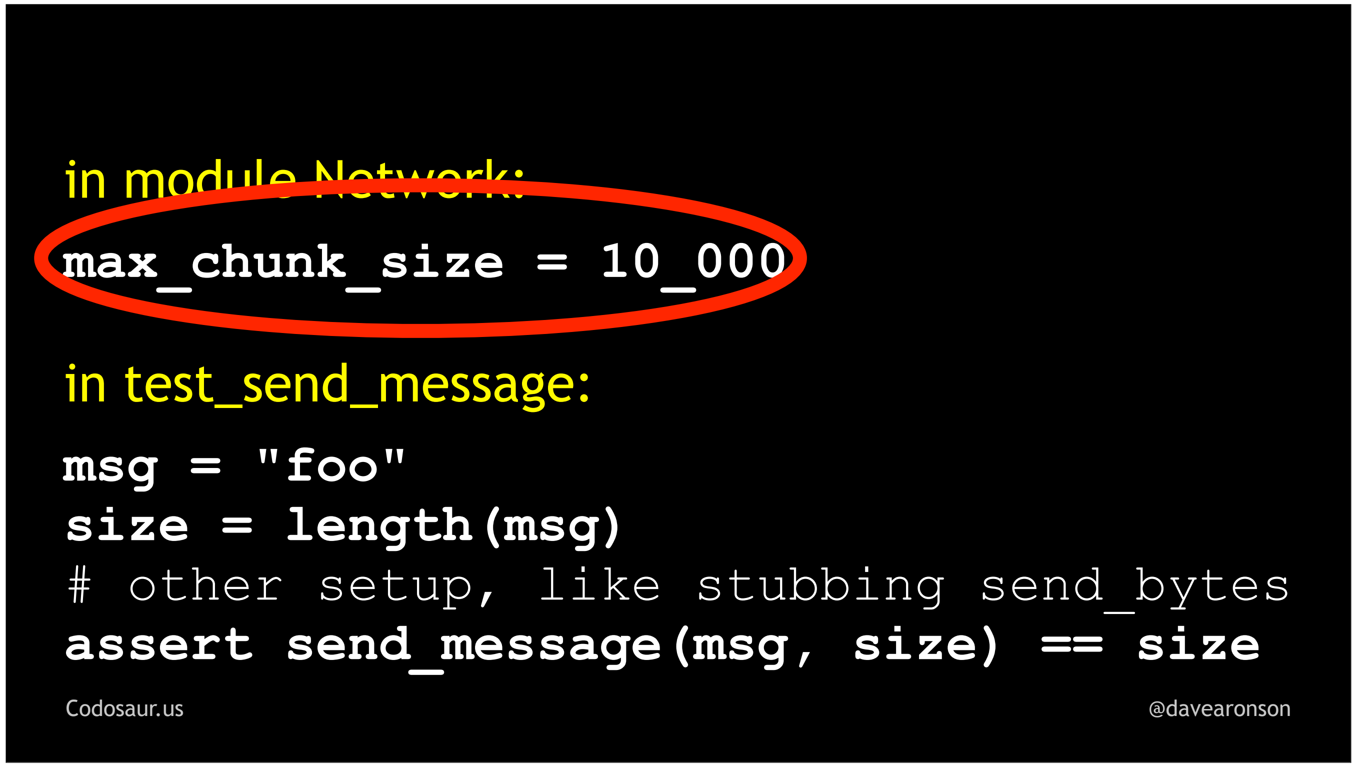
If a mutant that only goes through . . .

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

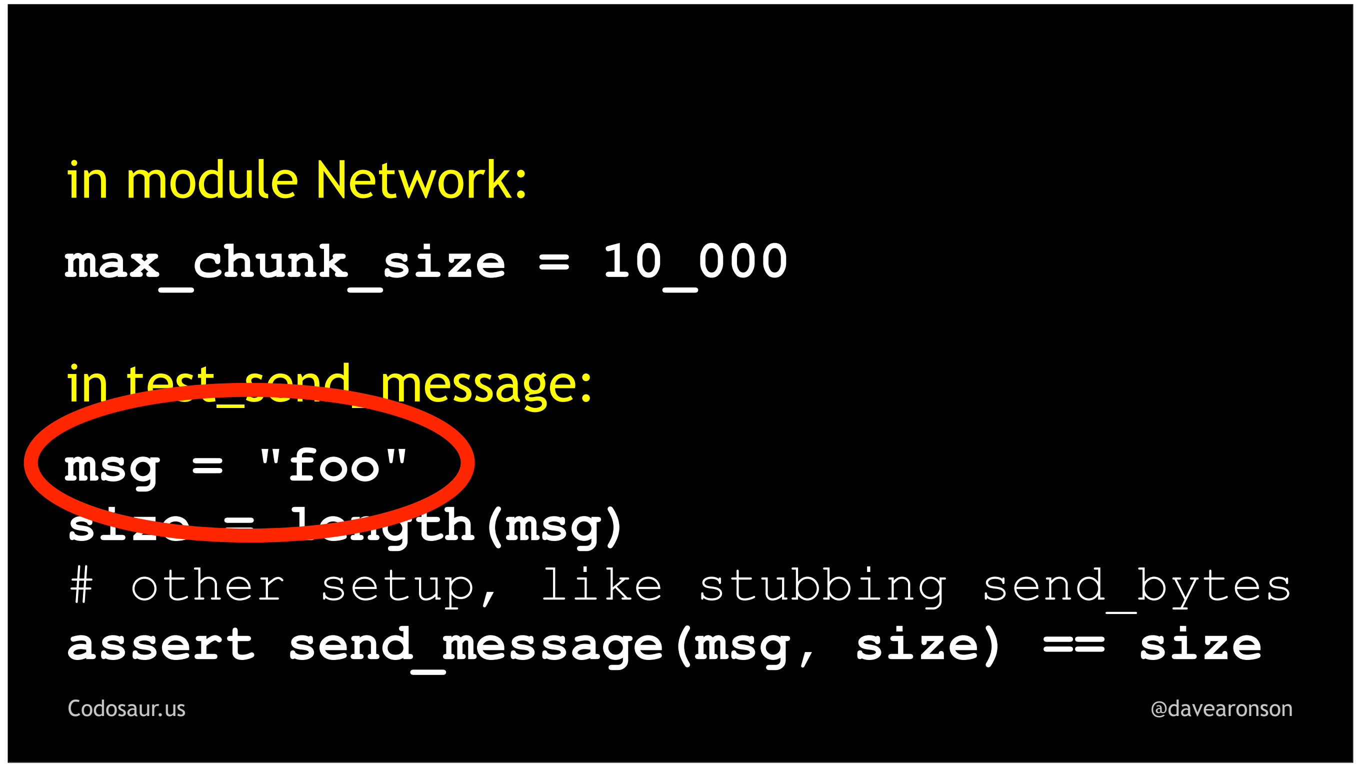
Codosaur.us @davearonson

. . . 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 often 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, . . .

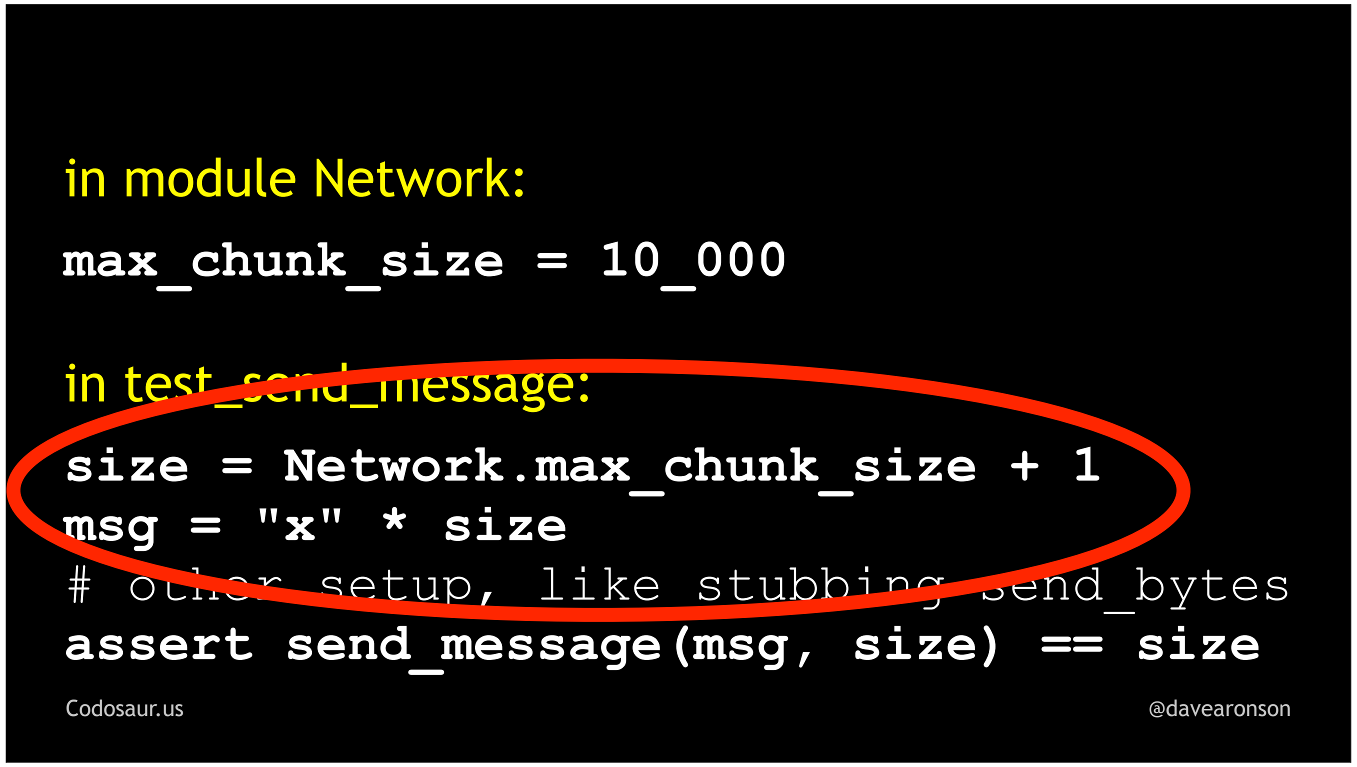


. . . suppose our maximum chunk size, what send_*bytes* can handle in one chunk, is a phenomenal 10,000 bytes. But . . .



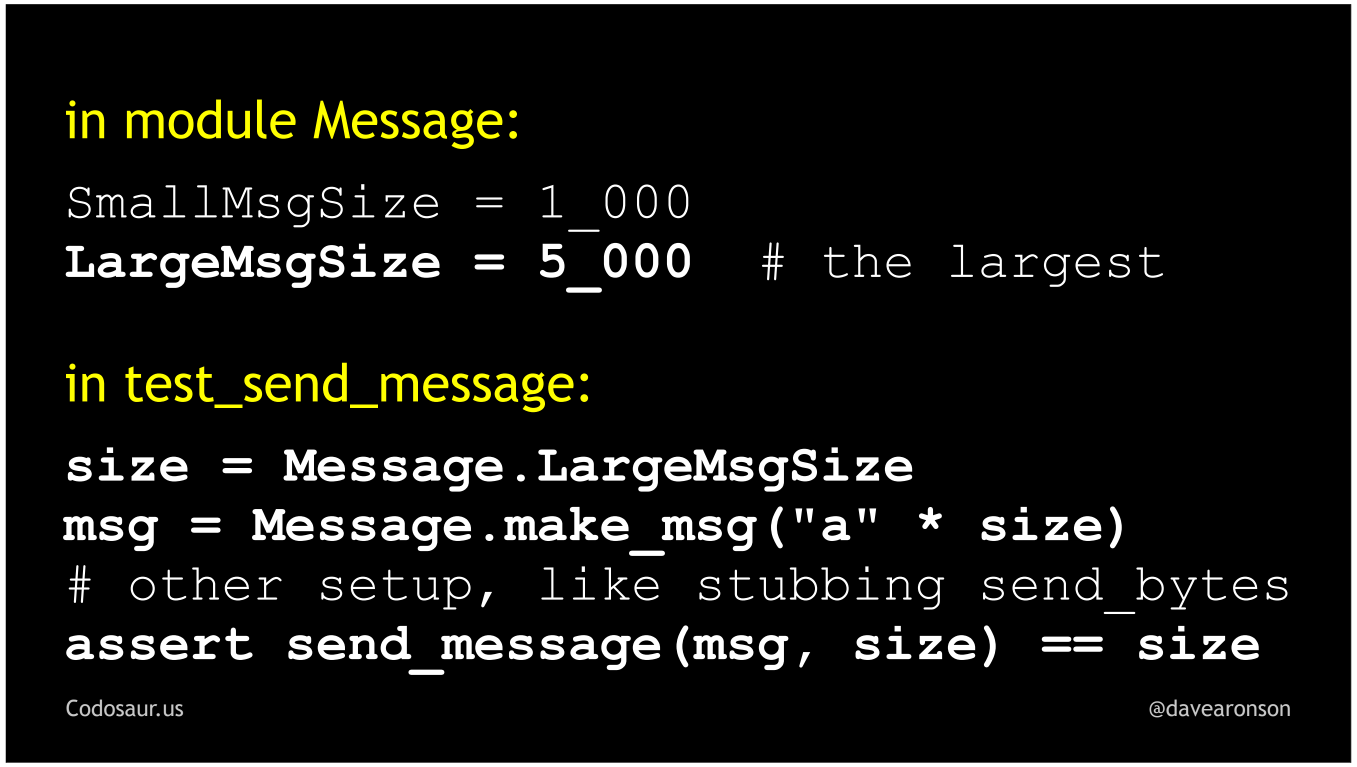
. . . we’re only testing with an itty-bitty *three* byte message. (PAUSE!)

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 . . .



. . . 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, . . .



. . . 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. Some other stuff is now clearly 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, len)
```

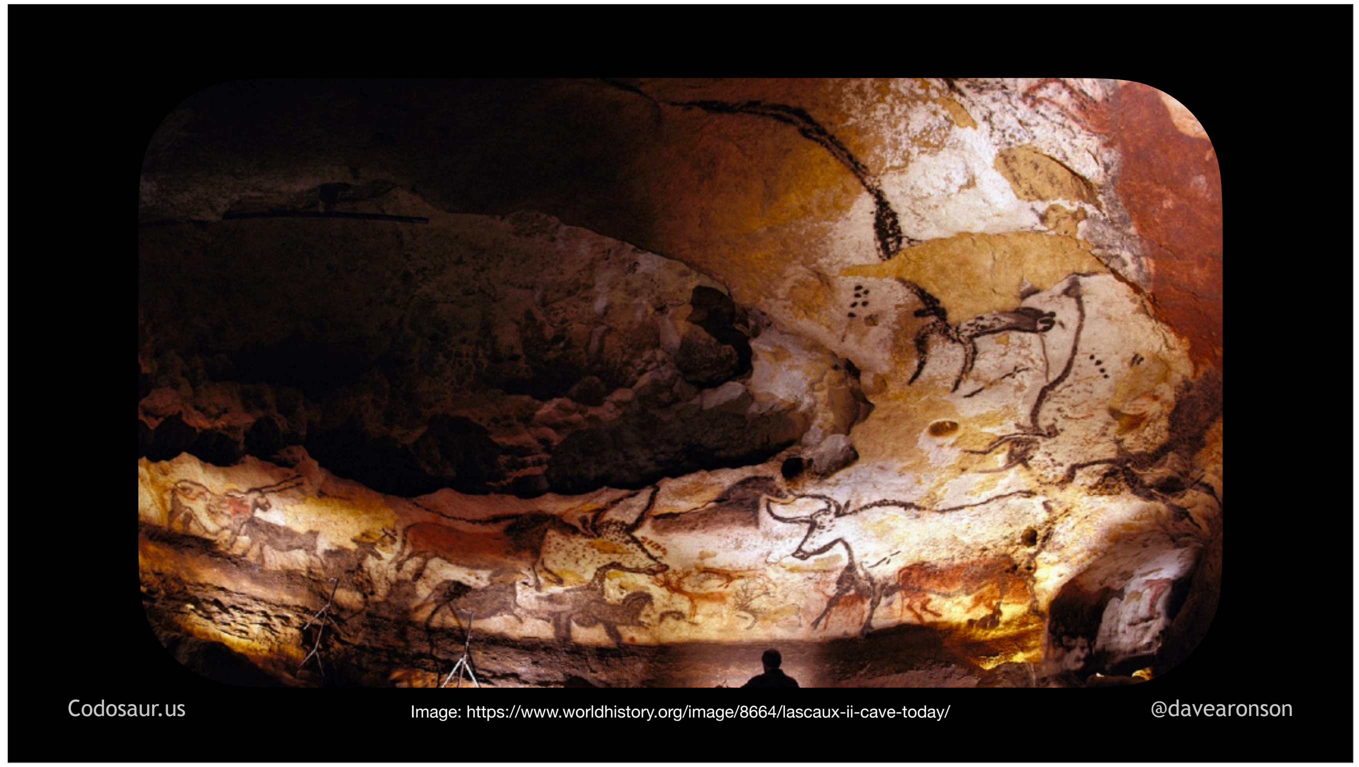
Codosaur.us @davearonson

. . . this. (PAUSE!) Now the ultimate message is crystal 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, that we can't shove down the stack into send_*bytes*, but at the very least, the *looping* was redundant. Fortunately, when it's this kind of problem, with meaningless code, 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. And for the security-conscious, this might also reduce potential attack surface.

Now that we've seen examples of finding both 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 . . .

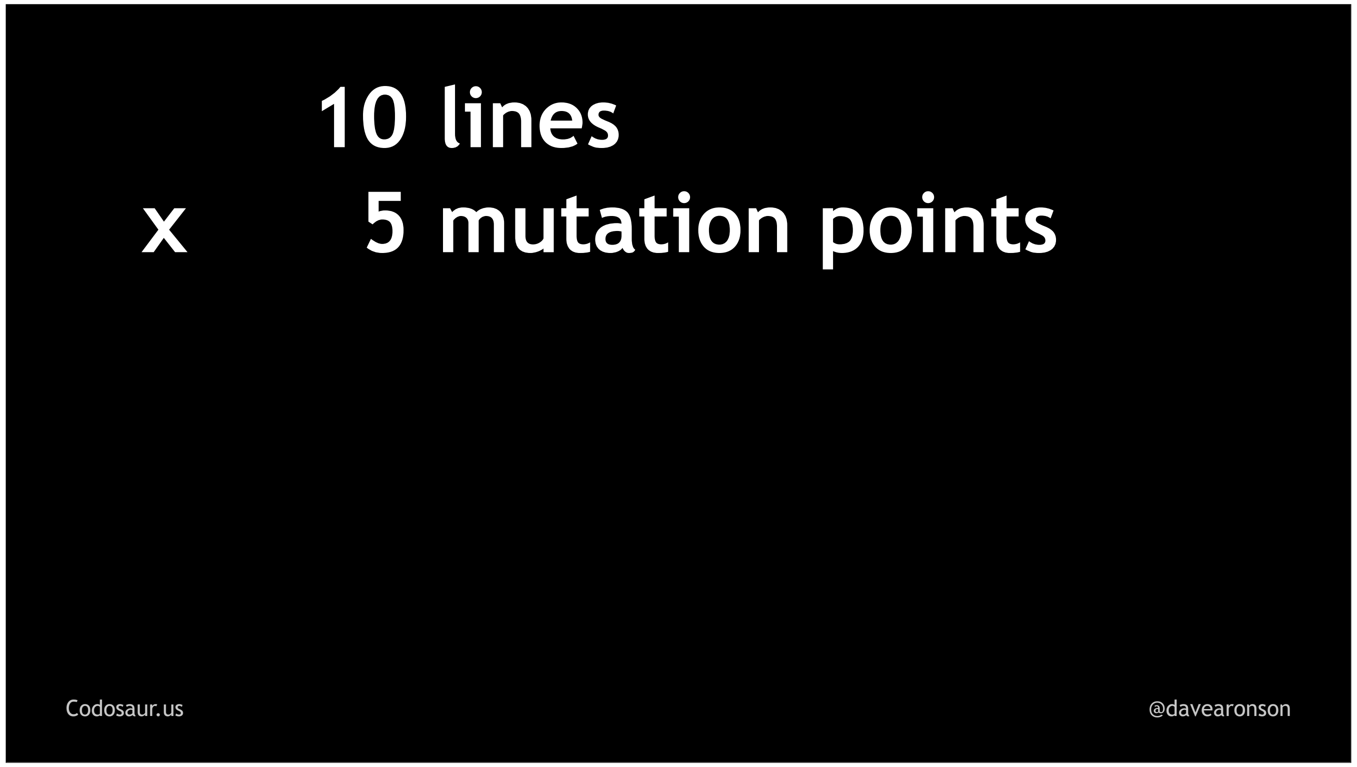
. . . bizarro idea come from anyway? Mutation testing has a surprisingly . . .



. . . 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 actual *tool* didn't appear until 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, after significant advances in CPU *speed*, multi-*core* CPUs, larger and cheaper memory, and so on. But now, it’s practical even on fairly low-end, but at least relatively modern, systems like this 2020 MacBook Air.

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, . . .

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



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

10 lines
x 5 mutation points
x 20 alternatives

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

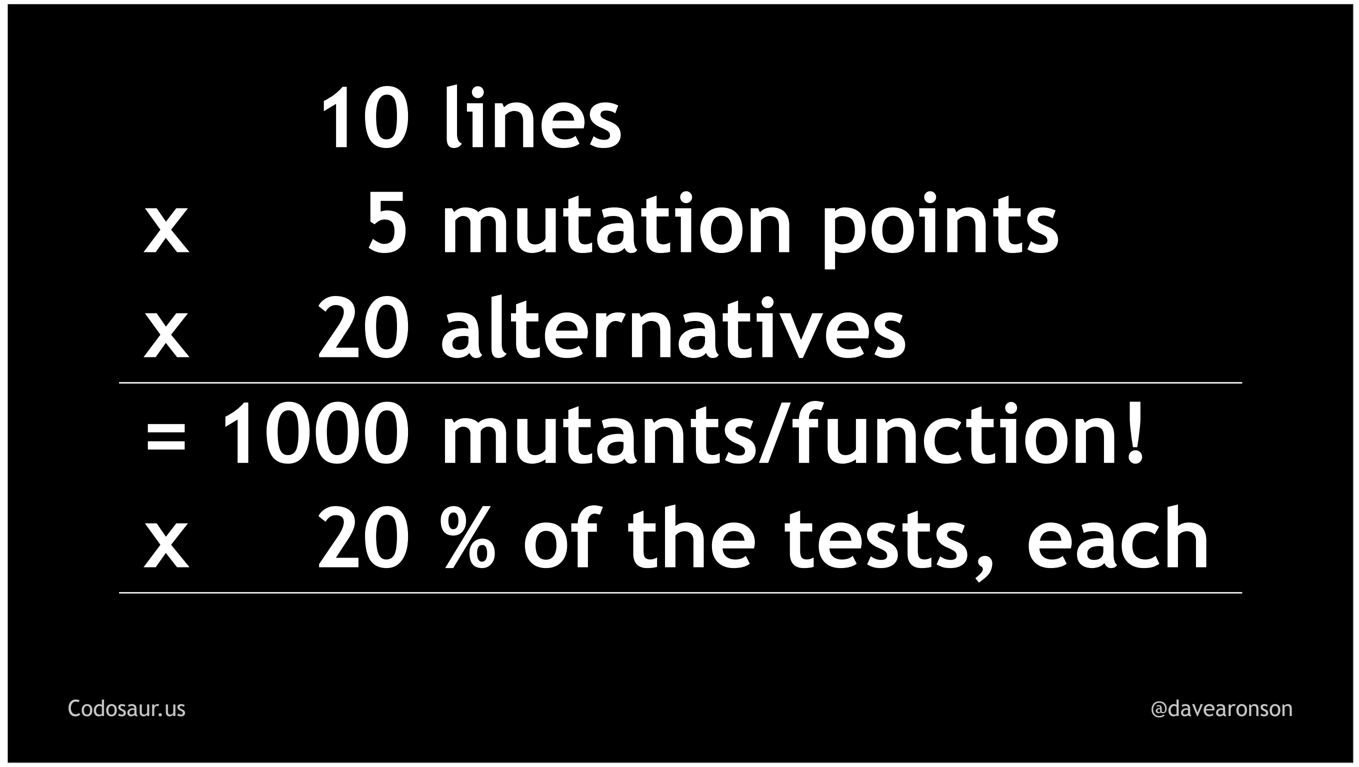
10 lines
x 5 mutation points
x 20 alternatives

= 1000 mutants/function!

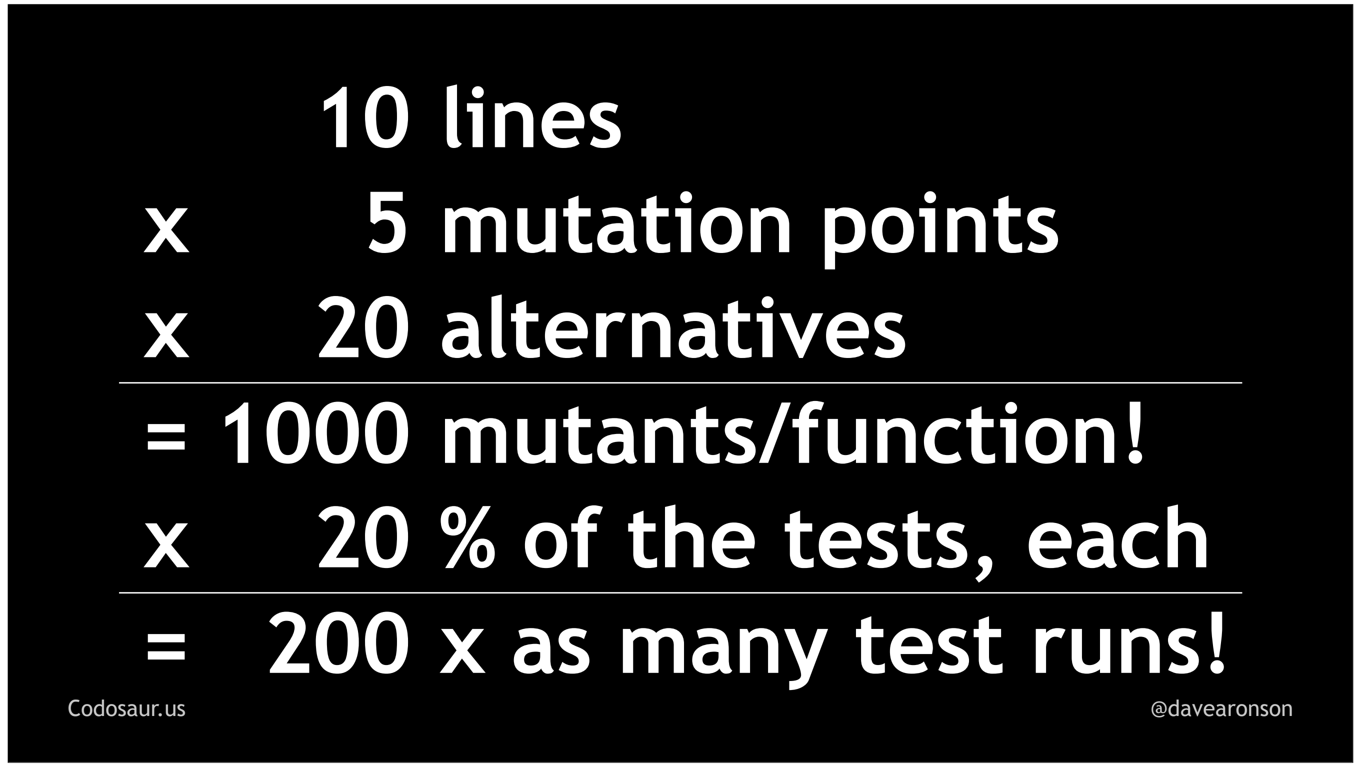
Codosaur.us @davearonson

. . . 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 the mutant 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 . . .



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



. . . *two hundred times* as many test runs for that function, as just doing 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!*

But there is some . . .

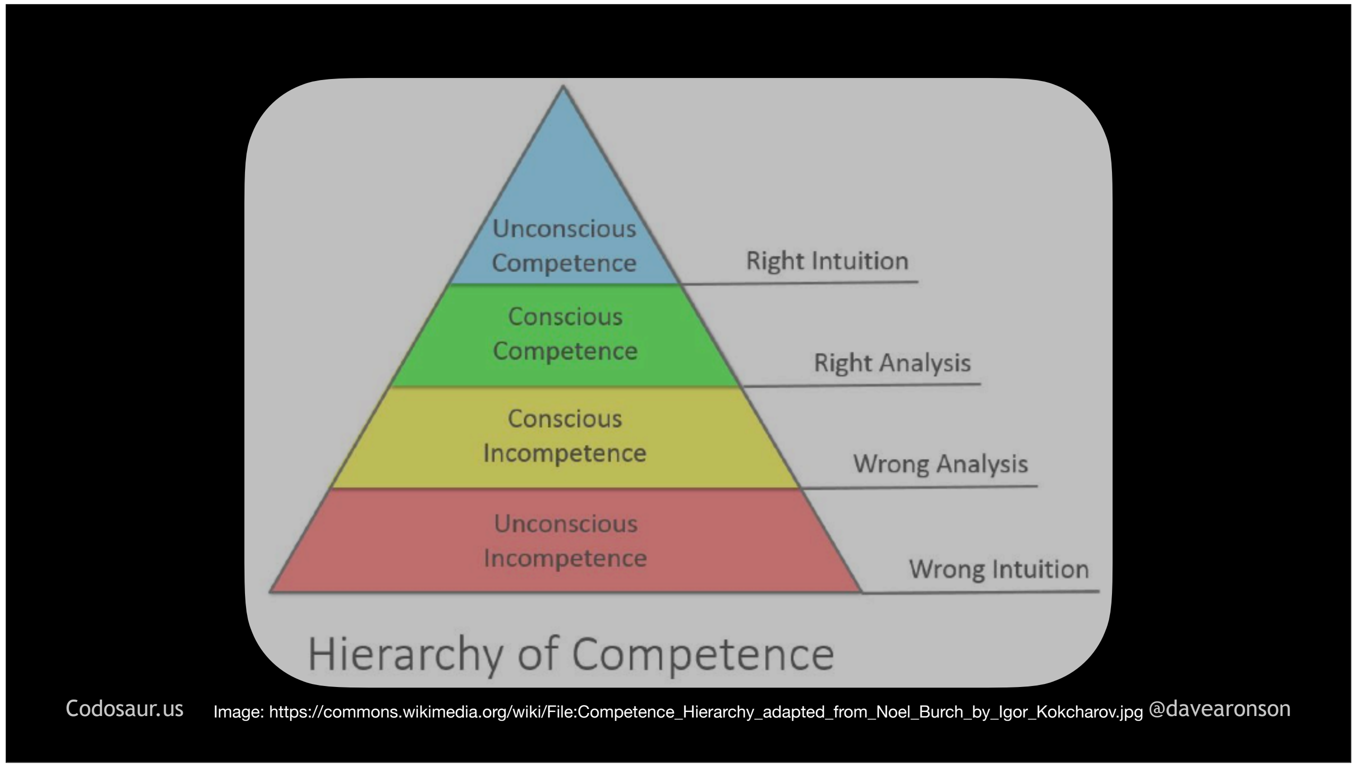
. . . good news! Over the past decade or so, there has been a lot of research on trimming down the number of mutants, mainly by weeding out ones that are semantically equivalent to the original code, redundant with other mutants, or trivial in various ways such as creating an obvious uncaught error condition. Such things have reduced the mutant horde down to about one third! But even with that rare level of success, it's still . . .

. . . no silver bullet, as this takes lots of CPU time itself -- and there are still quite a lot of mutants left to deal with.

The next question is, when making each mutant, why change it in only . . .

. . . one way?

There are multiple reasons. First off, the main theoretical underpinning of mutation testing is . . .



. . . the Competent Programmer hypothesis. Let's give that a quick check. Raise your hand if you're competent! (PAUSE!) Okay, looks like most of us. The rest of you, you probably really are competent, so you might want to read up on Impostor Syndrome.

Long story short, the Competent Programmer Hypothesis is the idea that we generally have a pretty good clue what we're doing, and when we make a mistake, it's usually a single small mistake, like adding when we should subtract, or comparing using "less than or equal" when we mean "strictly less than", or greater than, or whatever. Does this kind of simple substitution sound familiar? It's exactly what a mutation testing tool does! So we can think of mutation testing as sort of a "did you mean" function, like how Google suggests a different search if ours didn't have many hits.

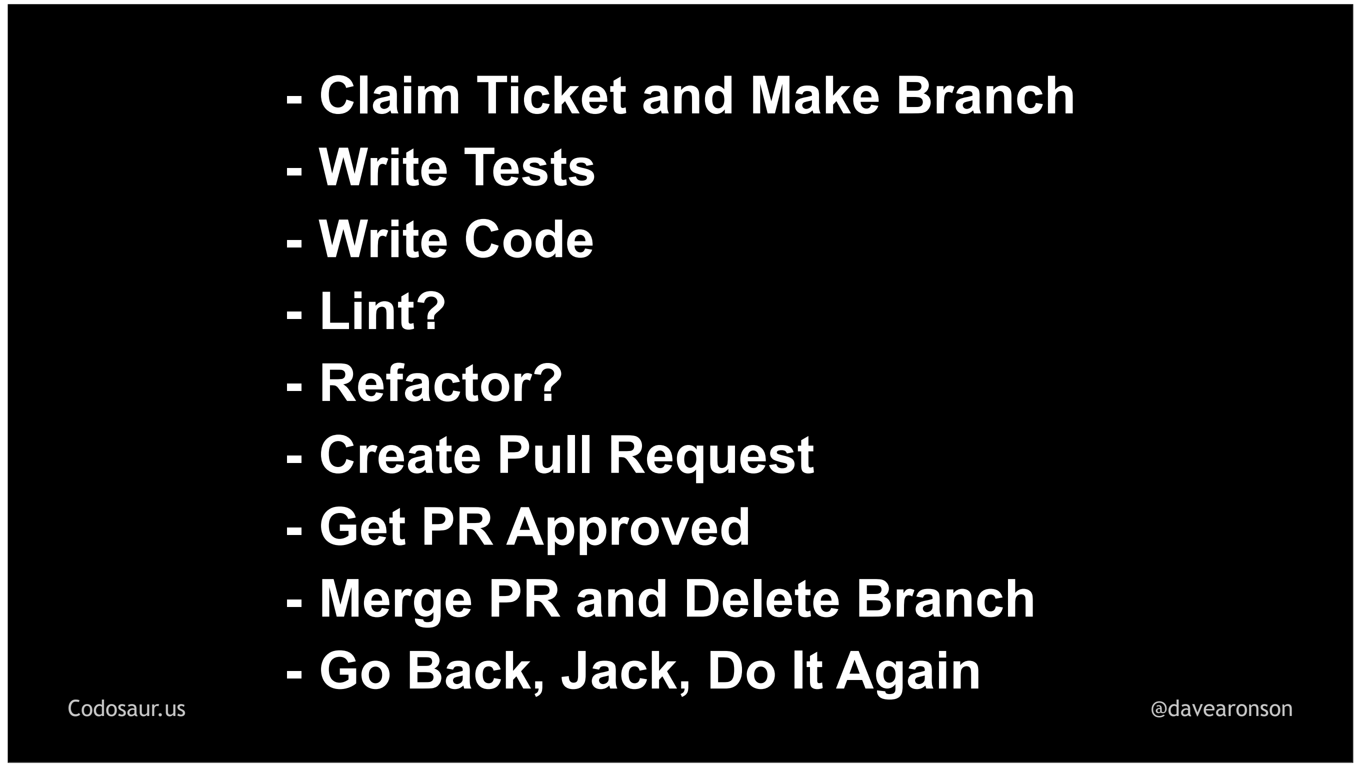
There are also practical considerations. For one, it helps us poor humans . . .

. . . FOCUS! It's much easier to tell what a surviving mutant is trying to say, if we're only talking about one thing in the first place. Another reason is that multiple changes may . . .

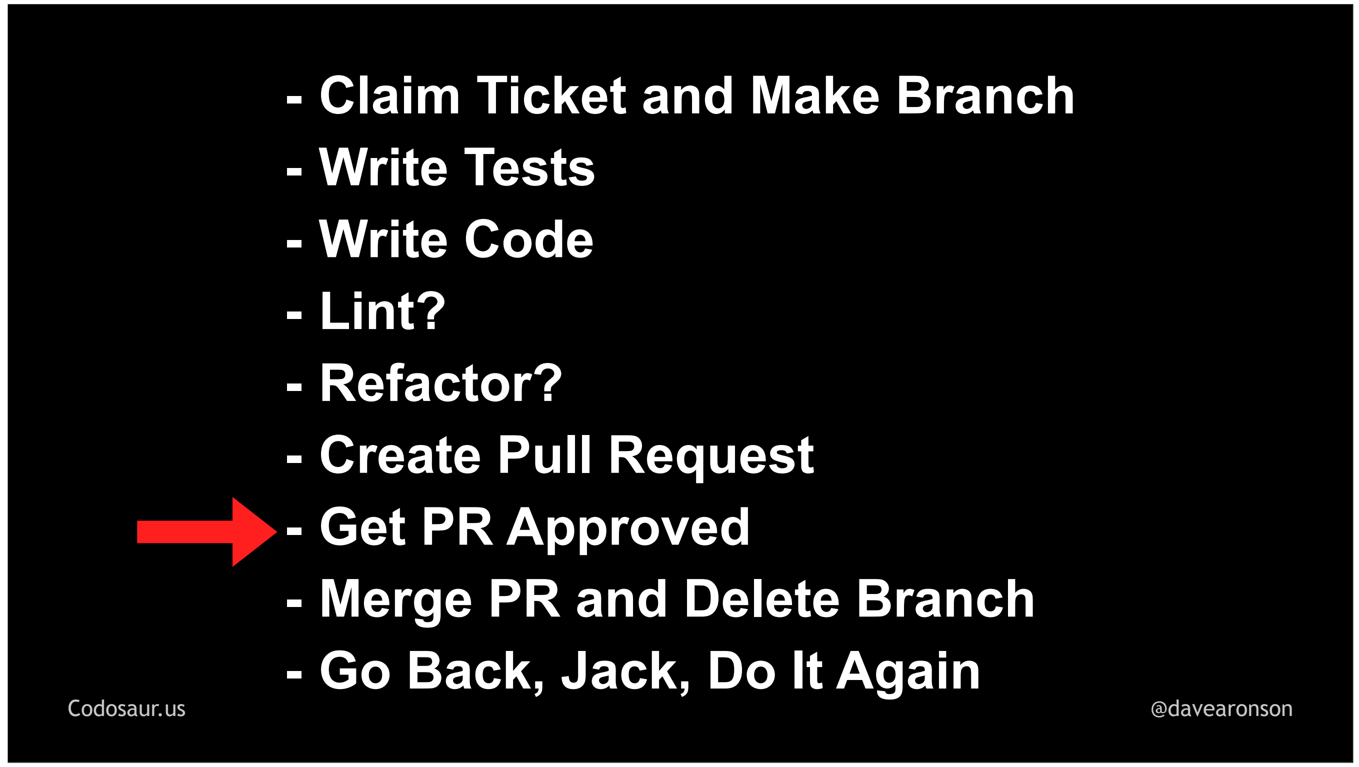
. . . balance each other out, leading to more false alarms.   Lastly, allowing multiple mutations would create a combinatorial . . .

. . . explosion of mutants, with the tool making multiple *orders of magnitude* more mutants per function, which would make it even *more* CPU- and memory-intensive.

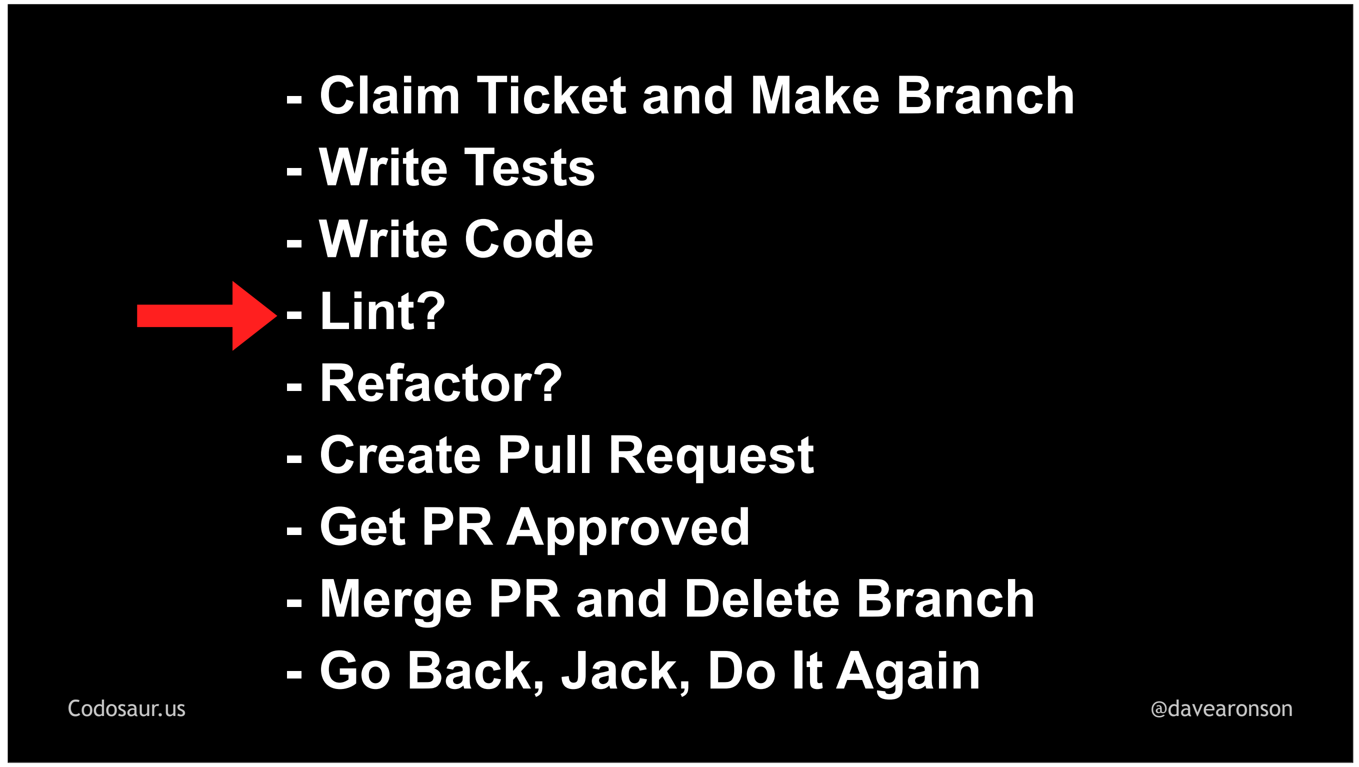
Lastly, a more practical question: where should we fit this into . . .



. . . our development process? There are many ways you can fit this in, and you don't have to limit yourself to just one. Primarily, I think it belongs at *least* . . .

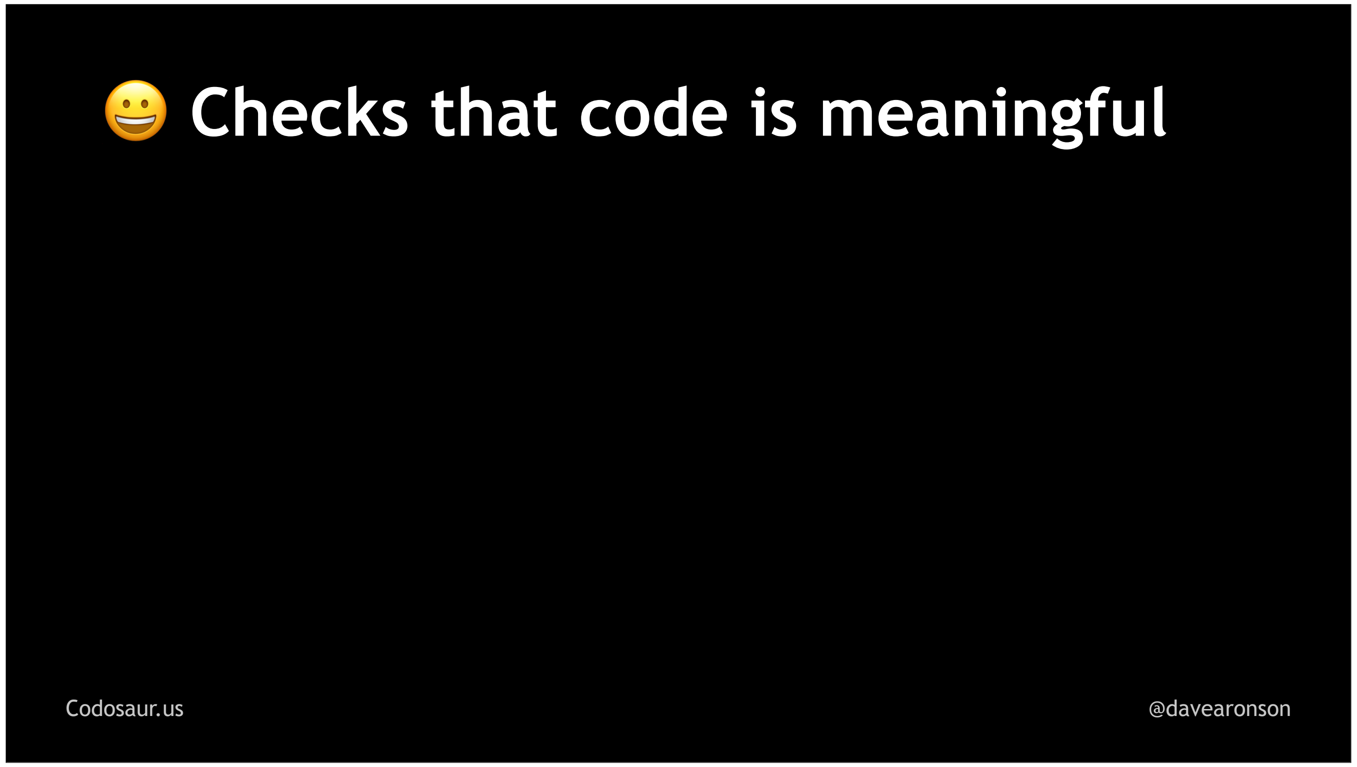


. . . here, as part of the requirements for a Pull Request (or whatever your process uses) to be approved. You can set some standards for what you're willing to tolerate, such as no surviving mutants on new code, and no *increase* on the whole codebase, to sort of rachet that down. Ideally this would be automated, as part of a CI pipeline, kicked off when the PR is created, that would block it if the criteria are not met. That said, I would also do it in my *own* work as part of . . .



. . . the Linting step, where I apply all sorts of tools to assess the quality of the code, its adherence to our group's agreed standards, and so on, before other people see it. *All* of that should *also* still be done in the CI pipeline, since, let's face it, most developers won't bother to do it on their own.

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



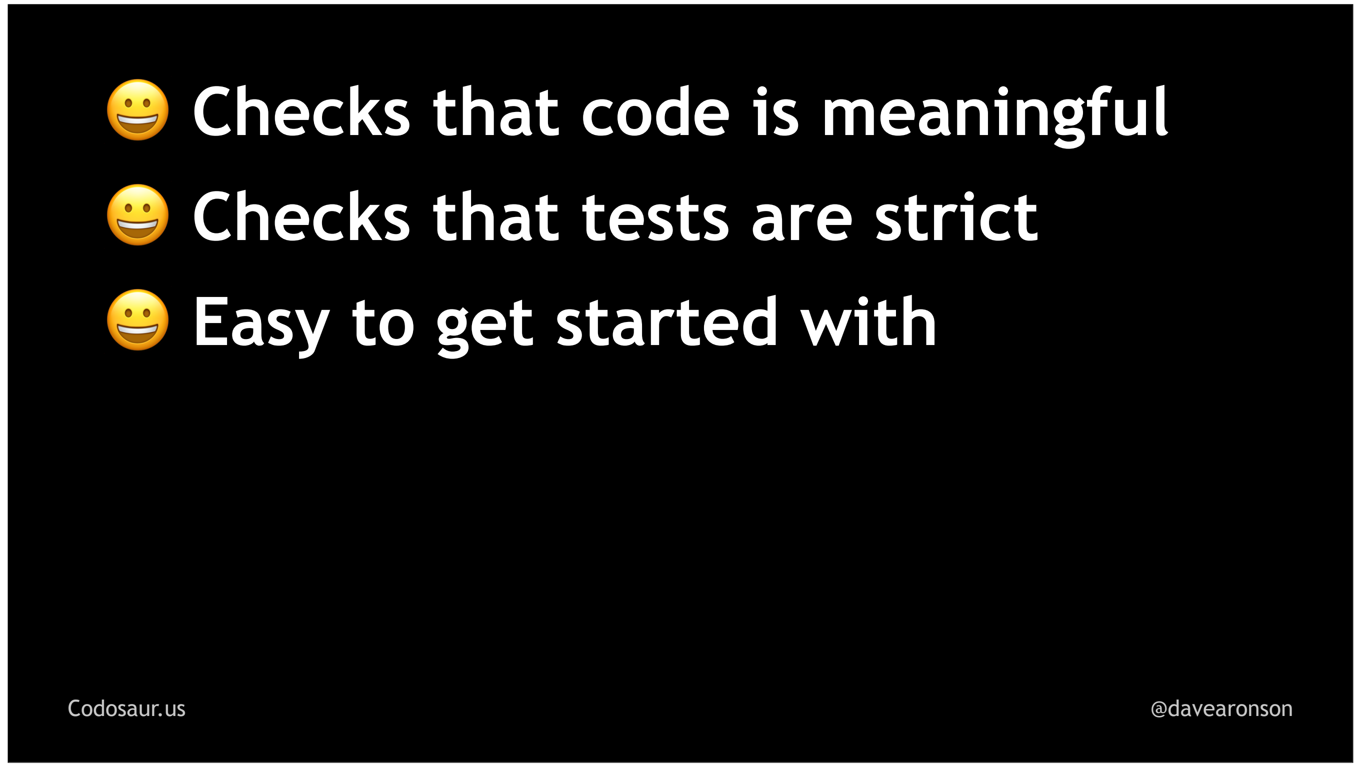
. . . help ensure that our code is meaningful and . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

Codosaur.us @davearonson

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



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**

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

😀 Checks that code is meaningful

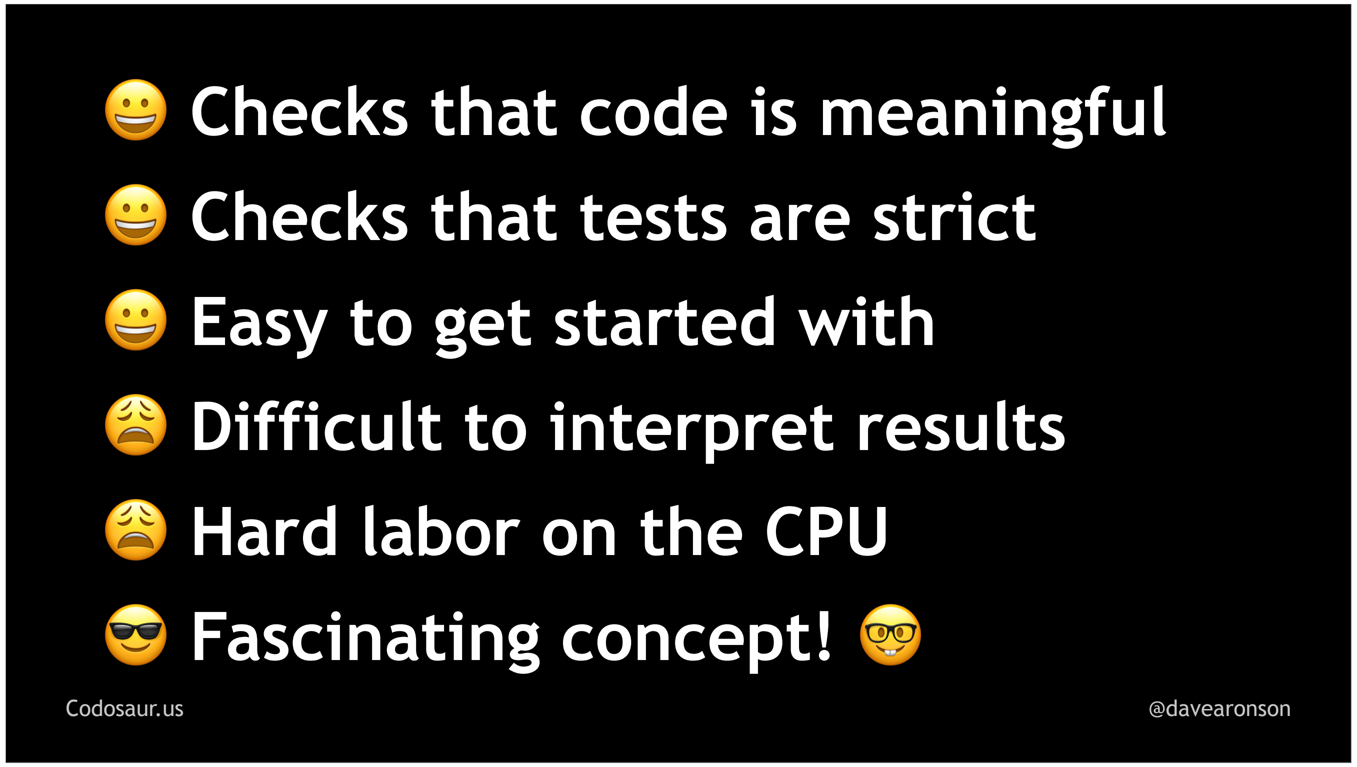
😀 Checks that tests are strict

😀 Easy to get started with

😩 Difficult to interpret results

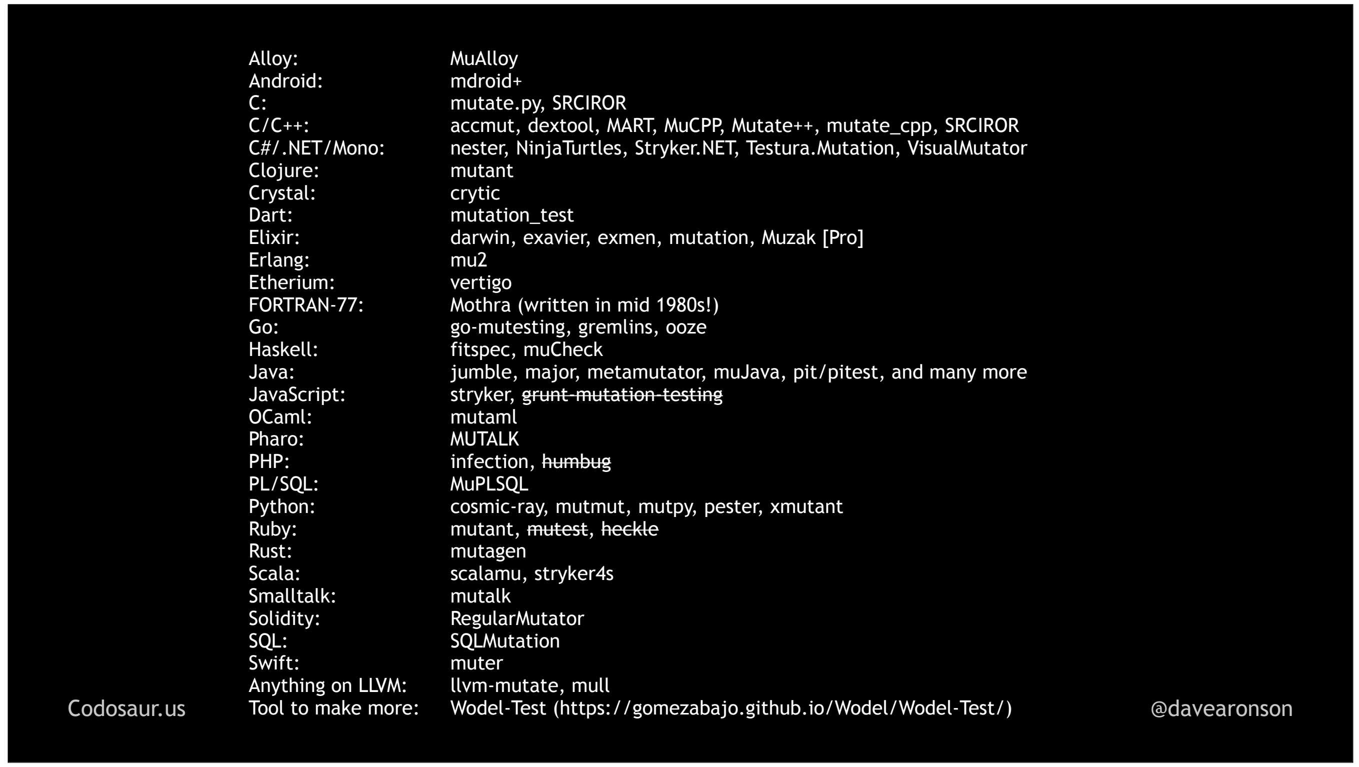
😩 Hard labor on the CPU

. . . 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 . . .



. . . 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. Don't bother trying to read all that now, the final slide has the URL for the whole deck. The tools I *know* are outdated, are crossed out, but I don't know or follow quite *all* those languages and tools.

To find out how a large tech company is using it for *real*, check out the next talk in this very room, where Lars Kempe and Clemens Bonse will be telling you how they use it at . . .

. . . Dolby, yes the sound people. Disclaimer: they have no connection to me or this talk, I just like promoting talks on my favorite topics.

And now . . .

. . . it's your turn! If you have any questions, I'll take them now, or if you think of anything later, there's my contact info, plus the URL for the slides, complete with a full script, which I've *mostly* stuck to. Any questions?