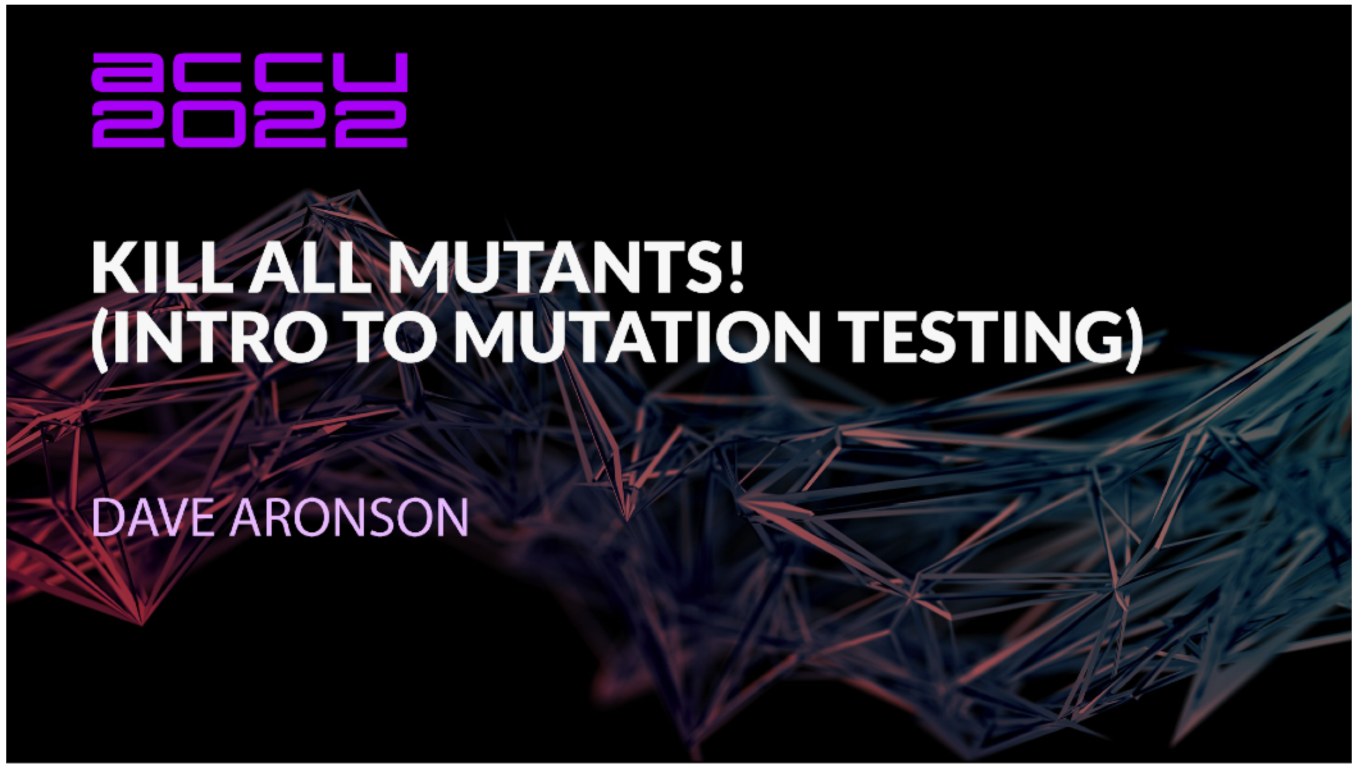


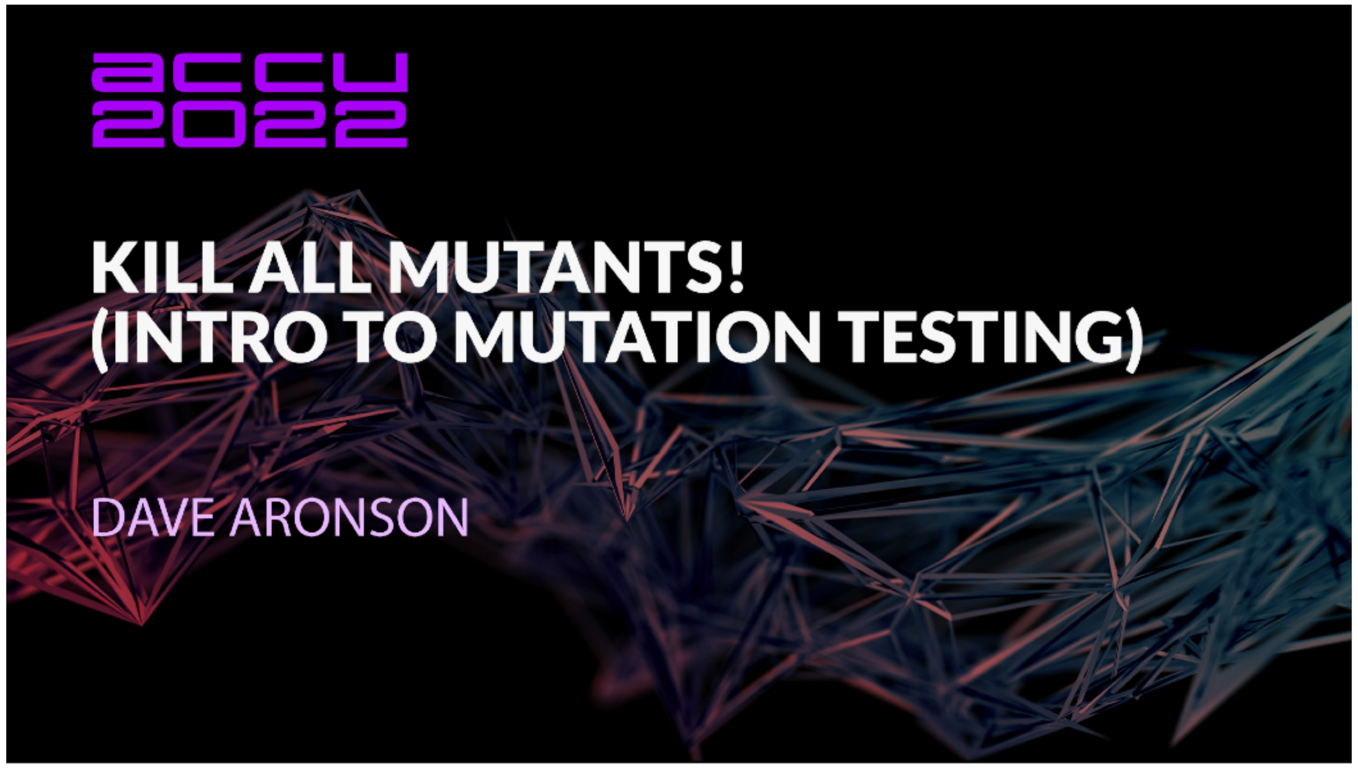
KILL ALL MUTANTS! (INTRO TO MUTATION TESTING)

accu 2022

DAVE ARONSON

CURRENT TOTAL TIME: ~1h20m

NOTES TO SELF:
- add example like codemanship's one about needing to handle invalid instruction
- REMEMBER TO Point out times where stricter typing would reduce range of mutations, especially value substitutions.



accu 2022

KILL ALL MUTANTS! (INTRO TO MUTATION TESTING)

DAVE ARONSON

GOOD MORNING BRISTOL!!! I hope you're not too hung over from last night's party! Oops, maybe you are, maybe I should do my Late Night Jazz Radio DJ impression. (very soft, low smooth) good morning, bristol. i hope you're not too hung over from last night's party. Naah, I can't do that, it would take too long and put you to sleep. Hi everybody, I'm Dave Aronson, the T. Rex of Codosaurus, and I flew all the way overhere on my pet pterodactyl to teach you to KILL MUTANTS!!!

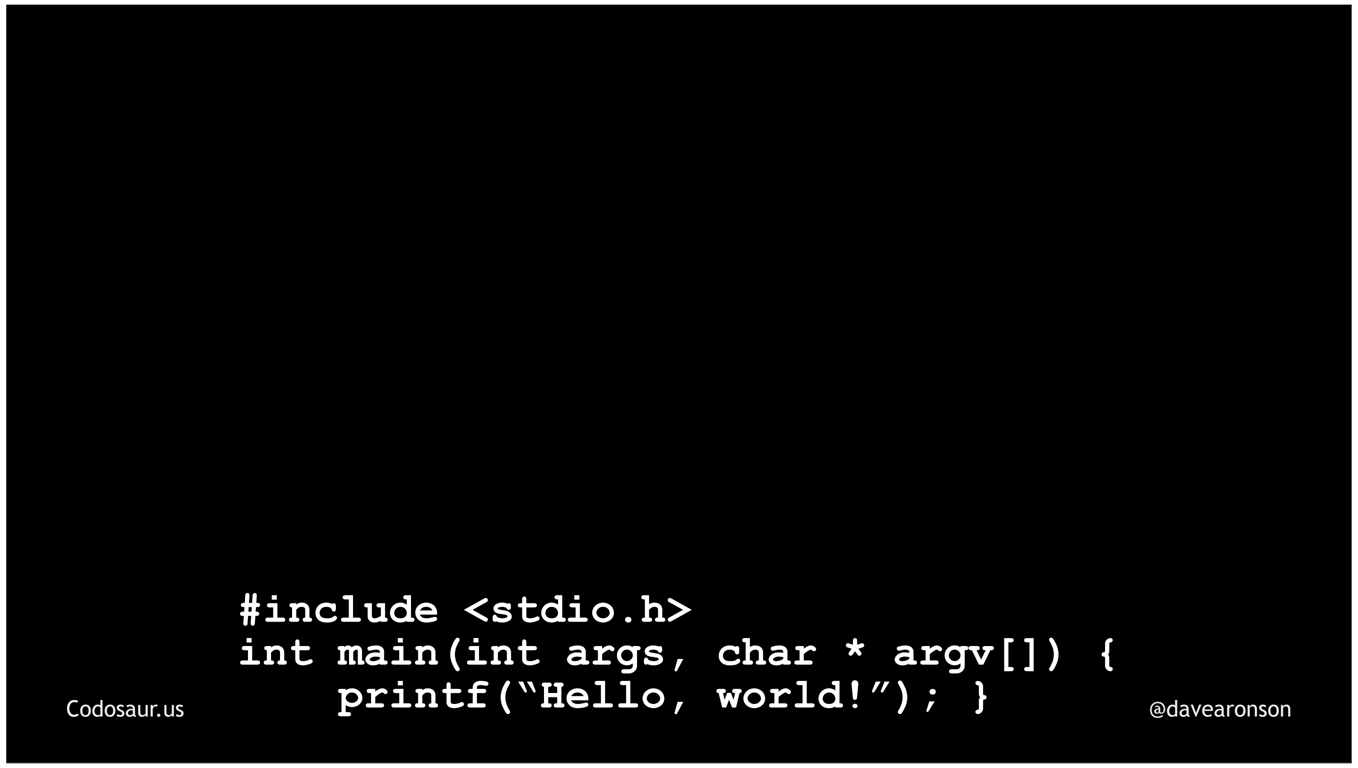
But first, a few caveats. First, this talk is about the overarching concepts of mutation testing, rather than using any particular . . .

Codosaur.us

Image: https://publicdomainvectors.org/en/free-clipart/Hammer-and-spanner-tools-vector-image/8108.html

@davearonson

. . . tools. So, there’s not very much code, just a few snippets to test, some tests, and a bit of supporting configuration. You could say it’s . . .



```
#include <stdio.h>
int main(int args, char * argv[]) {
    printf("Hello, world!"); }
```

Codosaur.us

@davearonson

. . . low-code.

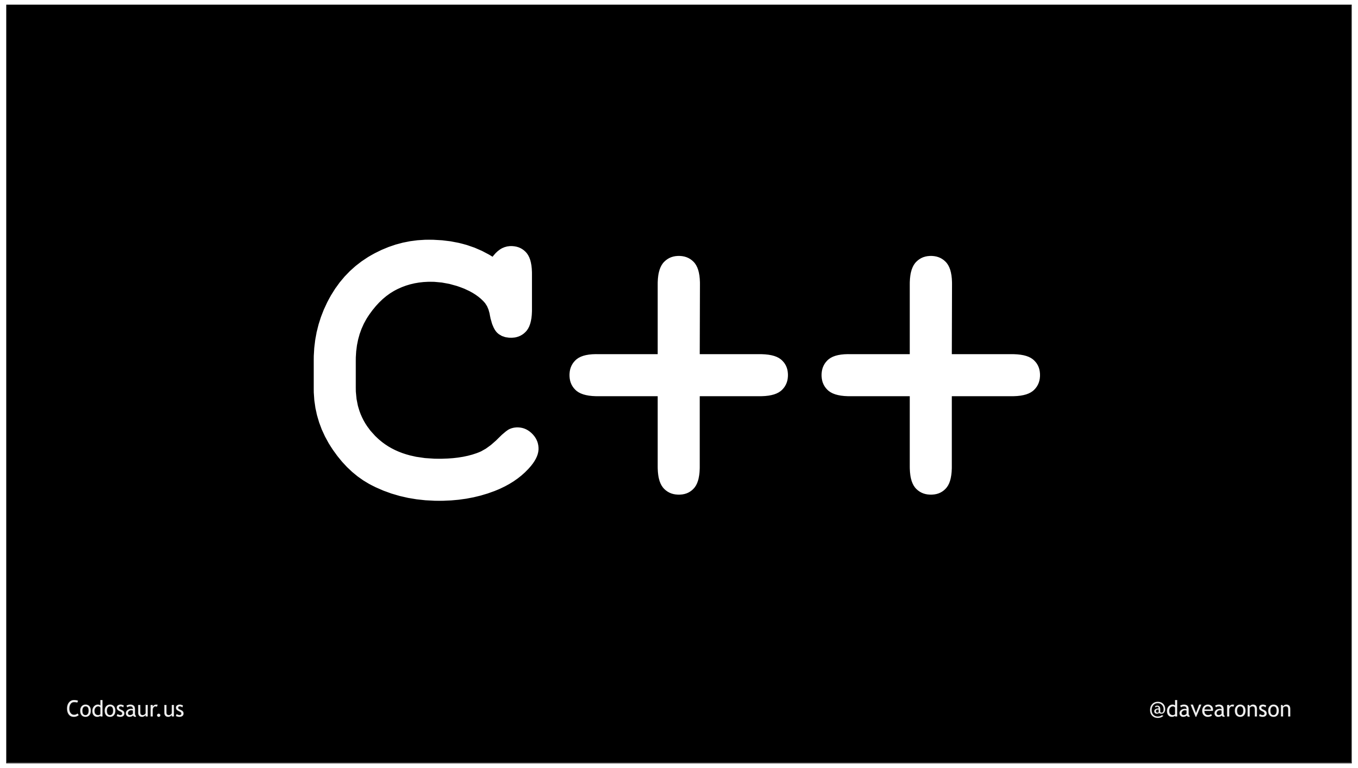
Second, I know this is mainly a . . .

C

Codosaur.us

@davearonson

. . . C, and even more so, . . .



C++

Codosaur.us

@davearonson

. . . C++ conference, and I do have a lot of experience in C, and a bit in C++, but all that was a looong time ago. So, what code there is, is in . . .

Codosaur.us

Image: shamelessly swiped from https://blog.devgenius.io/ruby-newbie-f1eb87795b52 which has no license details dagnabit!

@davearonson

. . . Ruby, the main language I've been using for the past thirteen years. If you object to the code being in another language, you can think of it as being in pseudocode, since Ruby is very readable, close to plain English. I'm not here to evangelize for Ruby, but I'm quite confident you'll understand it very easily.

Next, I'd like to . . .

Codosaur.us

Image: https://pixabay.com/photos/level-tool-equipment-repair-2202311/

@davearonson

. . . level-set some expectations.

This talk is listed on the agenda as an advanced talk, which could be considered a bit misleading. Calling it a beginner-level talk, however, would be even more misleading. What this is, is an *introduction*, *to* an *advanced topic*. So, if you're already well versed in mutation testing, I won't be *too* offended if you go seek better learning opportunities at another talk. But, I'd still prefer you stick around, so you can correct my mistakes . . . later . . . in private.

Second, I mention mistakes, because I do not consider myself an . . .

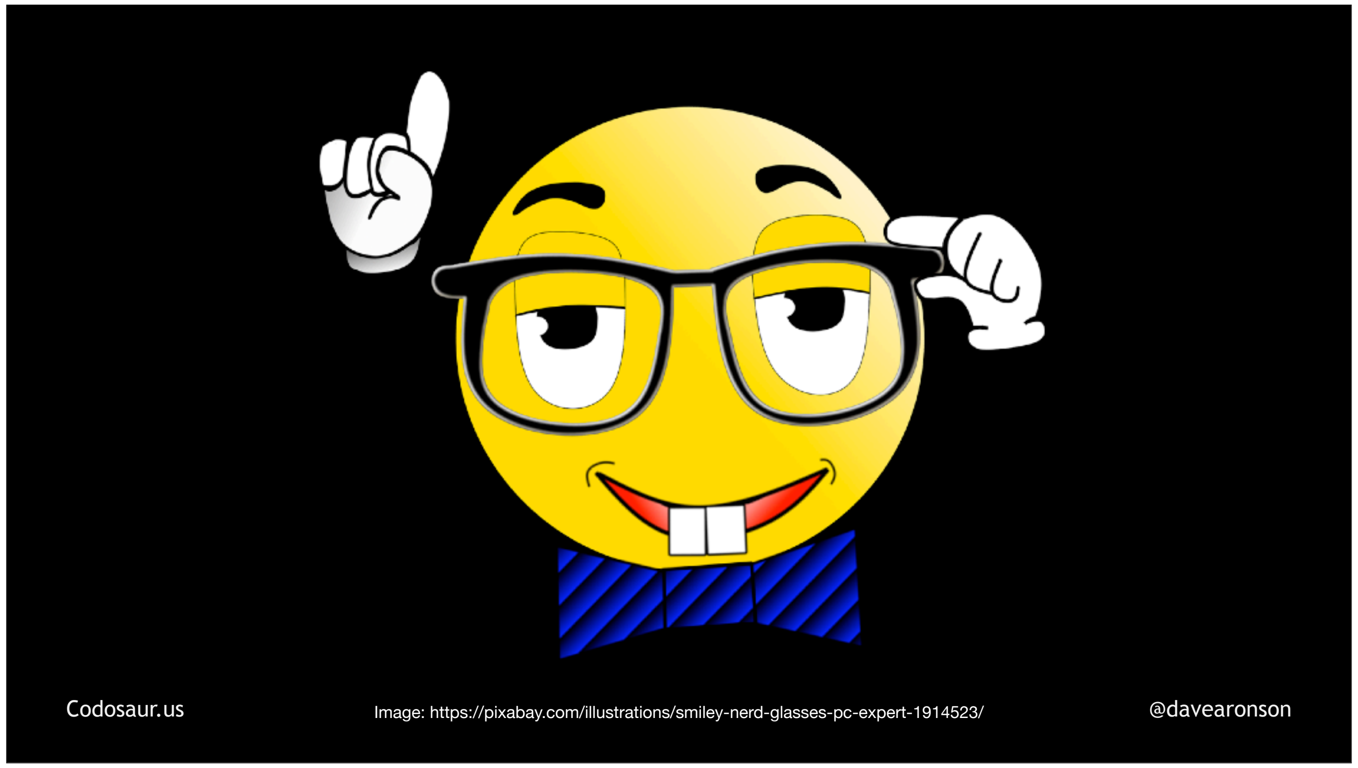

Codosaur.us

Image: https://pixabay.com/illustrations/smiley-nerd-glasses-pc-expert-1914523/

@davearonson

. . . expert on mutation testing. One of the dirty little secrets of public speaking is that you don't really have to be an expert on your topic! You just have to know a *little bit more* about it than the audience does, enough to make it worth their time to listen to you, and be able to *convey* it to them. And mutation testing is still rare enough, that most developers have never even *heard* of it!

So let's start with the basics. What on Infinite Earths is . . .

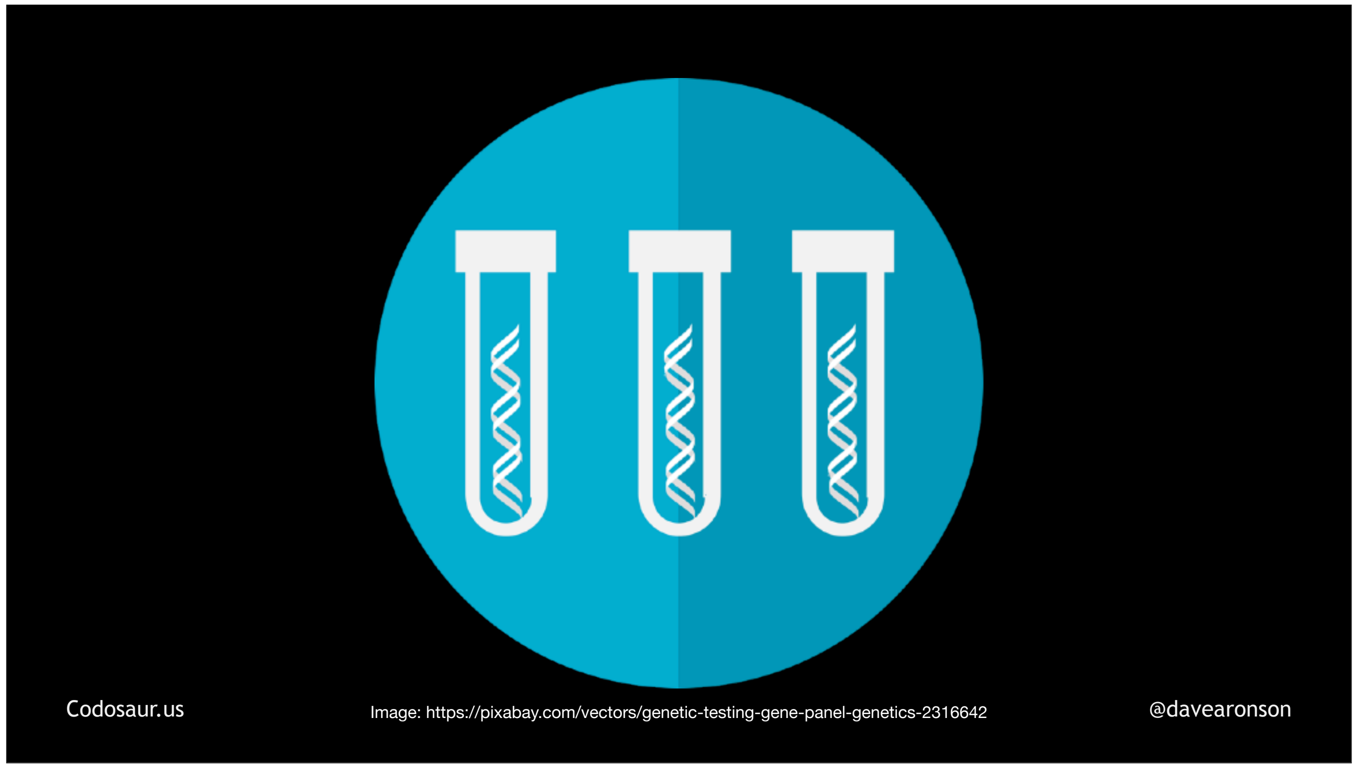
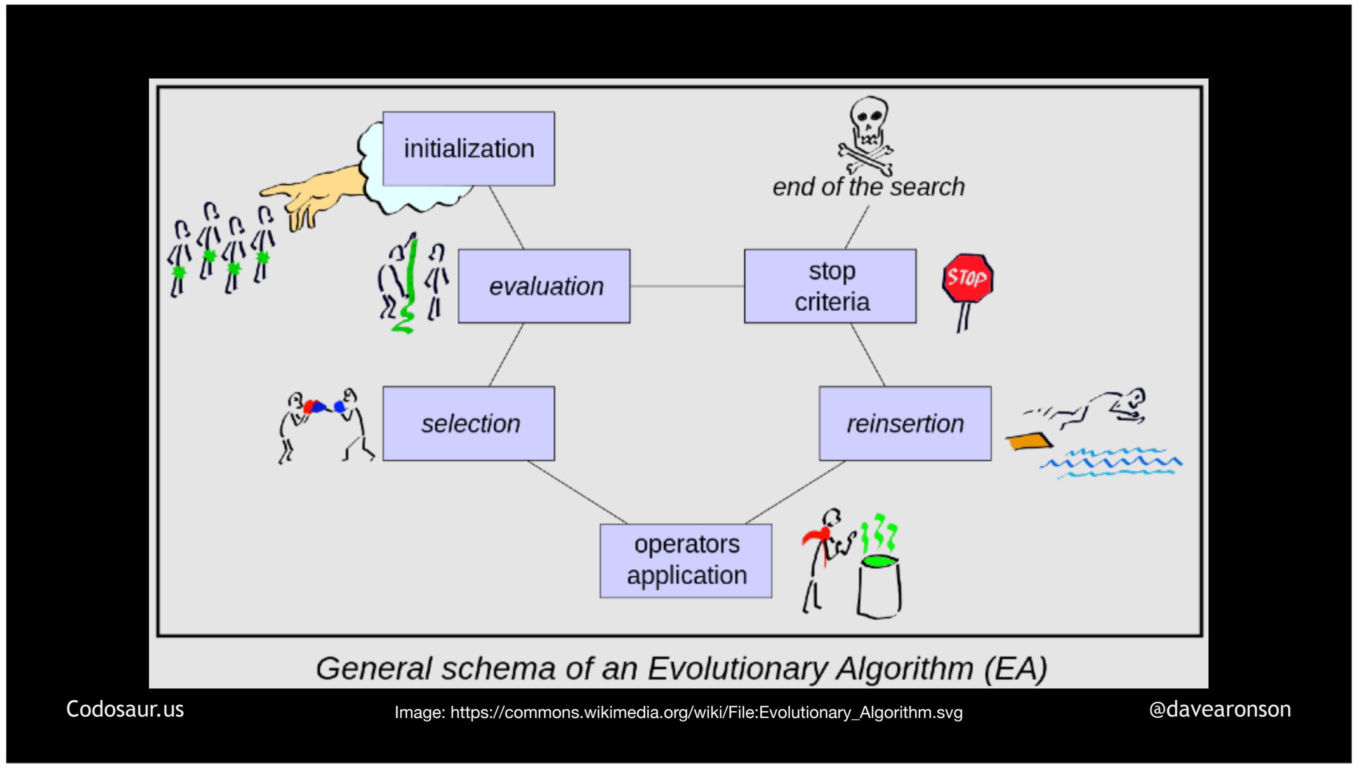

Codosaur.us

Image: https://pixabay.com/vectors/genetic-testing-gene-panel-genetics-2316642

@davearonson

. . . mutation testing? In *our* universe, that of software development, not comic books, it's a software testing technique. (Surprise!) But why is *this* software testing technique different from all *other* software testing techniques? One might look at the name and think "well *obviously* it's about testing the mutations used in . . .



Codosaur.us

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

@davearonson

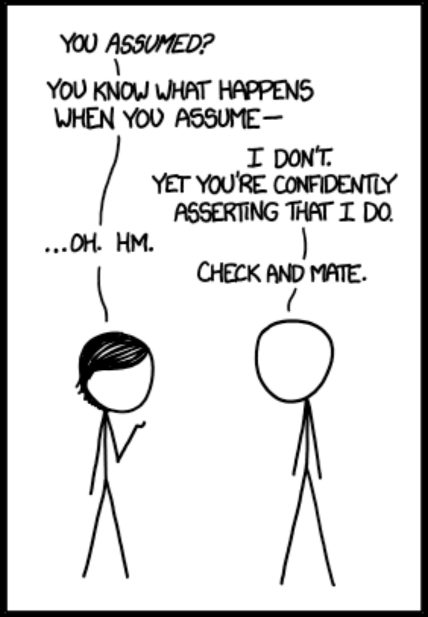
. . . genetic algorithms!" But no, that's not correct. The big difference is that most other software testing techniques are about checking whether or not our code . . .

Codosaur.us

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

@davearonson

. . . is correct. Mutation testing . . .

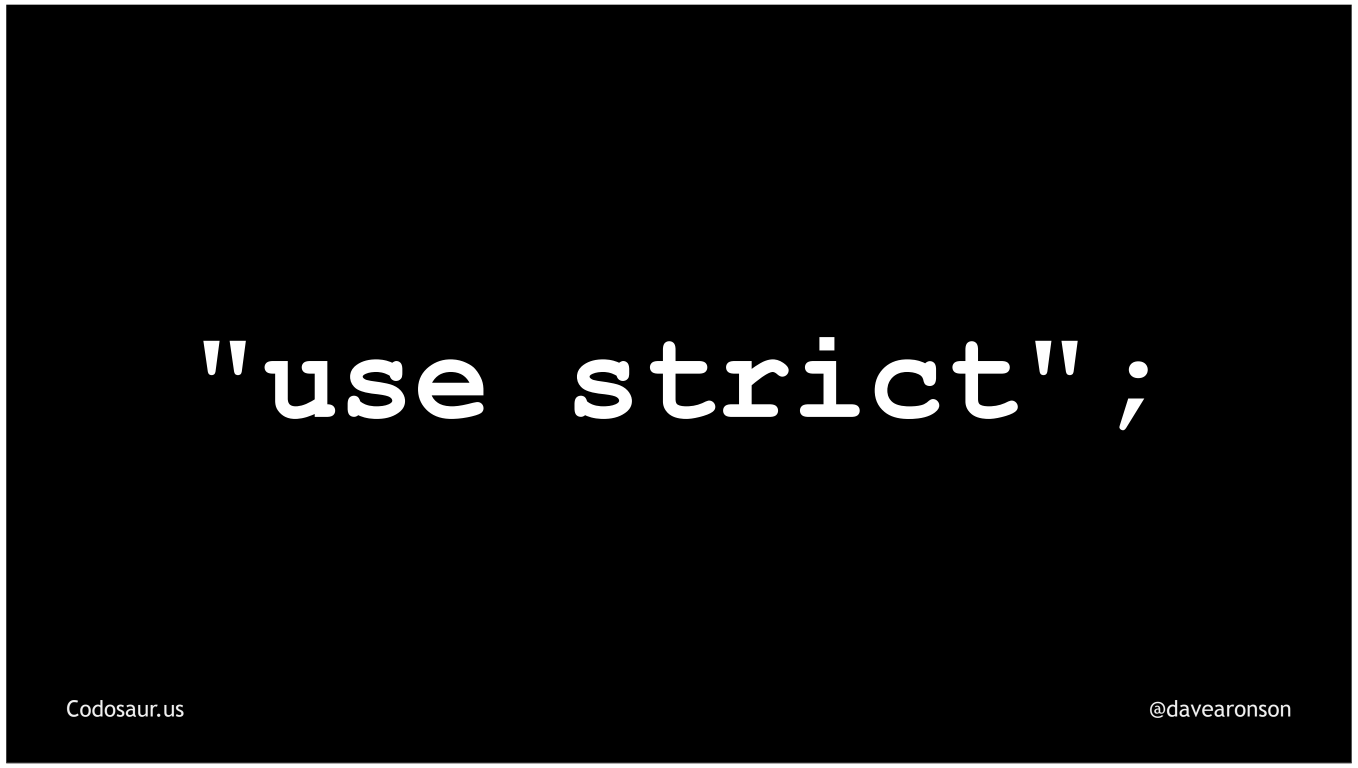

YOU ASSUMED?
YOU KNOW WHAT HAPPENS WHEN YOU ASSUME—
I DON'T. YET YOU'RE CONFIDENTLY ASSERTING THAT I DO.
...OH. HM.
CHECK AND MATE.

Codosaur.us

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

@davearonson

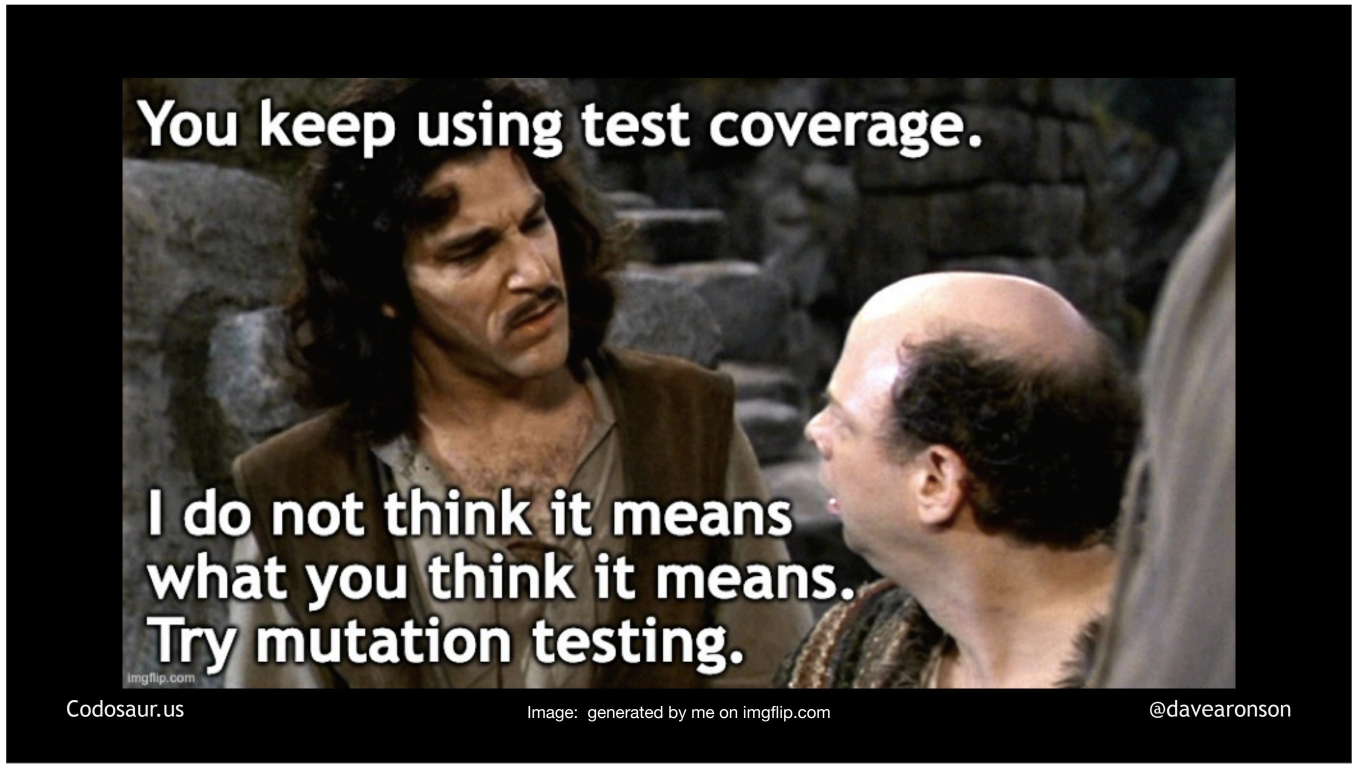
. . . *assumes* that our code is correct, at least in the sense of passing its tests. (This sort of means it *also* assumes that we *already have* tests. More on that later.) And I *figure* you *probably* know what happens when you ASSUME, since I first learned it from Benny Hill! Anyway, mutation testing is instead about checking for two *different* qualities. One is about our regular production code, as you might expect, but the other is in fact about our *test suite!* In my opinion, the more *important, interesting,* and *helpful* of these two qualities is that our test suite is . . .



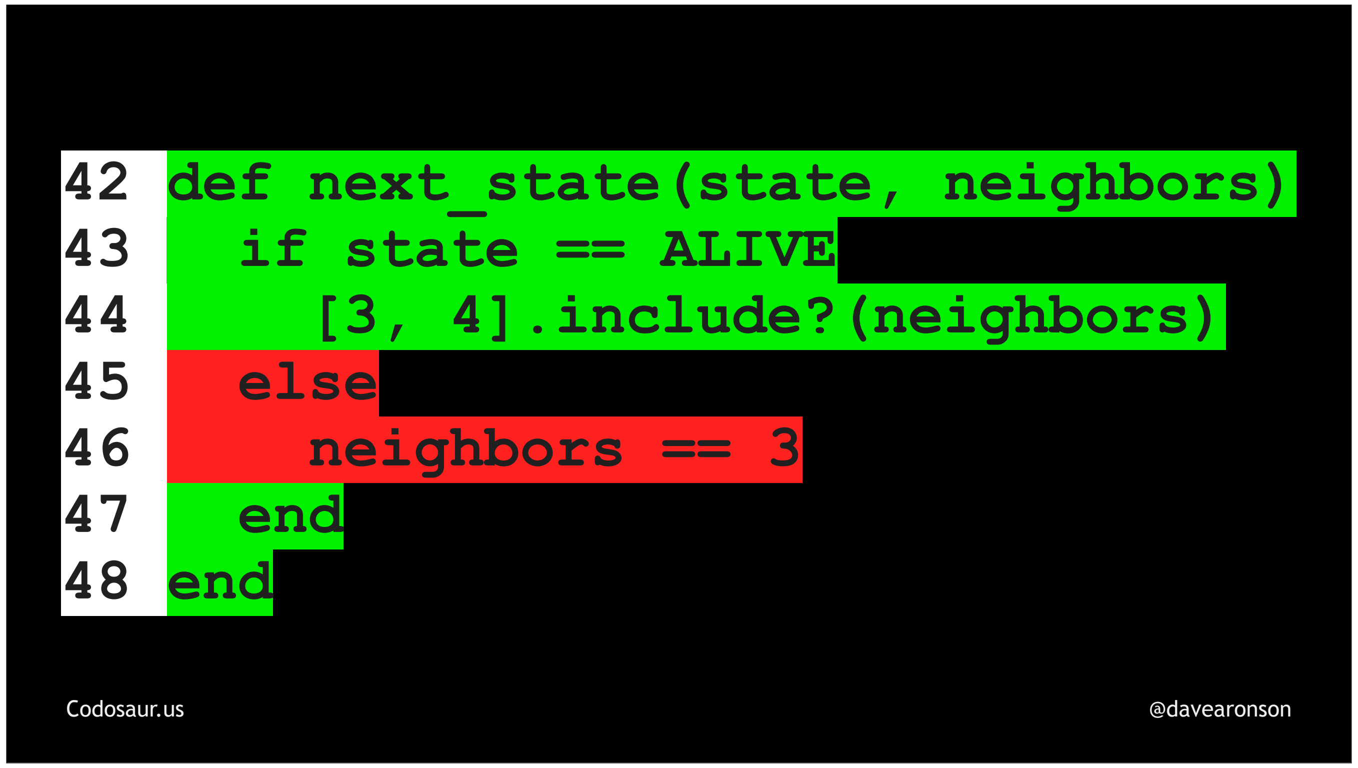
```
"use strict";
```

Codosaur.us @davearonson

. . . *strict*. Now you may be thinking, “But Dave, isn’t that what code coverage is for? If we have 100% code coverage, doesn’t that mean that all the code is fully tested?” Long story short . . .


You keep using test coverage.
I do not think it means what you think it means. Try mutation testing.
imgflip.com
Codosaur.us
Image: generated by me on imgflip.com
@davearonson

. . . NO! (PAUSE!) Phil Nash mentioned in his talk on software quality, right here on Wednesday afternoon, that test coverage is misleading because it shows lines of code, not data paths. To expand on that a bit, the *only* thing that code coverage tells us . . .



```
def next_state(state, neighbors)
  if state == ALIVE
    [3, 4].include?(neighbors)
  else
    neighbors == 3
  end
end
```

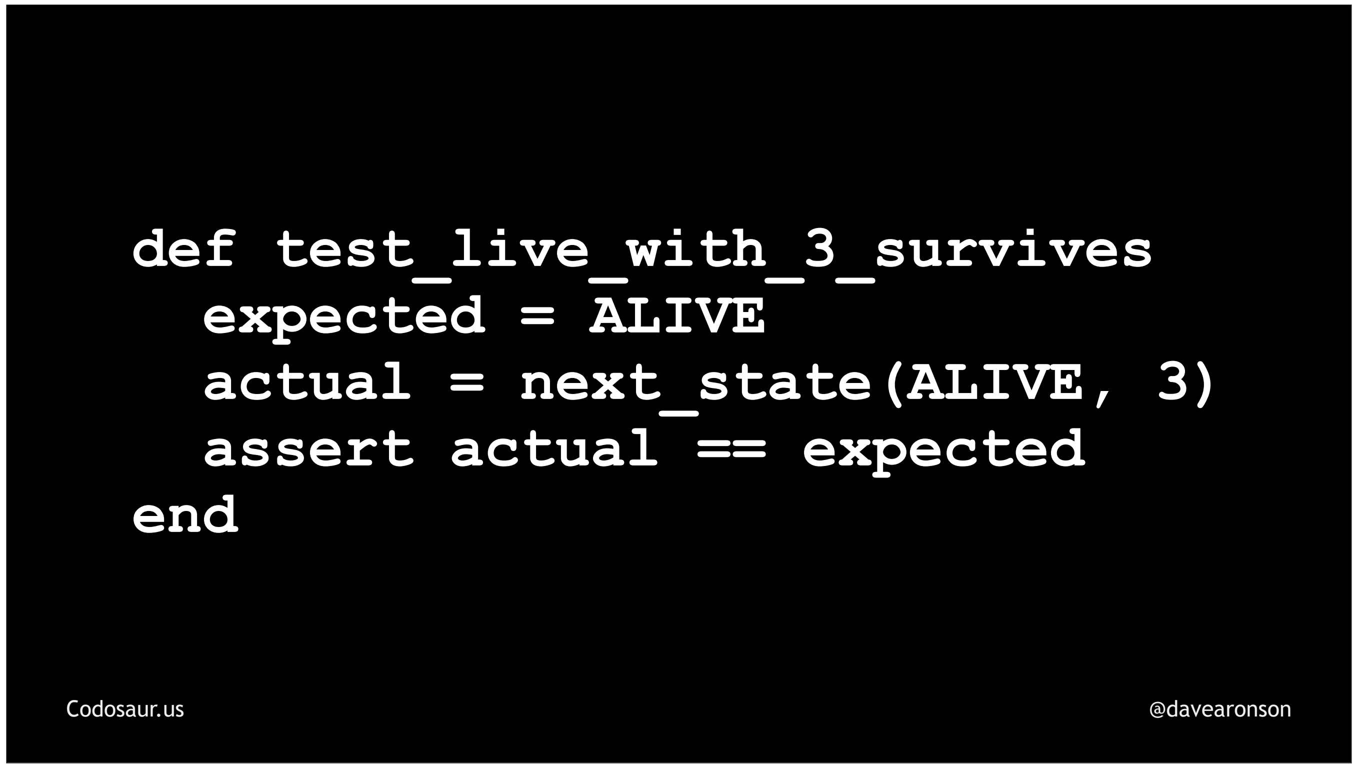
Codosaur.us

@davearonson

. . . is that at least one test *executed* that code. It tells us NOTHING about whether the *correctness* of that code made *any difference* to whether the test passed or failed.

Let that sink in for a moment. It's the fundamental flaw in relying on code coverage. To recap: code coverage only tells us that *some* test *executed* that green code, NOT whether the *correctness* of that code made *any difference* to whether *any* test passed or failed.

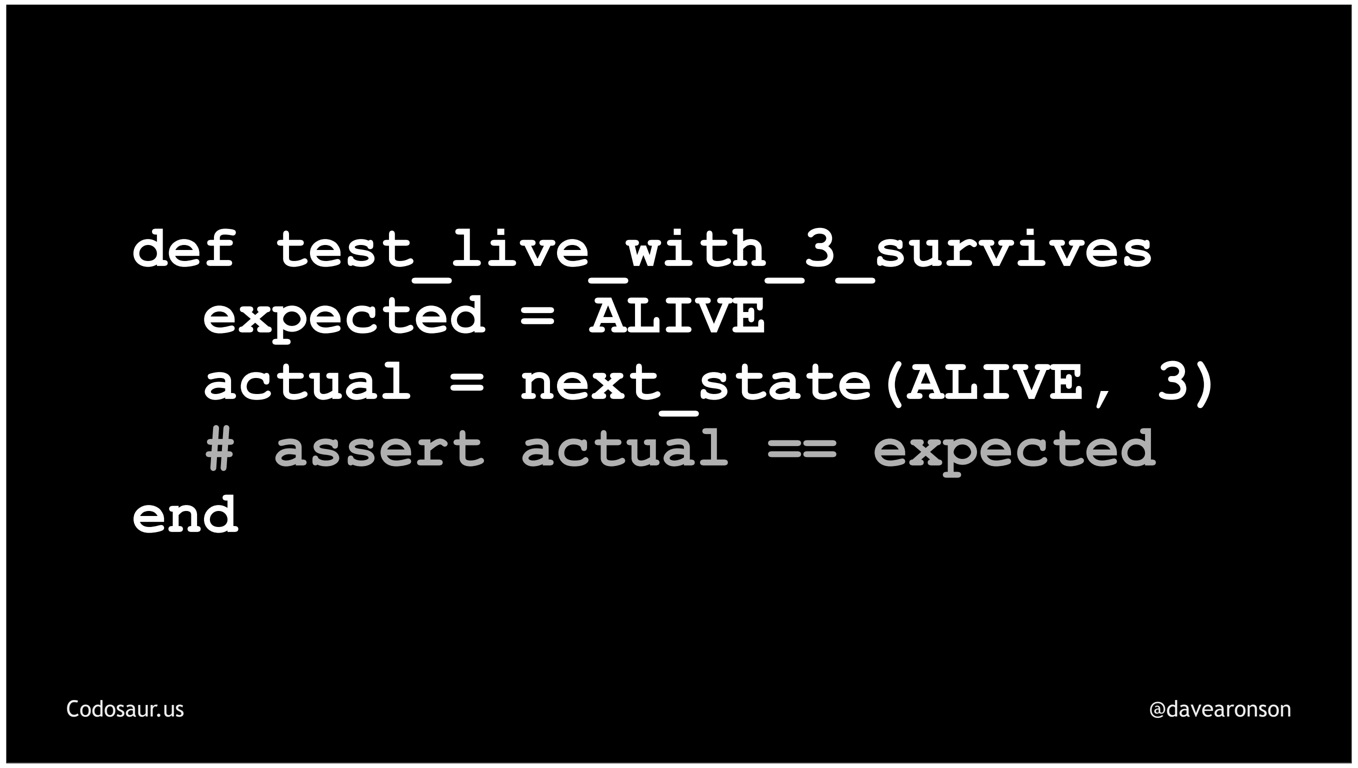
By way of illustration, let's look at . . .



```
def test_live_with_3_survives
  expected = ALIVE
  actual = next_state(ALIVE, 3)
  assert actual == expected
end
```

Codosaur.us @davearonson

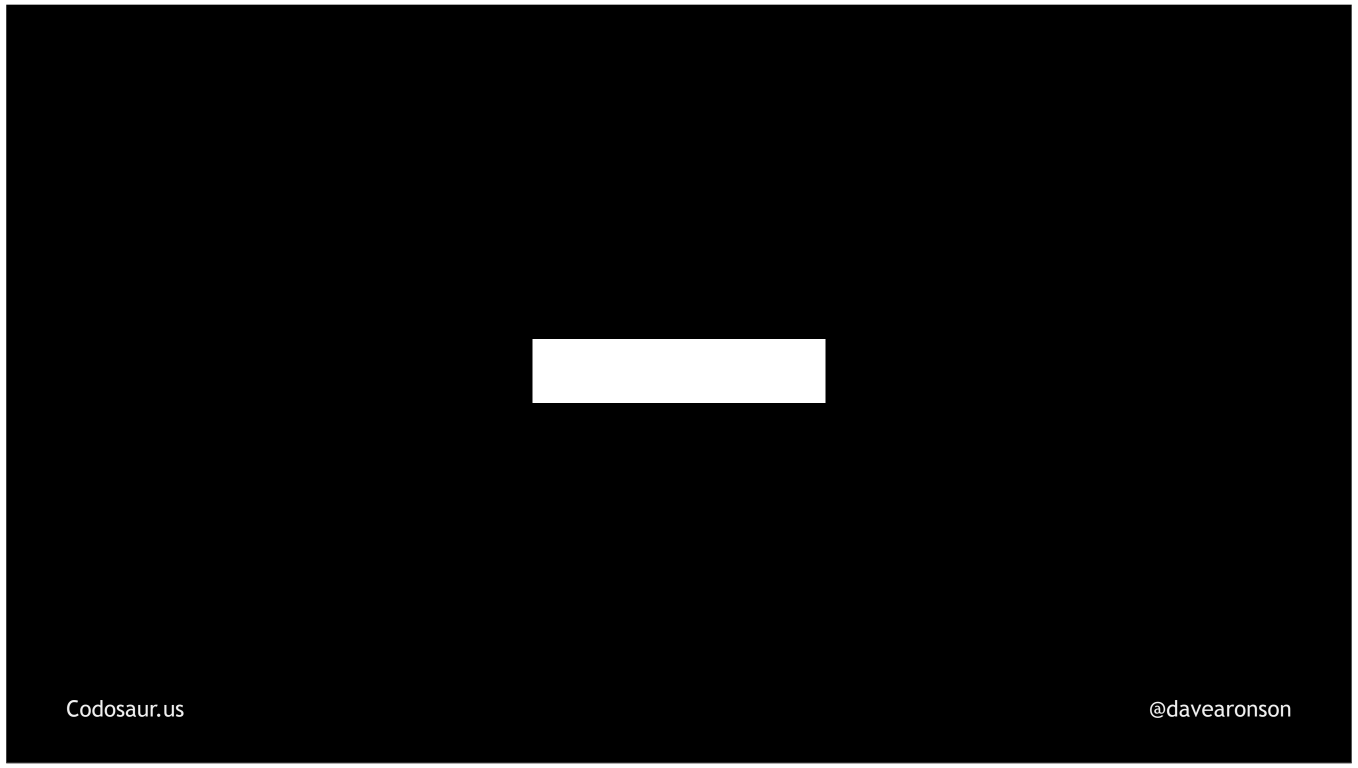
. . . a possible test for the code from the prior slide. It's pretty straightforward, once you realize what the code does. It's calculating the next state of a given grid cell, in Conway's Game of Life. If you're not familiar with that, don't worry, we're not going to delve into it, I just want you to think about what happens if we do . . .



```
def test_live_with_3_survives
  expected = ALIVE
  actual = next_state(ALIVE, 3)
  # assert actual == expected
end
```

Codosaur.us @davearonson

. . . *this*, to comment out the assertion. Our test still *runs* the function, but we're throwing away the answer, not bothering to check it. (Let's leave aside any quibbles about whether our test framework should even *let* us do this.) This may seem like an unrealistic thing to do, so let's take a poll -- who's actually *seen* assertions commented out, or removed, because a test was failing? I'm not asking who's *done* it, just who's *seen* it, so no shame! It's pretty easy to imagine cases where the assertions were not even written in the first place, because of any number of causes, like sloppiness, inability to figure out just *how* to obtain or check the result, or because the tests were only written to game the system and satisfy some pointy-haired manager's demand for high test coverage. *Any* of these scenarios, amply demonstrate the fallacy of relying on coverage. It's what I call a . . .



. . . negative indicator, more useful in its *absence* than its presence. The code that *isn't* covered, is *definitely not* tested. That is a useful or at least usable fact. But what about the rest of it, the parts that *are* covered? Is that tested? Who knows? It might be properly tested, or poorly tested, or not tested at all!

So how *can* we tell if the covered code being correct or not, made any difference to whether the test passes or fails? That . . . is where mutation testing comes in.

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

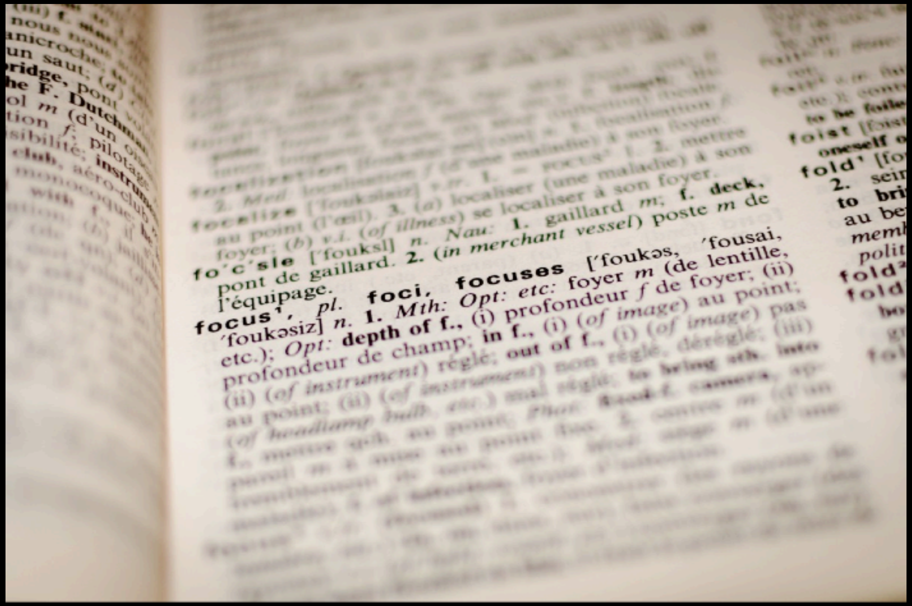
Codosaur.us

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 improve our test suite, by either adding tests or improving existing tests, to close the gaps. Lack of strictness comes mainly from *lack* of tests, poorly *written* tests, or poorly *maintained* tests, such as ones that didn't keep pace with changes in the code.

Speaking of which, the other thing mutation testing checks is that our code is . . .



Codosaur.us

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

@davearonson

. . . *meaningful*, by which I mean that any change to the code, will produce a noticeable change in its behavior. Lack of *meaningfulness*, or *meaning* if you *are* into the whole brevity thing, comes mainly from code being unreachable, redundant with other code, or otherwise just not having any real effect. Once we find "meaningless" code, we can make it meaningful, if that fits our intent, or just remove it.

Mutation testing . . .

Codosaur.us

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

@davearonson

. . . puts these two together, by checking that every possible tiny little change to the code does indeed result in a noticeable change to its behavior, *and* that the test suite is indeed strict enough notice that change, and fail. Not all of the tests have to fail, but each change should make *at least one* test 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 . . .

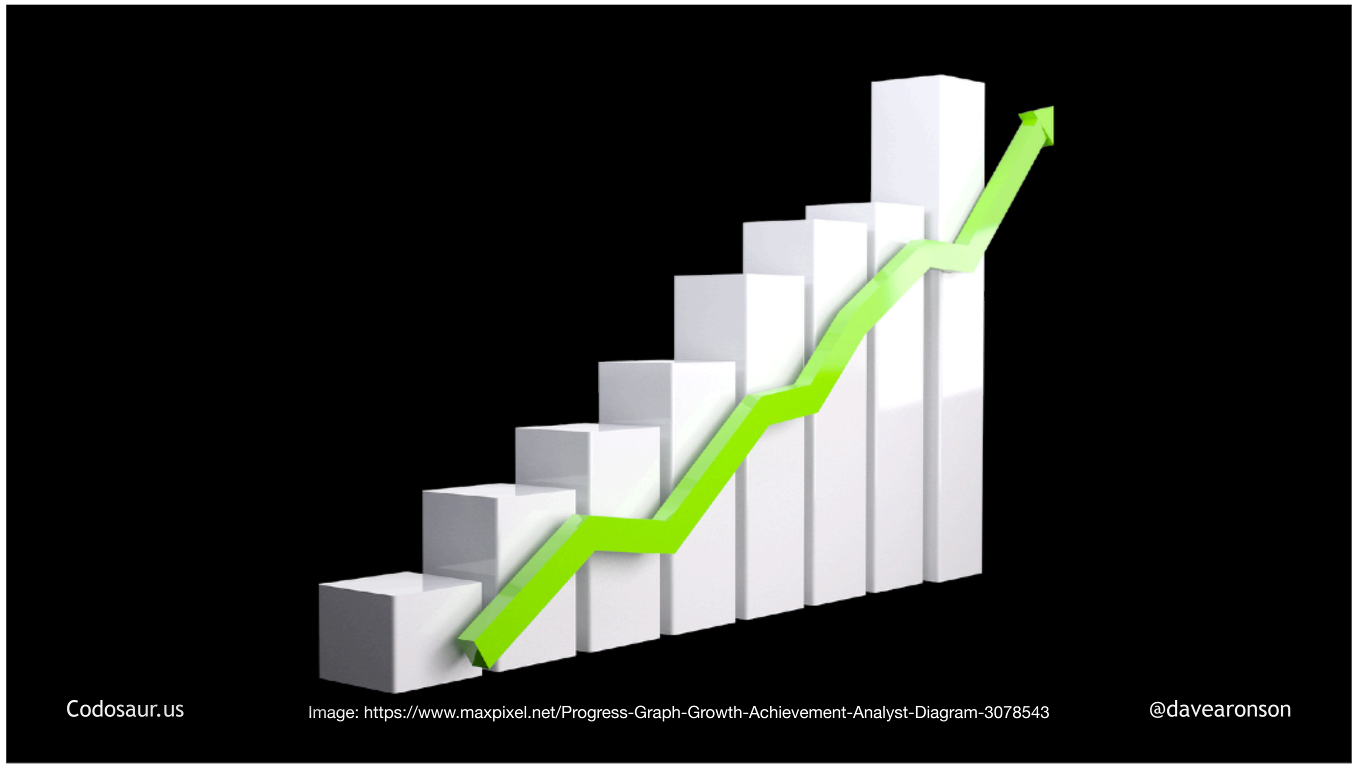
Codosaur.us
Image: https://www.publicdomainpictures.net/en/view-image.php?image=199986
@davearonson

. . . 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 overnight for a larger one, maybe even a weekend. Fortunately, most tools let us just check specific files, classes, modules, functions, and so on, plus they usually include some kind of . . .


Codosaur.us
Image: https://www.maxpixel.net/Progress-Graph-Growth-Achievement-Analyst-Diagram-3078543
@davearonson

. . . incremental mode, so that we can test only whatever has changed since the last mutation testing run, or the last git commit, or the difference from the main git branch, or some other such milestone. That, maybe we can do on each save, for a very small system, or at least over a shorter break for the rest.

Also, its CPU-intensive nature can really . . .

Codosaur.us

Image: https://pixabay.com/illustrations/clouds-dollar-symbol-characters-96588/

@davearonson

. . . run up our bills on cloud platforms such as AWS or Azure! (Or aZURE, however you PROnounce it.)

Another drawback is that mutation testing is . . .

Codosaur.us

Image: https://www.flickr.com/photos/ell-r-brown/5866767106

@davearonson

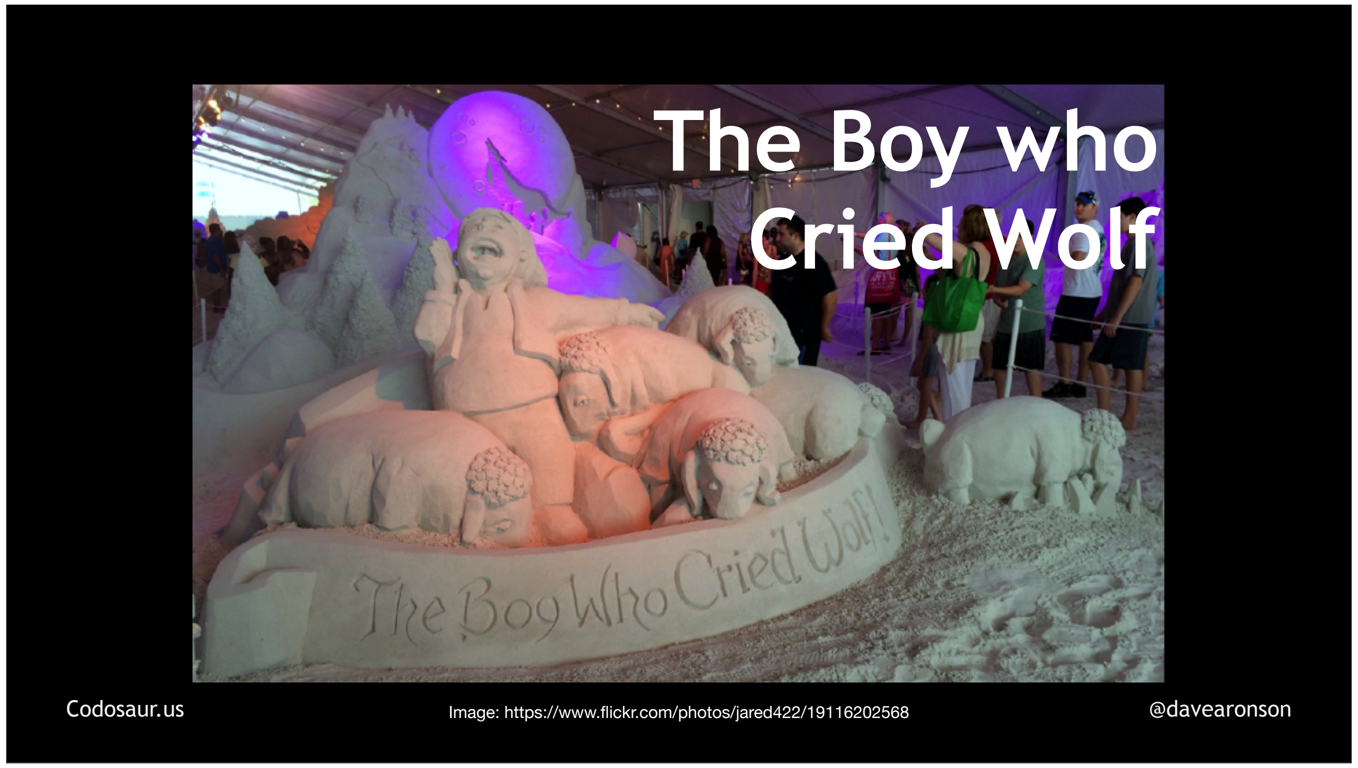
. . . not at all a beginner-friendly technique! It tells us that some particular change to the code made no difference to the test results, but what does *that* even mean? It takes a lot of interpretation to figure out what a mutant is trying to tell us. Their accent is verrah straynge, and they're almost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

Codosaur.us

Image: https://pixabay.com/vectors/zombie-halloween-dead-monster-521243/

@davearonson

. . . 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 . . .



The Boy who Cried Wolf

Codosaur.us Image: https://www.flickr.com/photos/jared422/19116202568 @davearonson

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

Even if a mutation *does* make a difference, there is normally quite a lot of code that we . . .

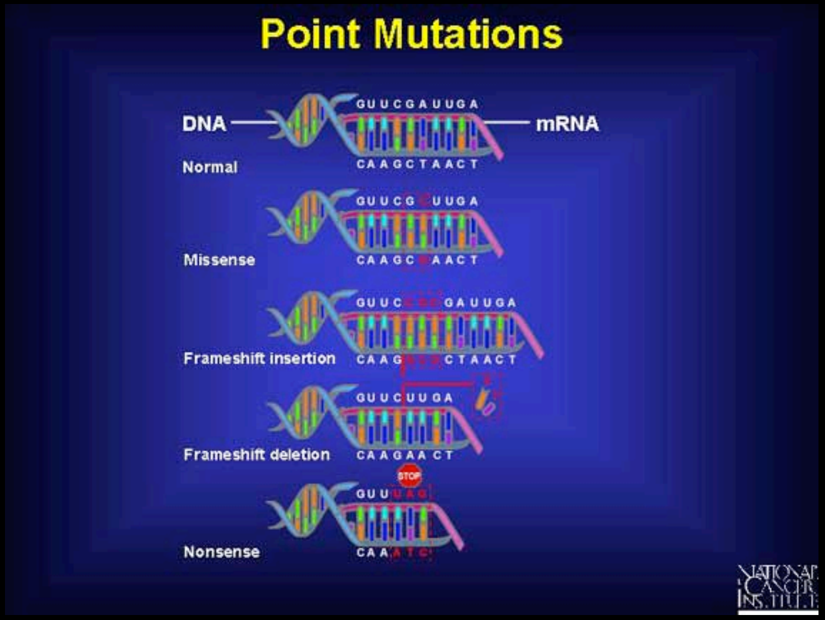
Codosaur.us

Image: https://pixabay.com/fi/vectors/työ-rekisteröidy-laiska-47200/

@davearonson

. . . *shouldn't bother* to test. For instance, if we have a debugging log message that says "The value of X is" and then the value of X, that constant part will get mutated, but we don't really care! Fortunately, most tools have ways to say "don't bother mutating this line", or even this whole function . . . 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 the guy in this silhouette? It . . .

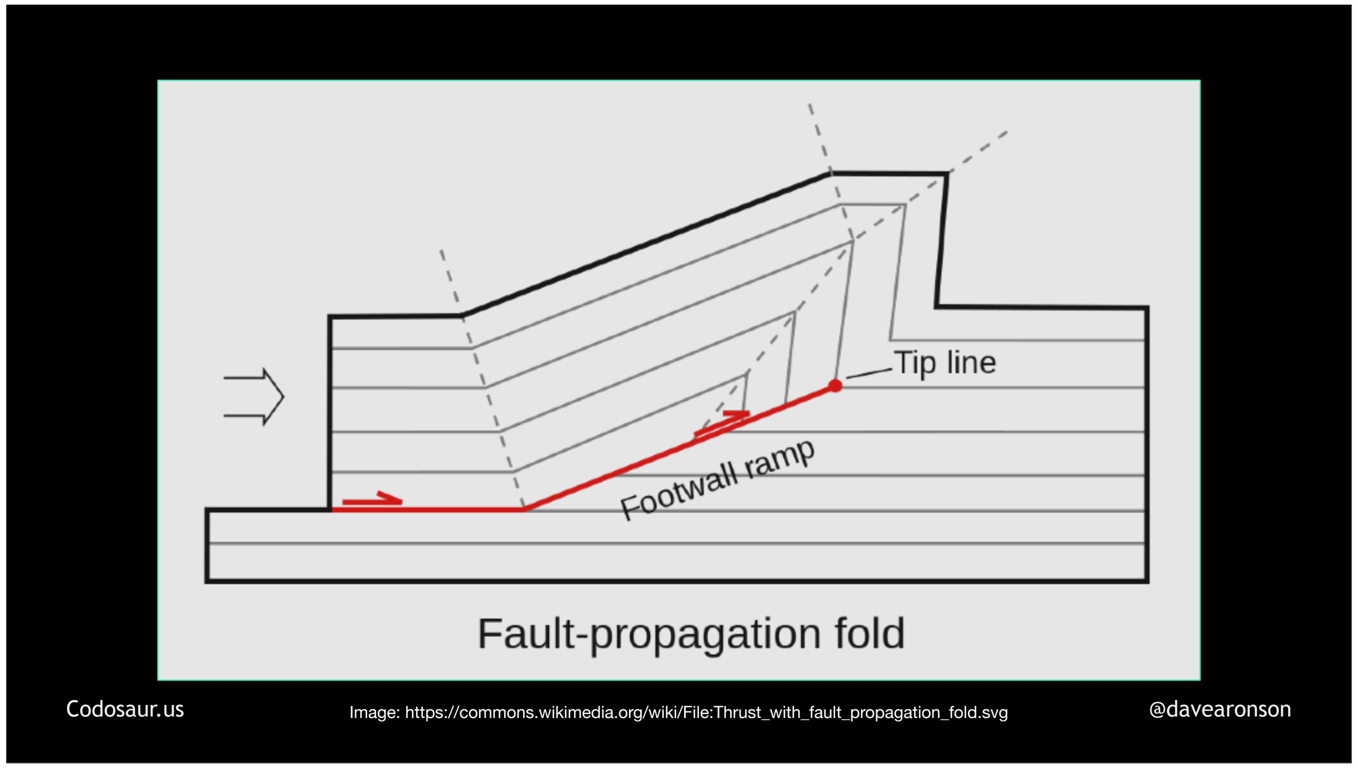


Codosaur.us

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 known as . . .



Codosaur.us

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

@davearonson

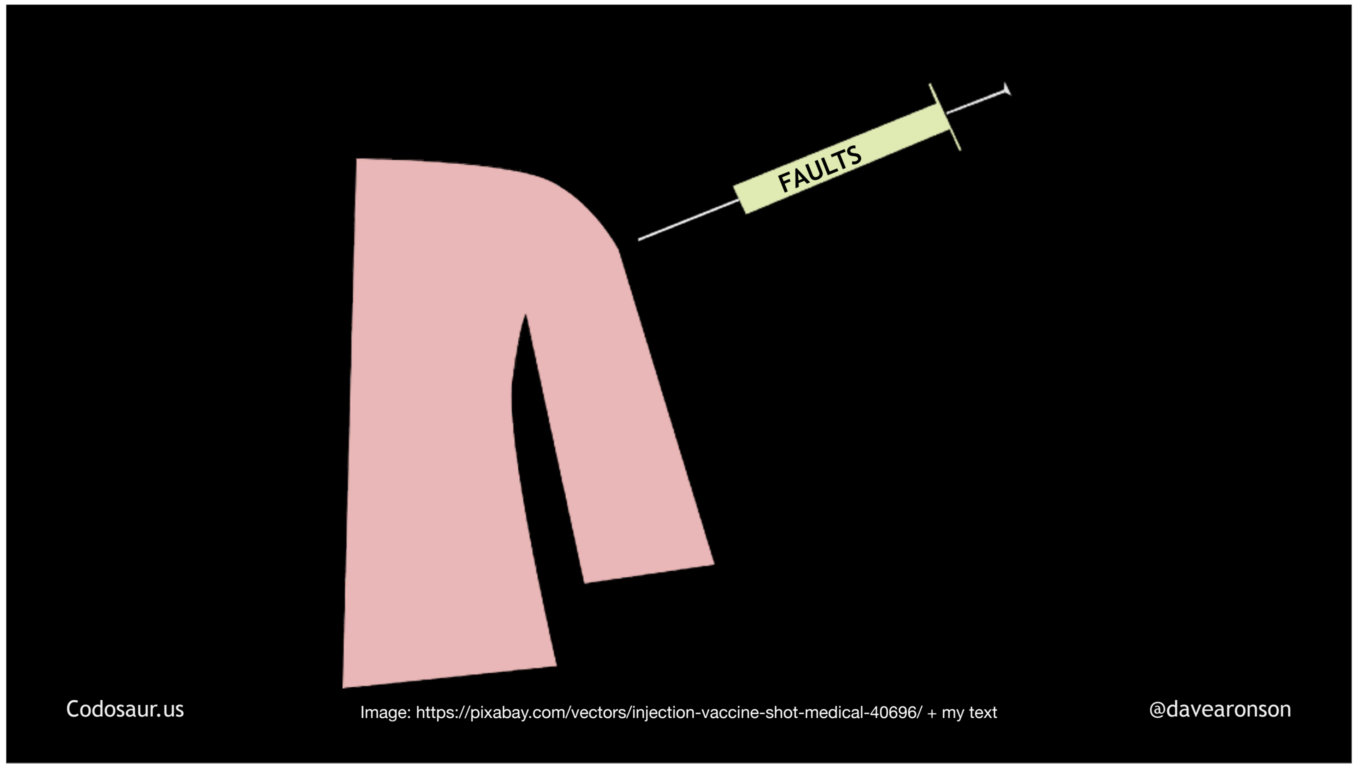
. . . faults. So, mutation testing can be categorized as a *fault-based* testing technique. This means it is related to something you might already be familiar with:

Image: https://github.com/Netflix/chaosmonkey/raw/master/docs/logo.png
(used for educational Fair Use purposes)

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

Image: https://github.com/Netflix/chaosmonkey/raw/master/docs/logo.png
(used for educational Fair Use purposes)

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



Codosaur.us

Image: https://pixabay.com/vectors/injection-vaccine-shot-medical-40696/ + my text

@davearonson

. . . injecting faults, such as dropped connections, into Netflix's . . .

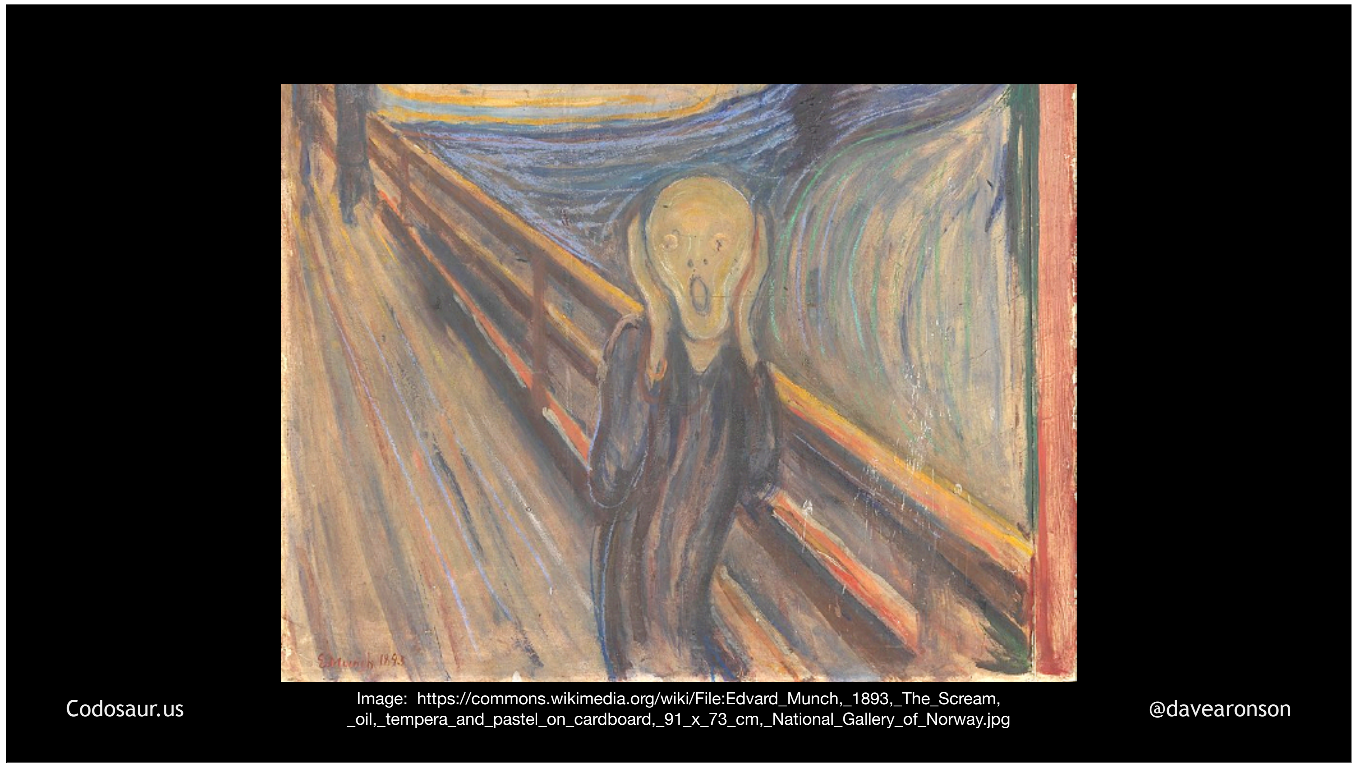

Codosaur.us

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

. . . production network.

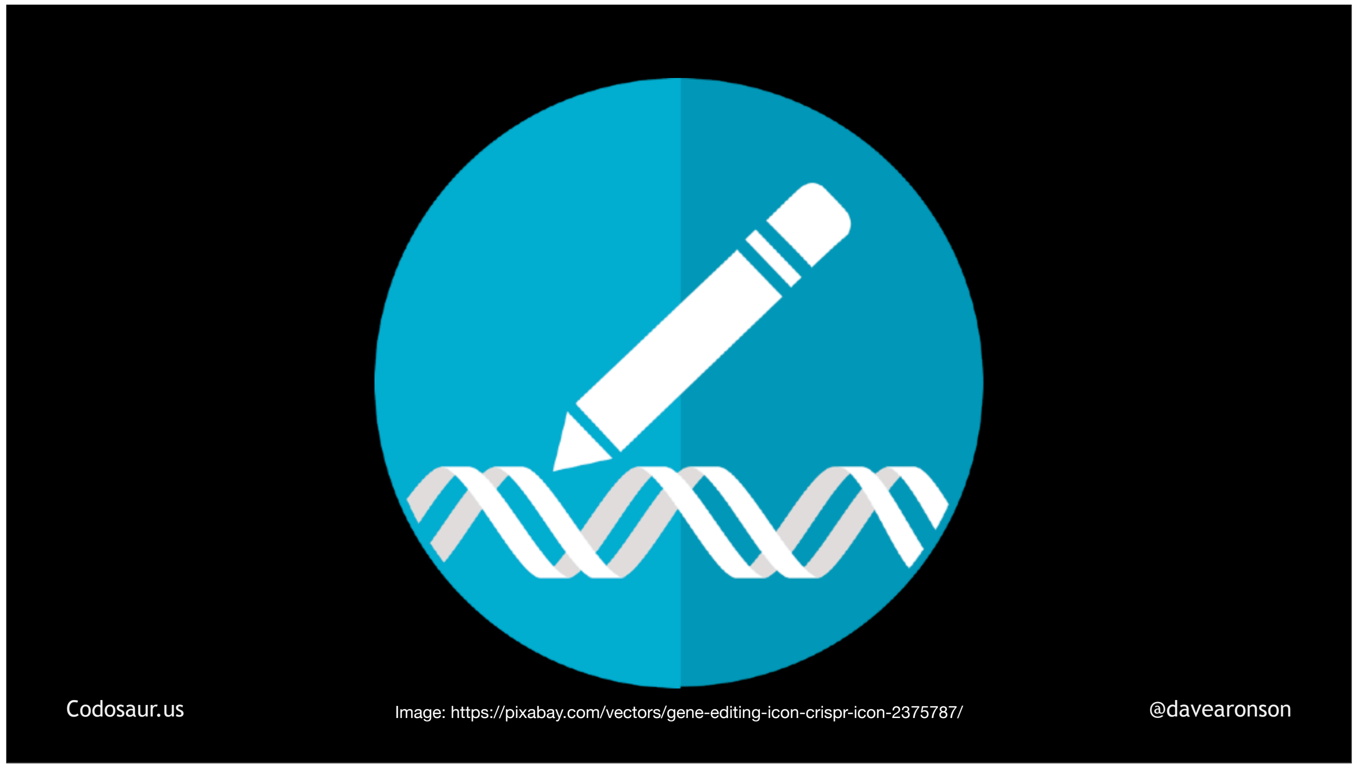
If all still goes well, in the sense that Netflix's customers don't notice, and their metrics are still good, then Netflix knows that their error recovery is working fine. Mutation testing, however, injects semantic . . .

Codosaur.us

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

@davearonson

. . . *changes*, not necessarily *problems*.   It doesn't usually *know* whether these semantic changes will create *faults* or not. We certainly hope they all will, but that depends on the test suite. It injects them *into* . . .


Codosaur.us
Image: https://pixabay.com/vectors/gene-editing-icon-crispr-icon-2375787/
@davearonson

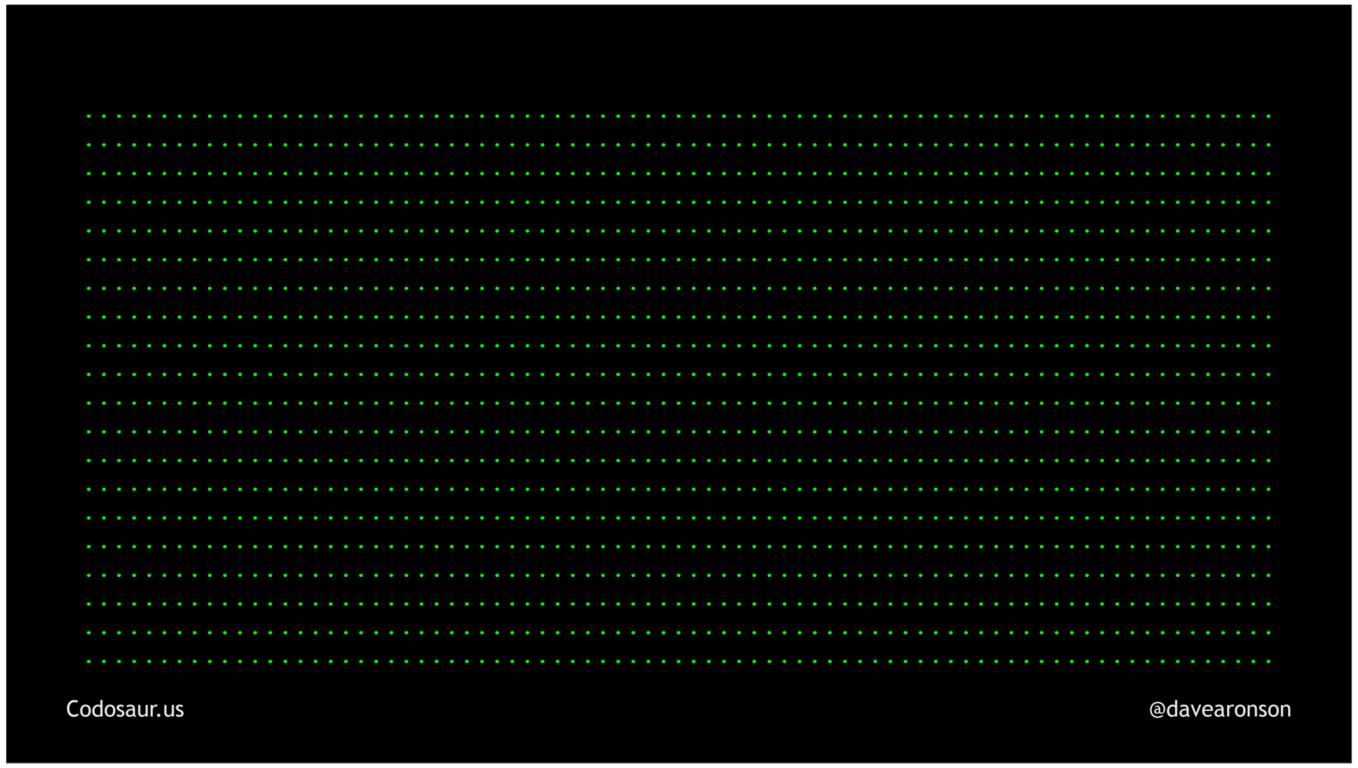
. . . copies of our code, not our actual network. It does its work in our . . .

Codosaur.us

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

@davearonson

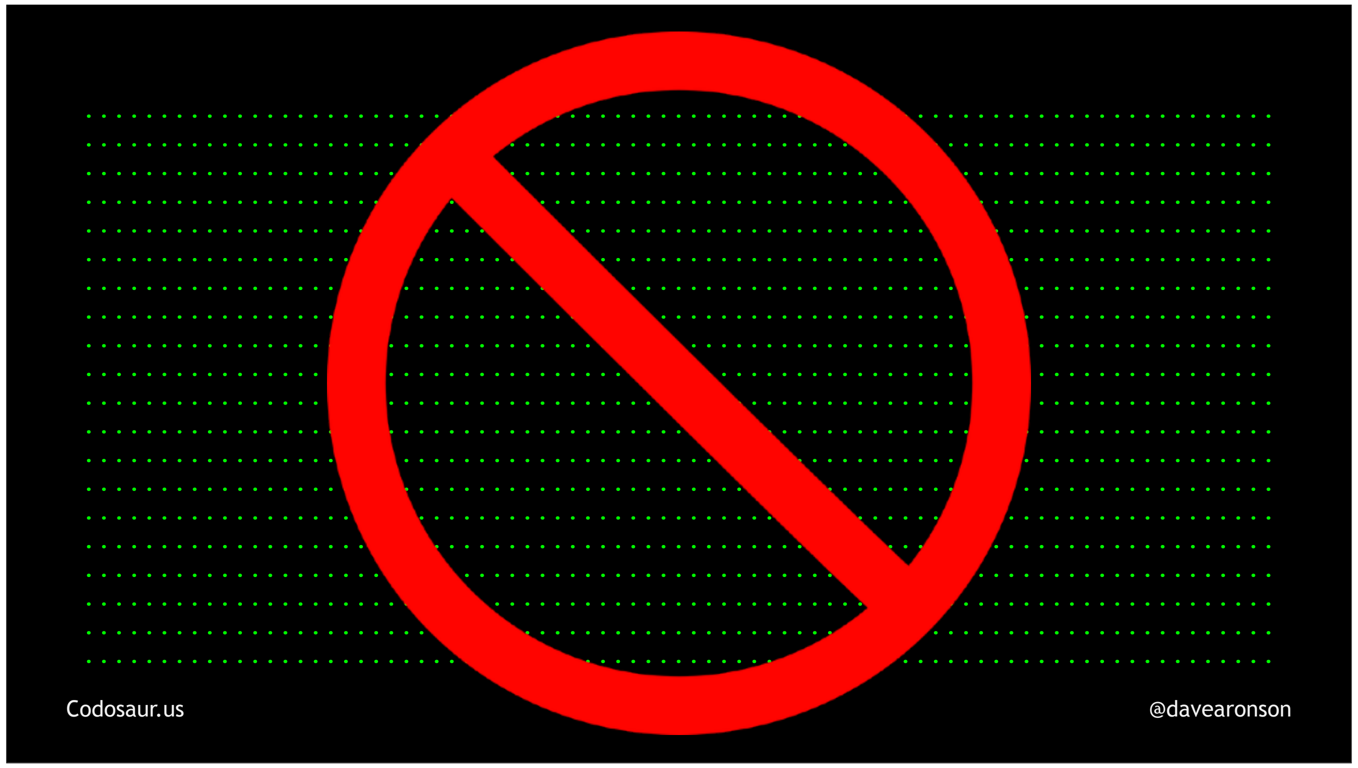
. . . *test* environment, not production. (Whew!) And if everything still goes well, *in the sense that* . . .



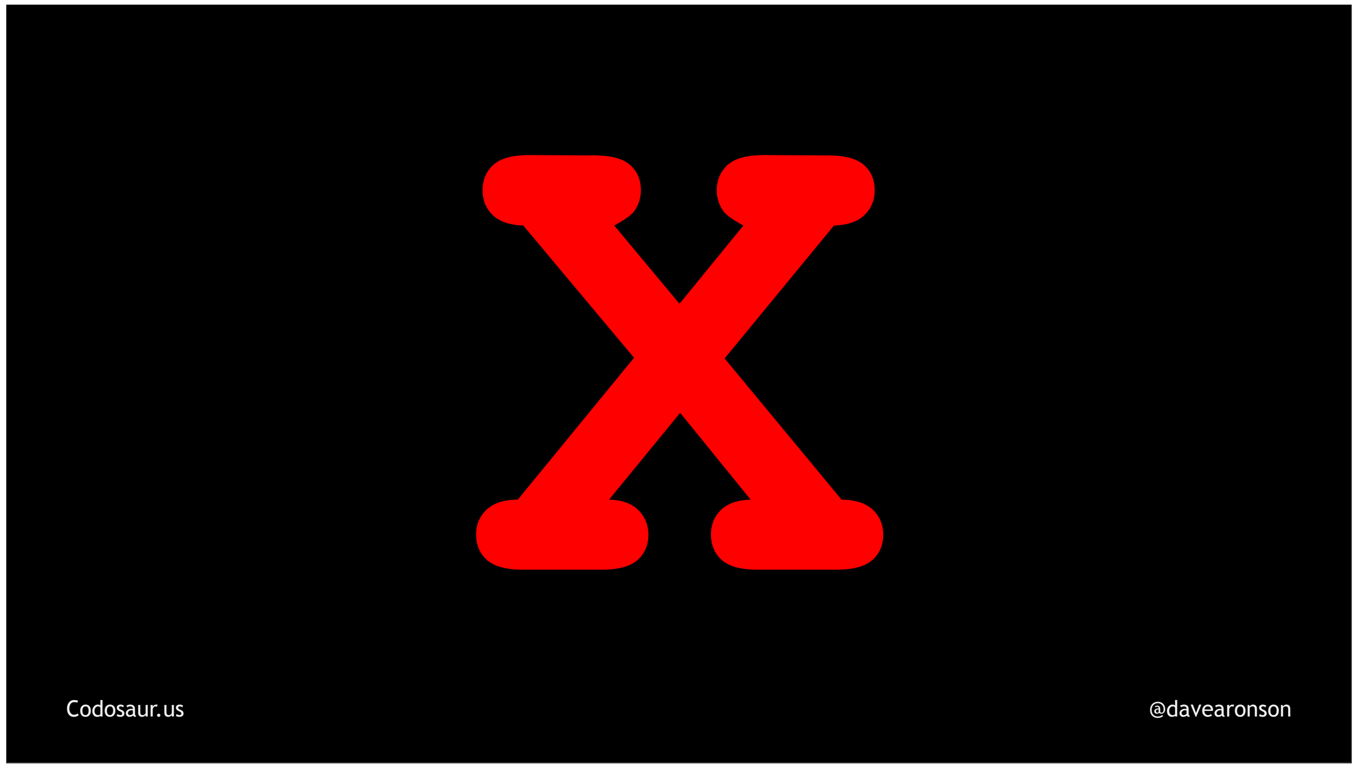
```
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
................................................................................
```

Codosaur.us @davearonson

. . . our units 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 . . .

Codosaur.us

Image: https://www.wannapik.com/vectors/15576 plus gradient-filled rounded rectangle

@davearonson

. . . fuzzing, short for fuzz testing, 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 generally . . .

Codosaur.us
Image: https://www.wannapik.com/vectors/15576 plus gradient-filled rounded rectangle
@davearonson

. . . not random. Most mutation testing engines apply all the mutations that they know how to do. The smarter ones can use the results from some simpler mutations to know they don't need to bother with more complex mutations, but still, it's not random.

But enough about differences. What exactly does mutation testing *do*, and how? Let's start with a high-level view. First, *ideally*, 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 an object-oriented language, but I'm just going to say functions. Some tools *don't* do this, but operate on a file-by-file basis, or maybe even the whole program. Both of these are less efficient for reasons we'll see in a moment. From here on, I'll assume that our tool is operating on a function-by-function basis. So, then, for each function, it tries to find . . .

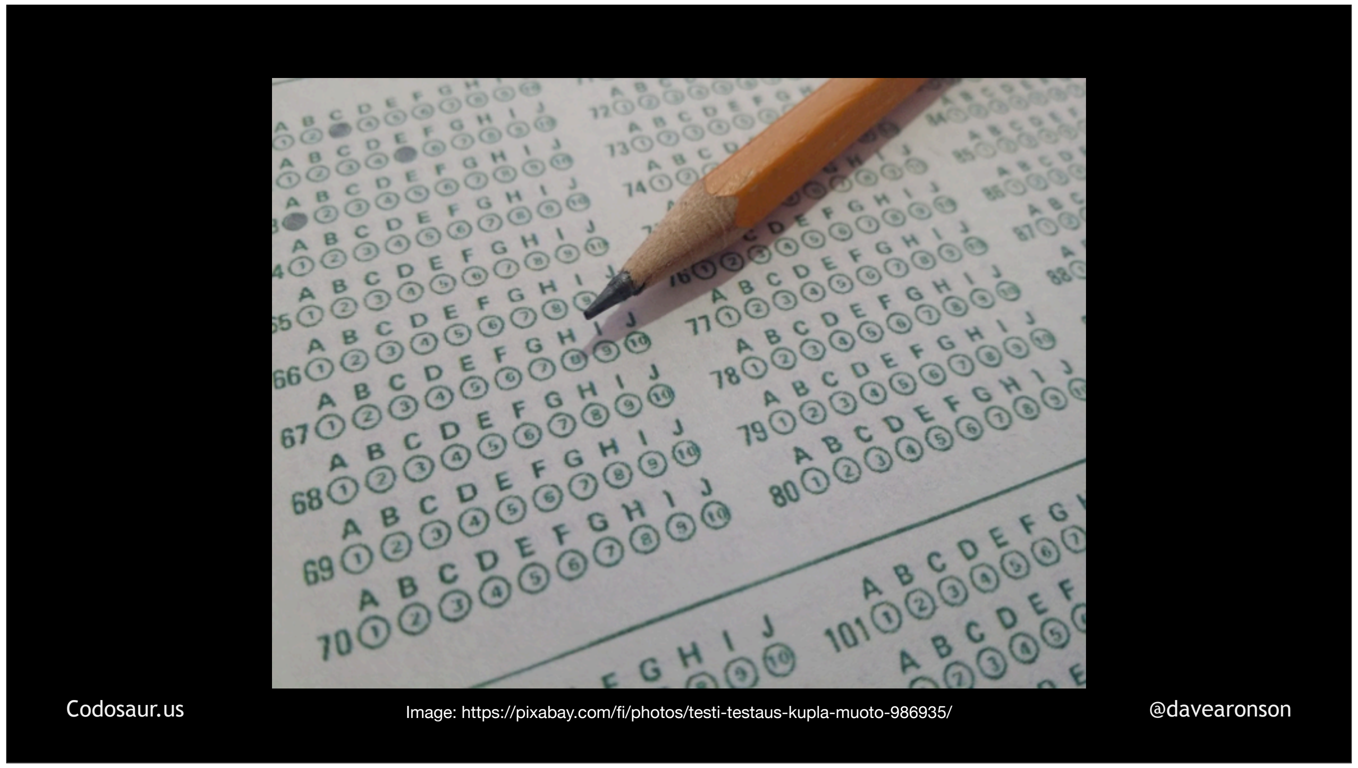

Codosaur.us

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

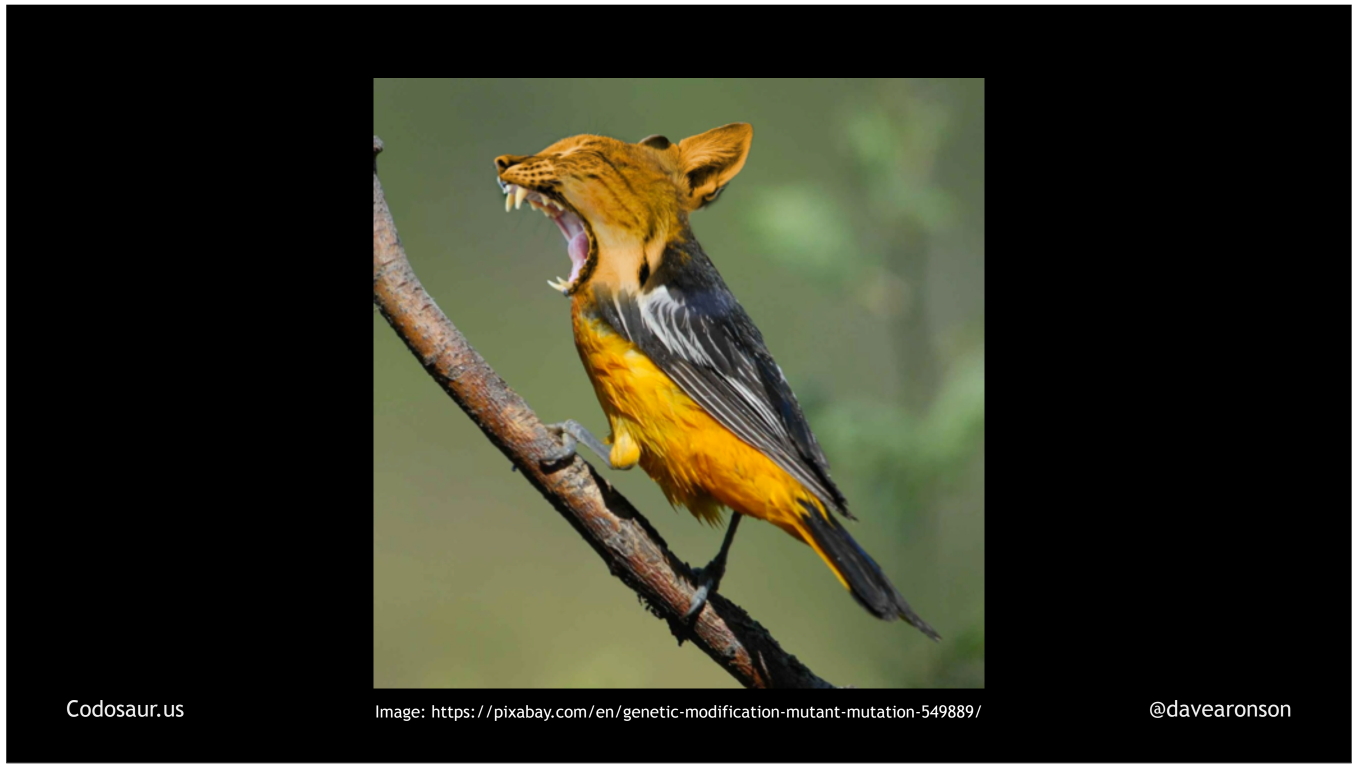
@davearonson

. . . the *tests* that cover that function. If the tool can't find any applicable tests, most will simply skip this function. Better yet, most of those will warn us, so we know we should go add or annotate some tests. (More on that later.) Some, though, will use the whole test suite, which is horribly inefficient, because it's running a lot more tests than it needs to.

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

Codosaur.us
Image: https://www.deviantart.com/polaris-xforce/art/The-Brotherhood-of-Evil-Mutants-390550995 (used by permission)
@davearonson

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



Codosaur.us

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

@davearonson

. . . one mutant, with *that one tiny little change*.

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

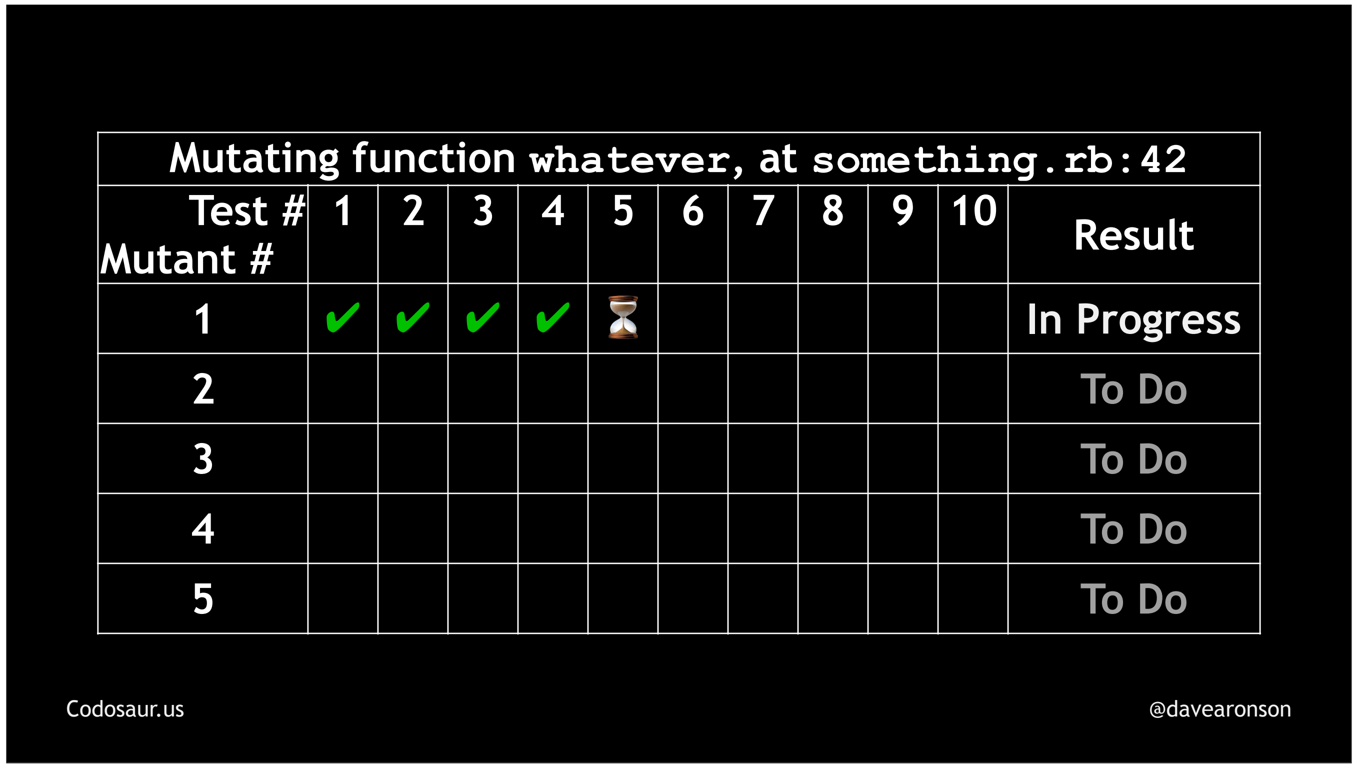
Ages 4 and up.
TEENAGE MUTANT NINJA TURTLES
"We're looking for a few good mutants!"
"Hey, kids, wanna take over the world?"

Codosaur.us

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

@davearonson

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



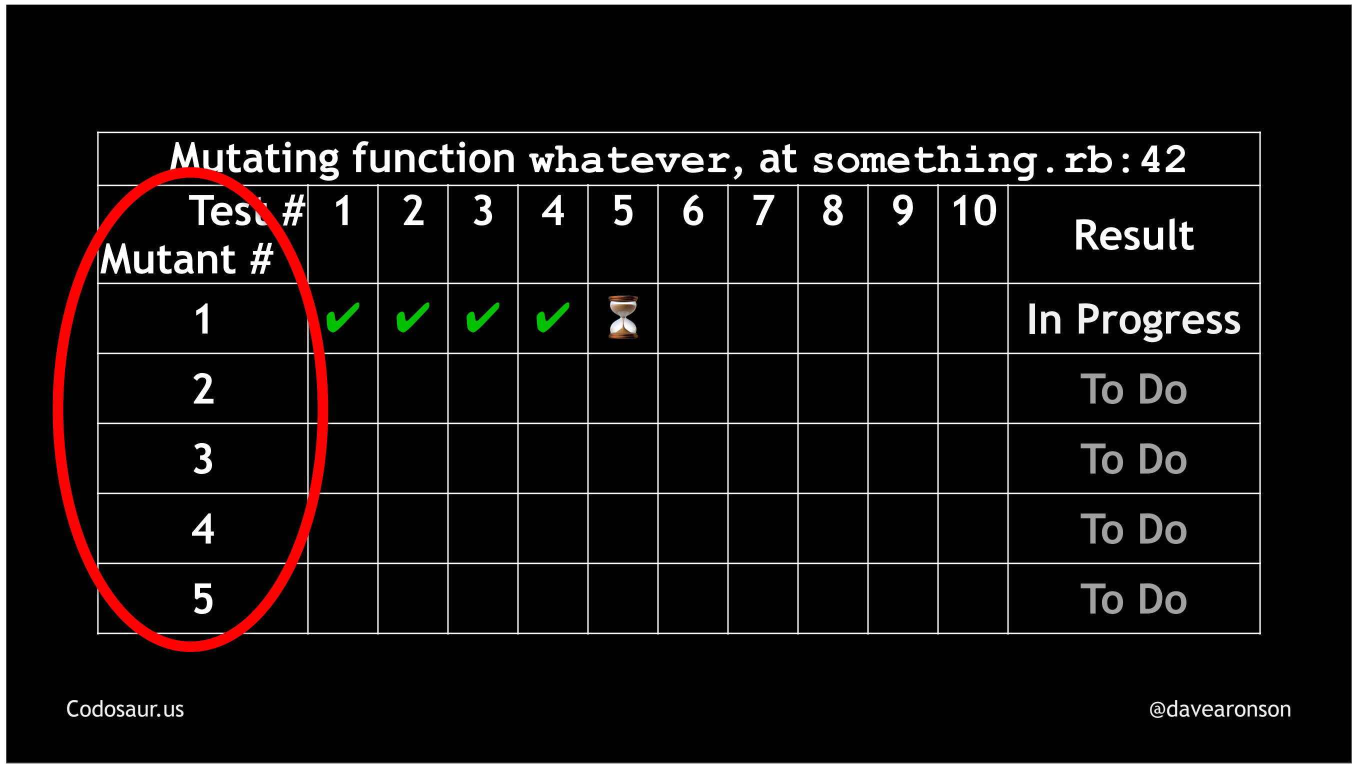
Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

This chart represent the progress of our tool. The tools generally don't give us any such sort of thing, but it's a conceptual model I'm using to help illustrate the point.

For each . . .

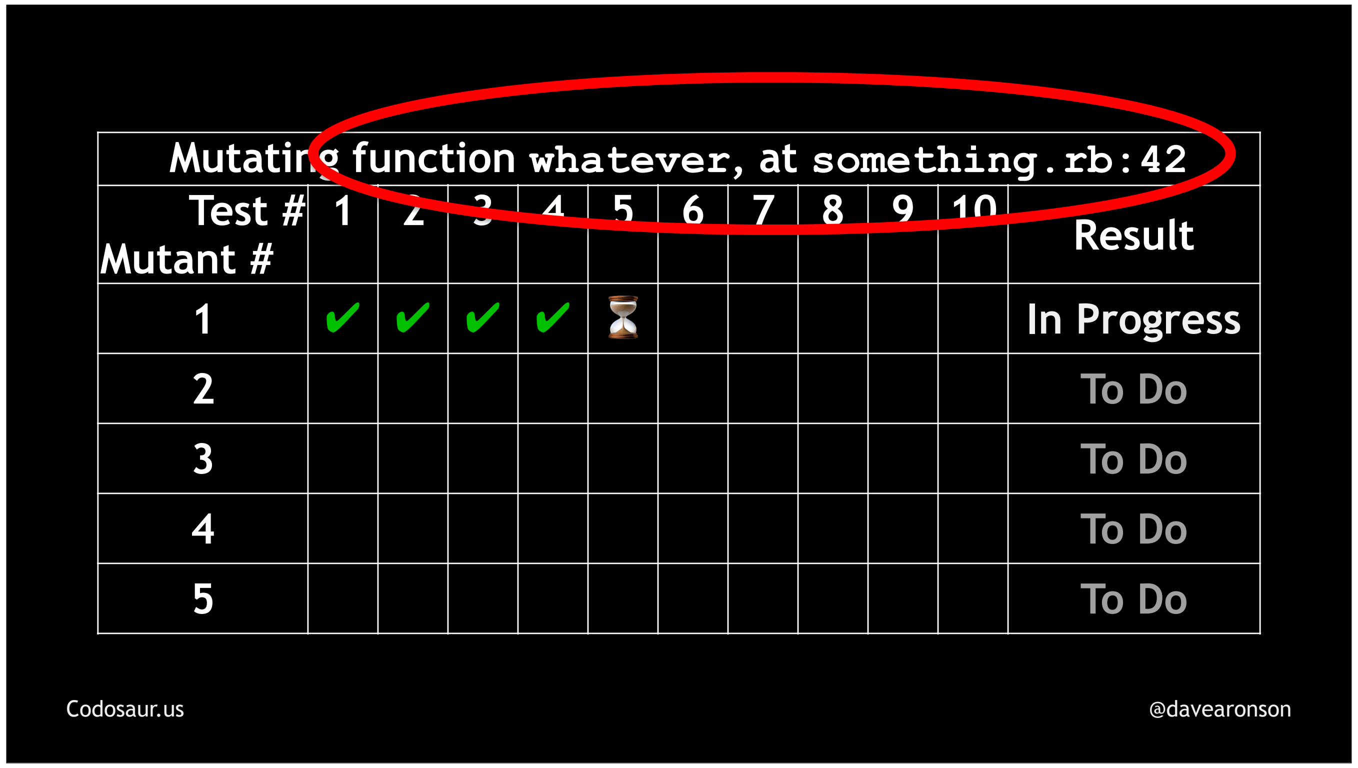


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . mutant, derived from . . .

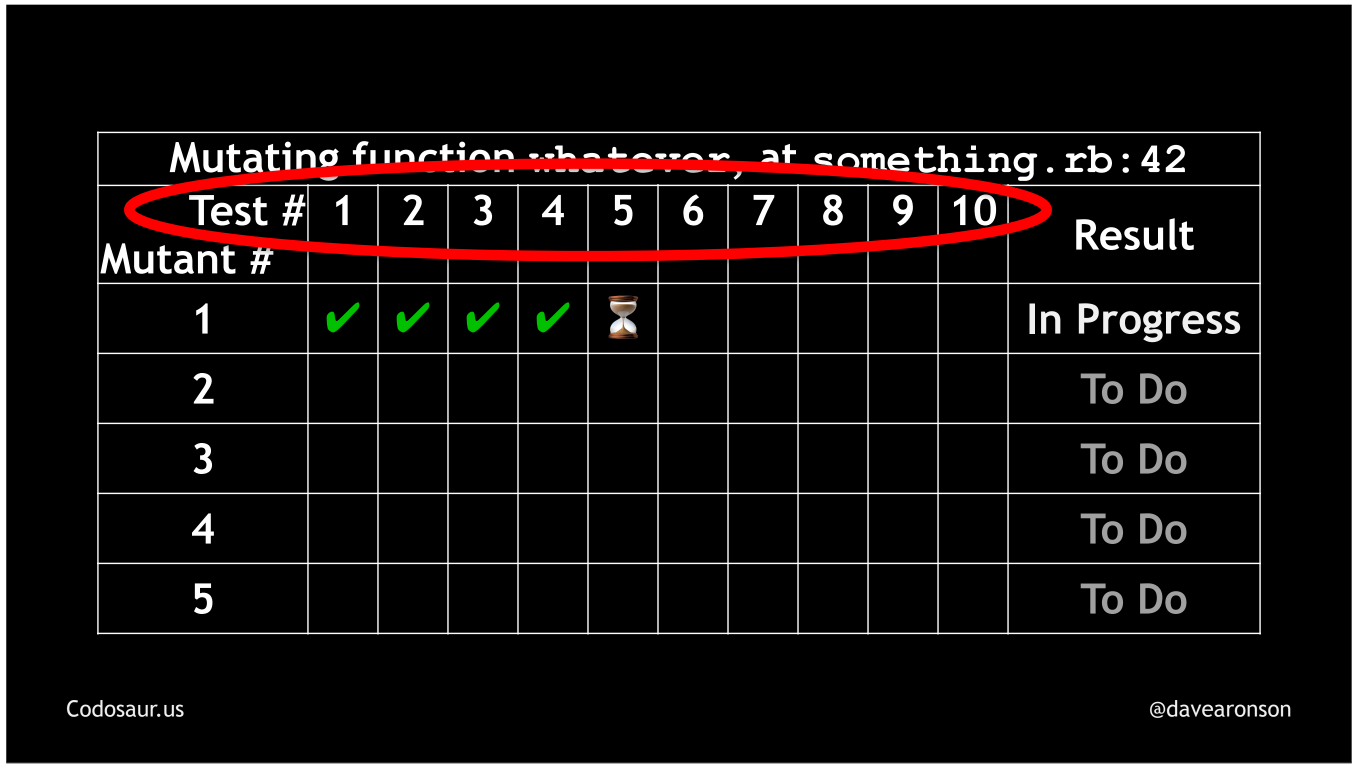


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

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

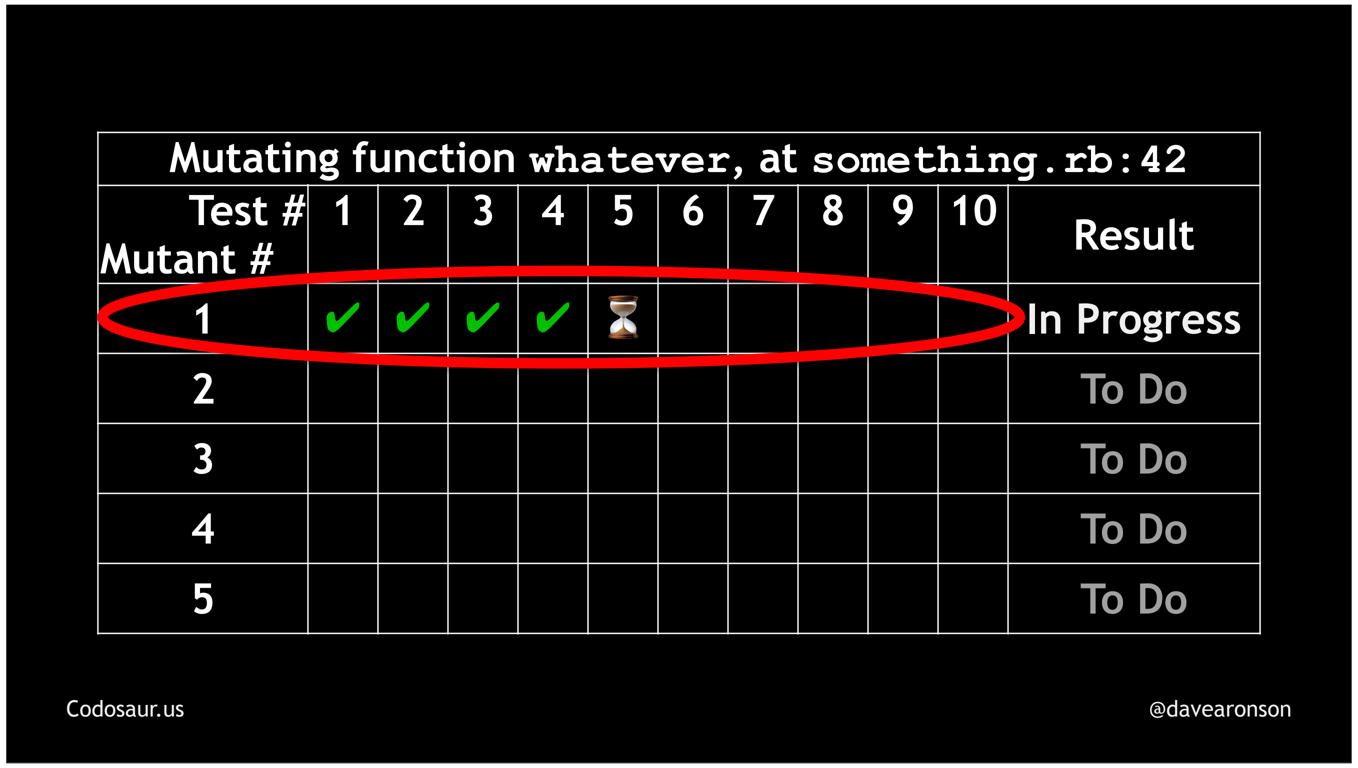


Mutating function `whatever,` at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

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



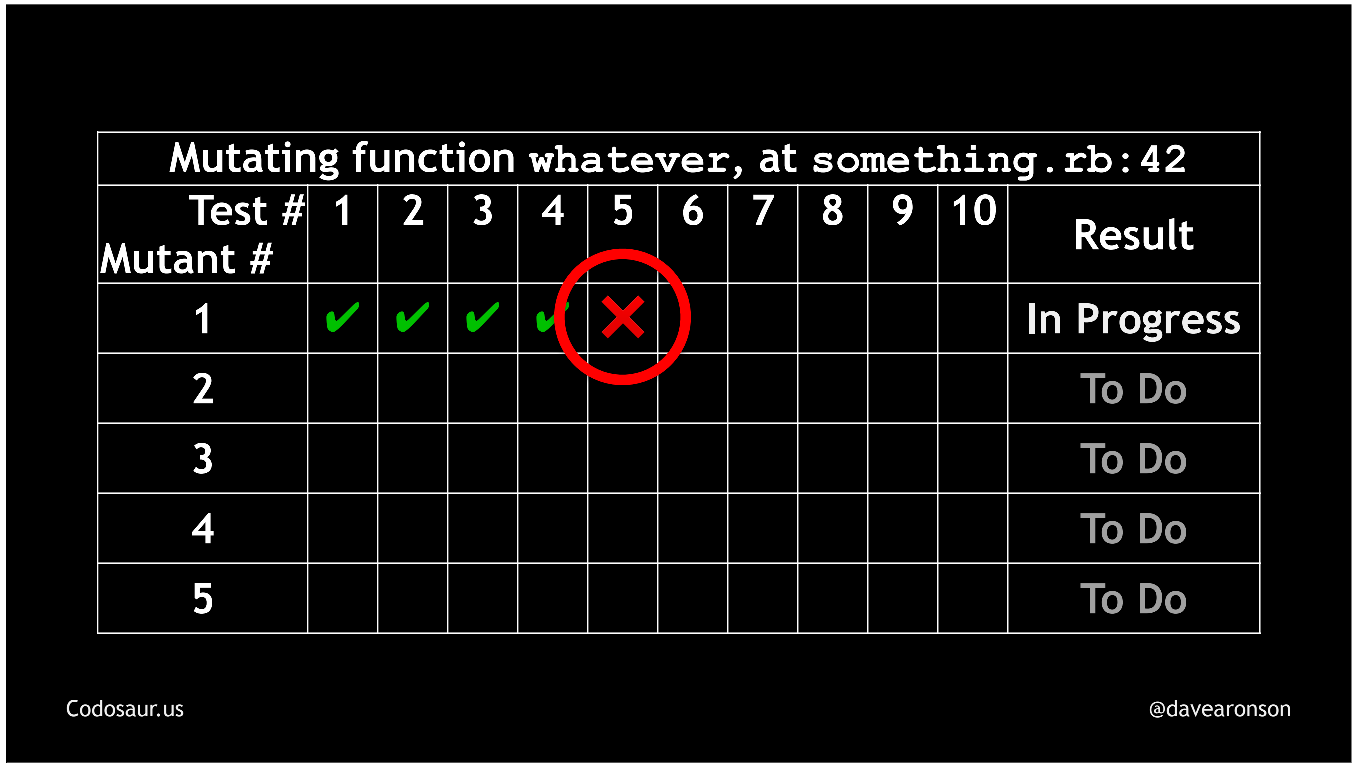
Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	⏳						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

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

(PAUSE) If any test . . .

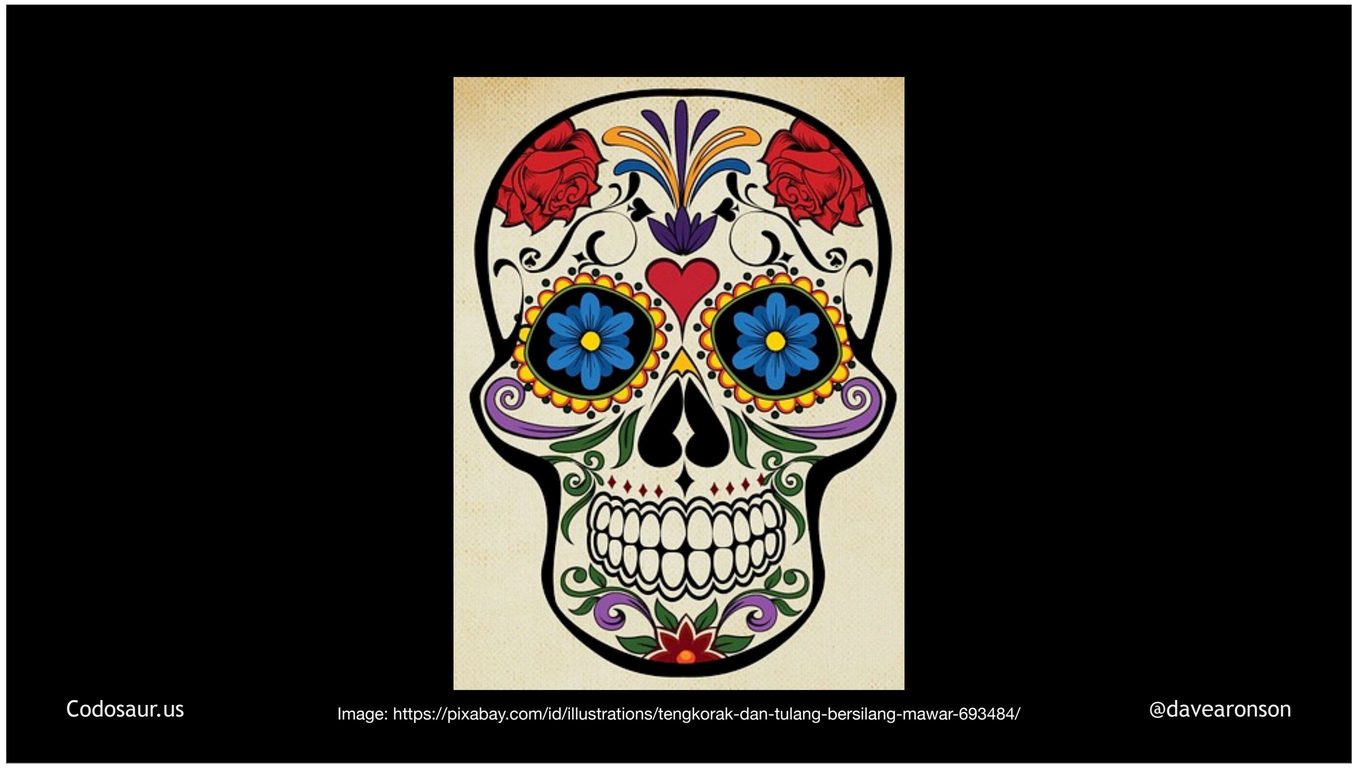


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . *fails*, this is called, in the standard industry terminology, by the unfortunate name of . . .



Codosaur.us

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

@davearonson

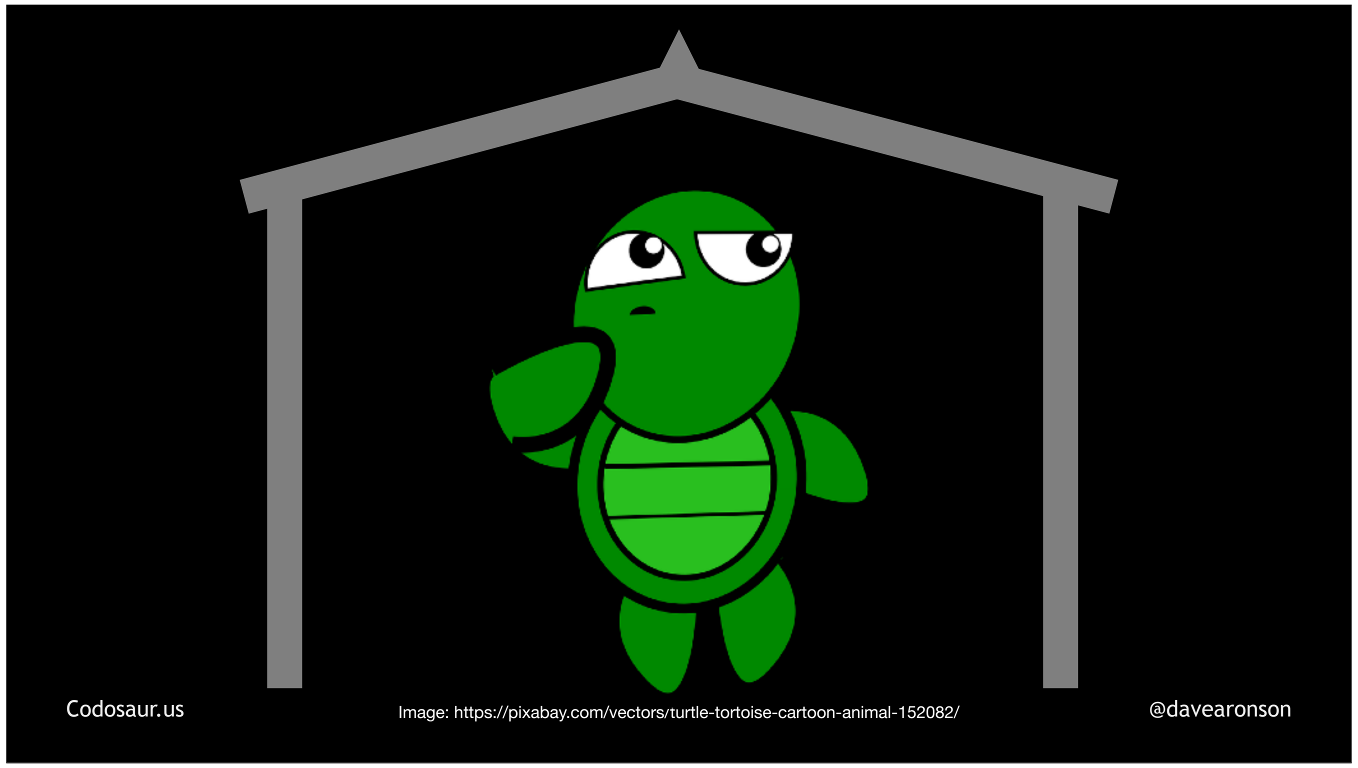
. . . “killing the mutant”. Here, I’m going to take a bit of a detour, because some people . . .

Codosaur.us

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

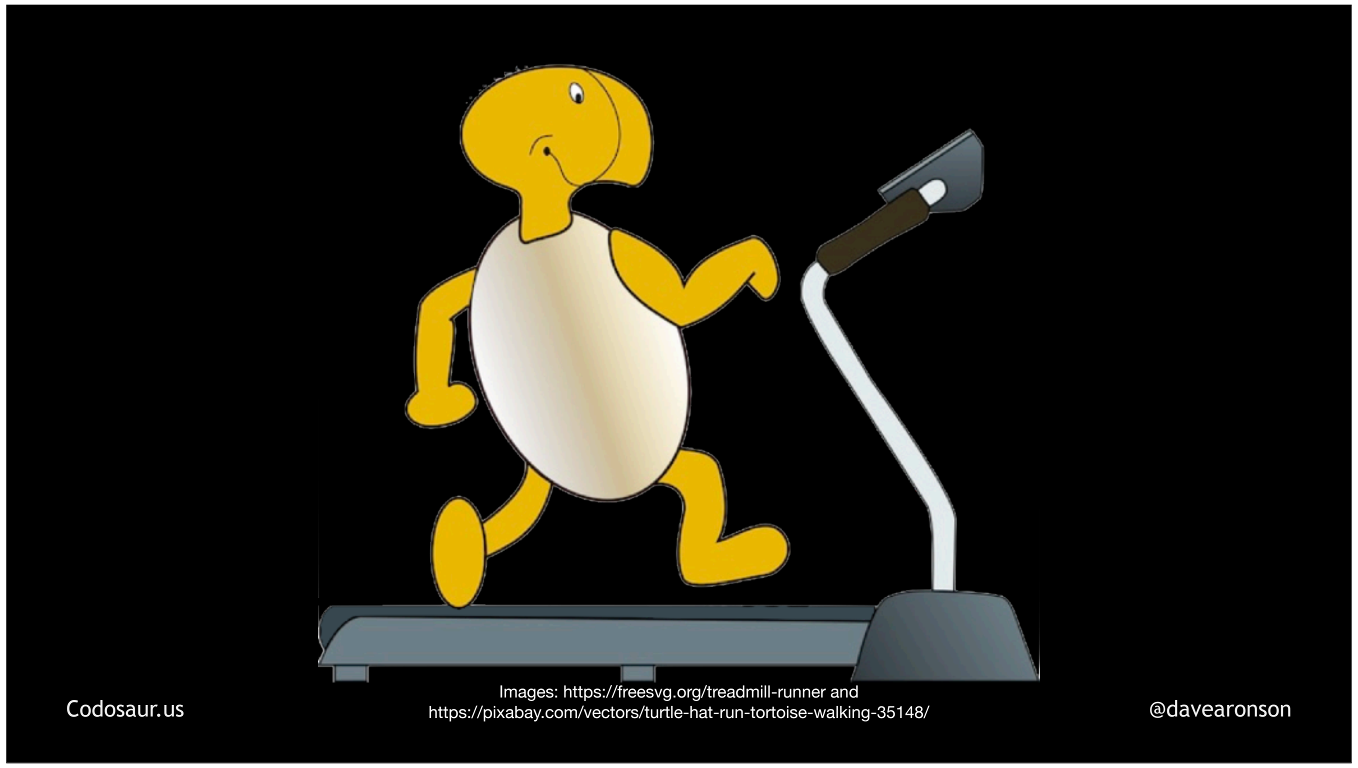
@davearonson

. . . object to this “violent communication”, especially since, in the comic books, mutants are often metaphors for marginalized groups of people, and the tech industry is *finally* starting to become more sensitive to such issues. So, I’m trying to come up with some nicer terminology. So far, I'm leaning towards terms like *rescuing*, or better yet . . .


Codosaur.us
Image: https://pixabay.com/vectors/turtle-tortoise-cartoon-animal-152082/
@davearonson

. . . *"covering"* the mutant. This makes sense if you think about it in terms of what really happens in mutation testing. In normal use of test coverage, *normal* code should be "covered" by at least one test, and let all tests pass. By way of analogy, in mutation testing, *mutants* should be "covered" by at least one *failing* test. Remember, each change should make at least one test fail.

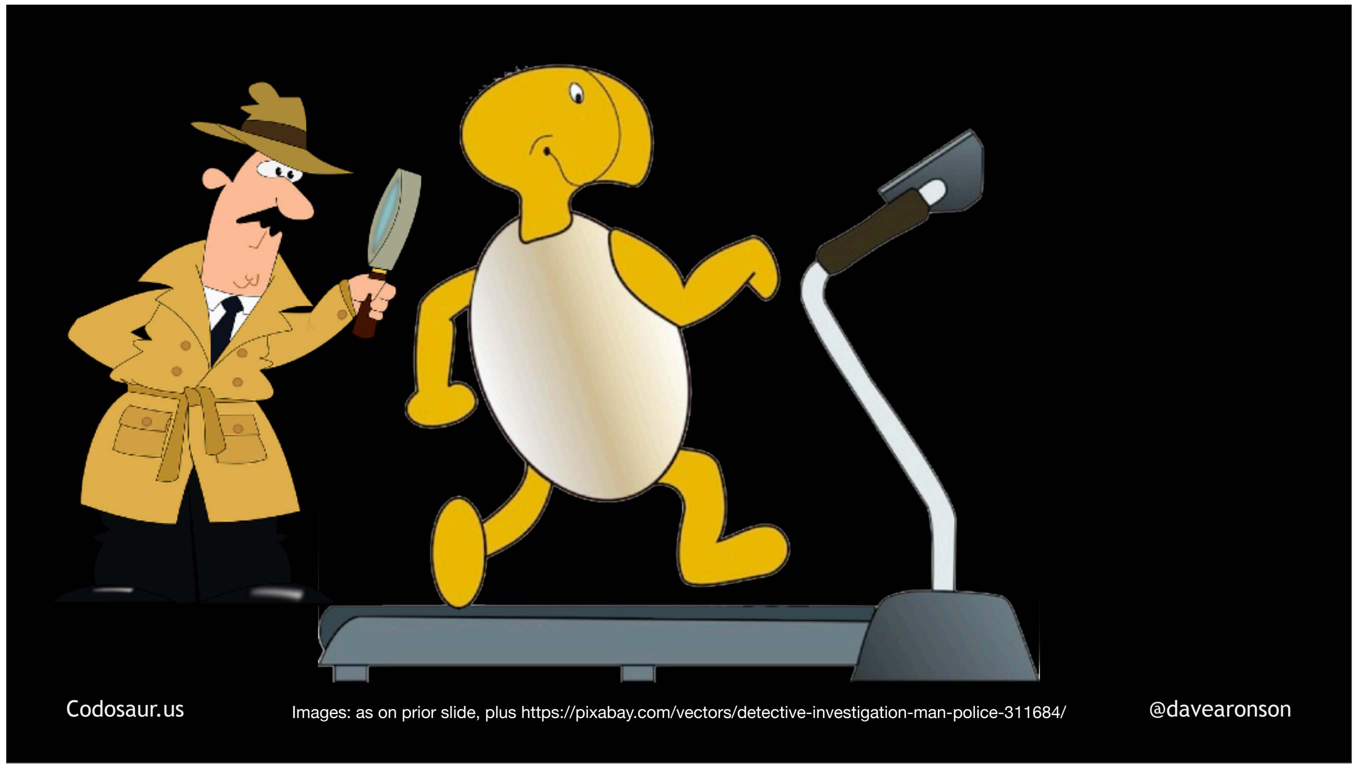
Unfortunately, the term "covered" is already used, to mean that the mutated code is *run* by at least one test, whether failing or not, much like the normal concept of test coverage, that we discussed earlier. So, I'd like to replace *that* with saying that the mutant is . . .


Codosaur.us

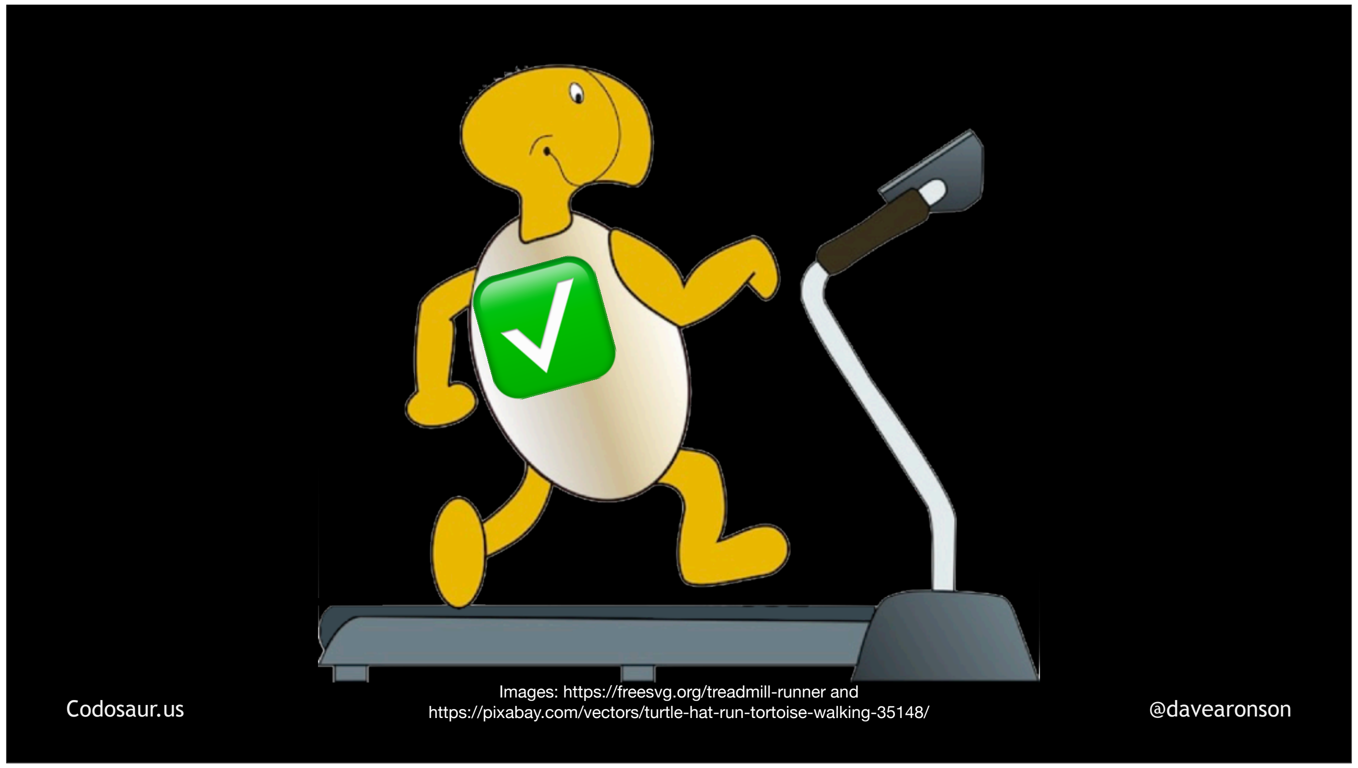
Images: https://freesvg.org/treadmill-runner and
https://pixabay.com/vectors/turtle-hat-run-tortoise-walking-35148/

@davearonson

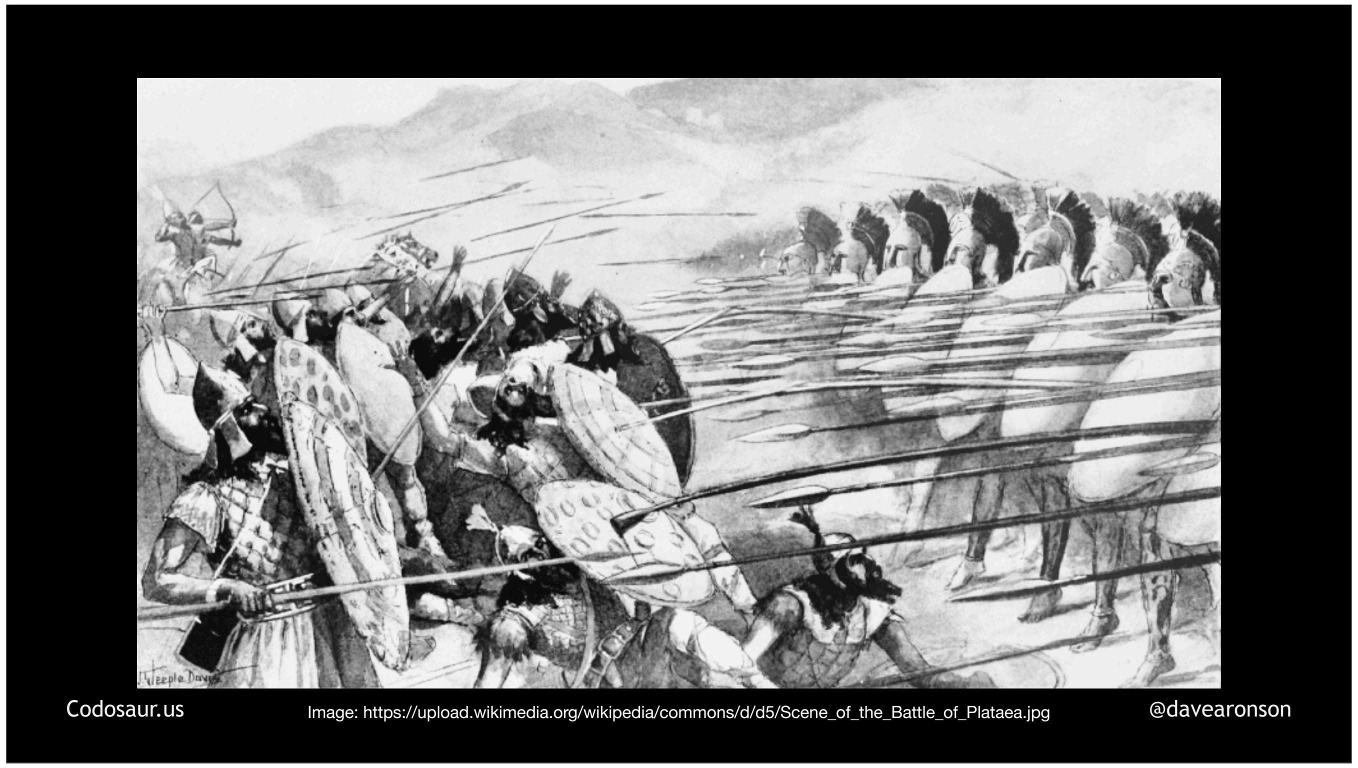
. . . *exercised*, or possibly . . .


Codosaur.us
Images: as on prior slide, plus https://pixabay.com/vectors/detective-investigation-man-police-311684/
@davearonson

. . . *inspected*, or . . .

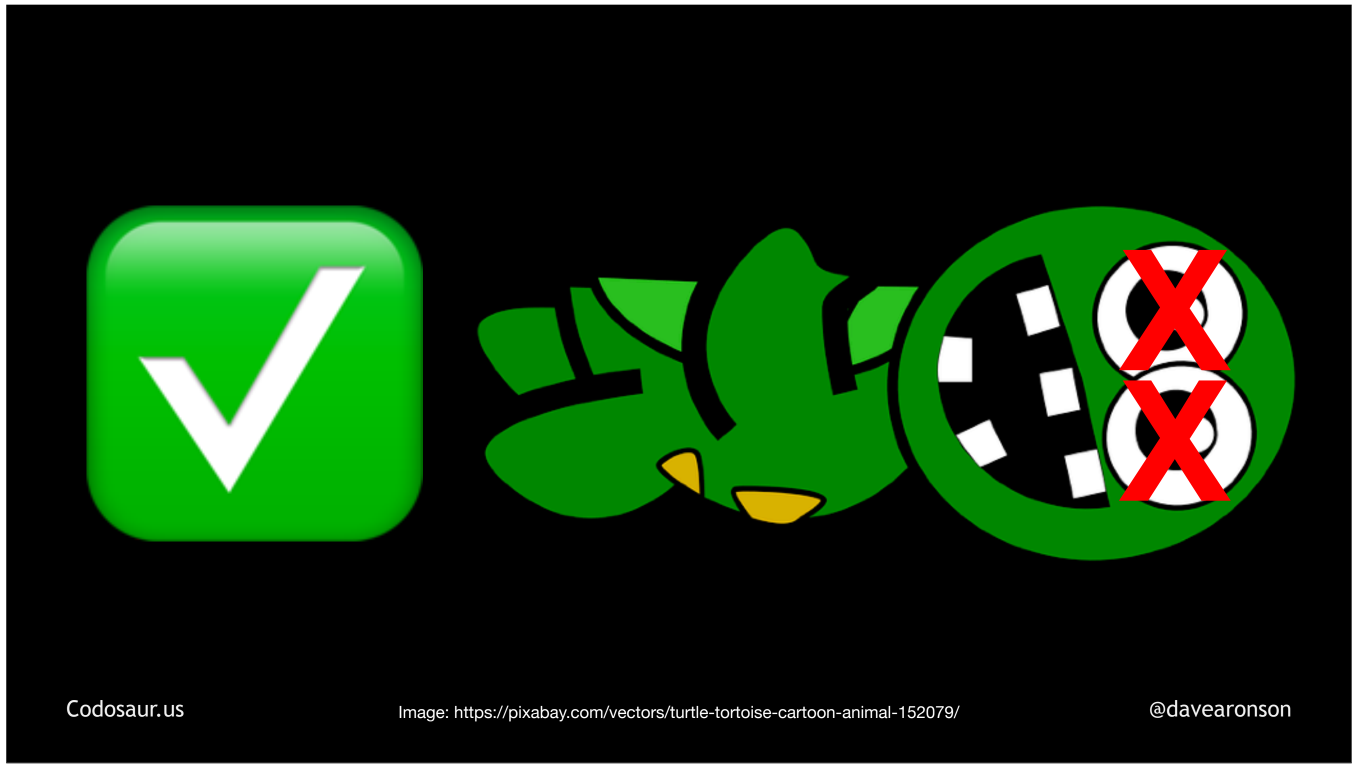

Codosaur.us
Images: https://freesvg.org/treadmill-runner and https://pixabay.com/vectors/turtle-hat-run-tortoise-walking-35148/
@davearonson

. . . *checked*. But, it's a long hard . . .

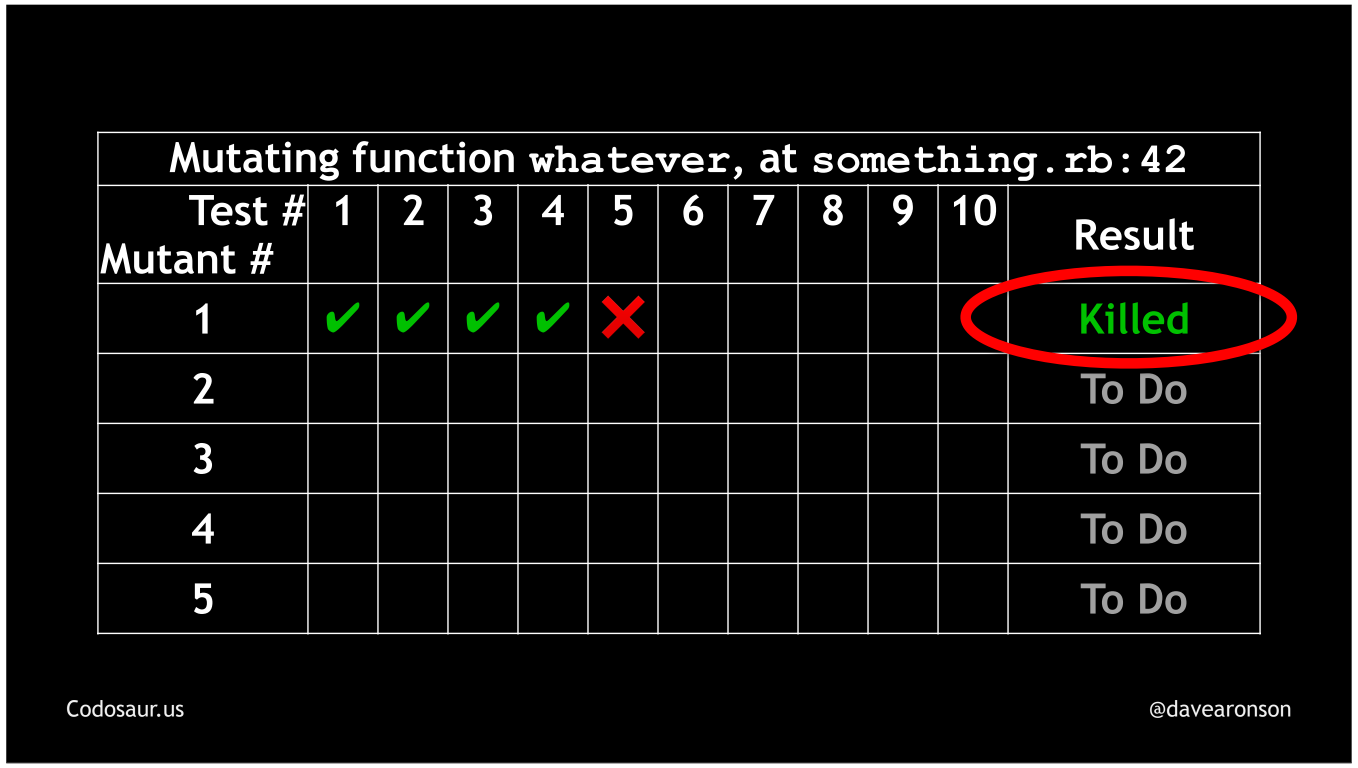

Codosaur.us
Image: https://upload.wikimedia.org/wikipedia/commons/d/d5/Scene_of_the_Battle_of_Plataea.jpg
@davearonson

. . . uphill battle, trying to change terminology that people are already using. Oh well.

Anyway, whatever we *call* it when a mutant makes a test fail, and I'm going to stick to the standard term of "killing" it in this presentation, each mutant *should* make at least *one* test fail. So when it *does*, that's a . . .


Codosaur.us
Image: https://pixabay.com/vectors/turtle-tortoise-cartoon-animal-152079/
@davearonson

. . . *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. It also means that our *test* suite is *strict* enough that at least one test actually *noticed* that difference, and failed. Then, the tool will . . .

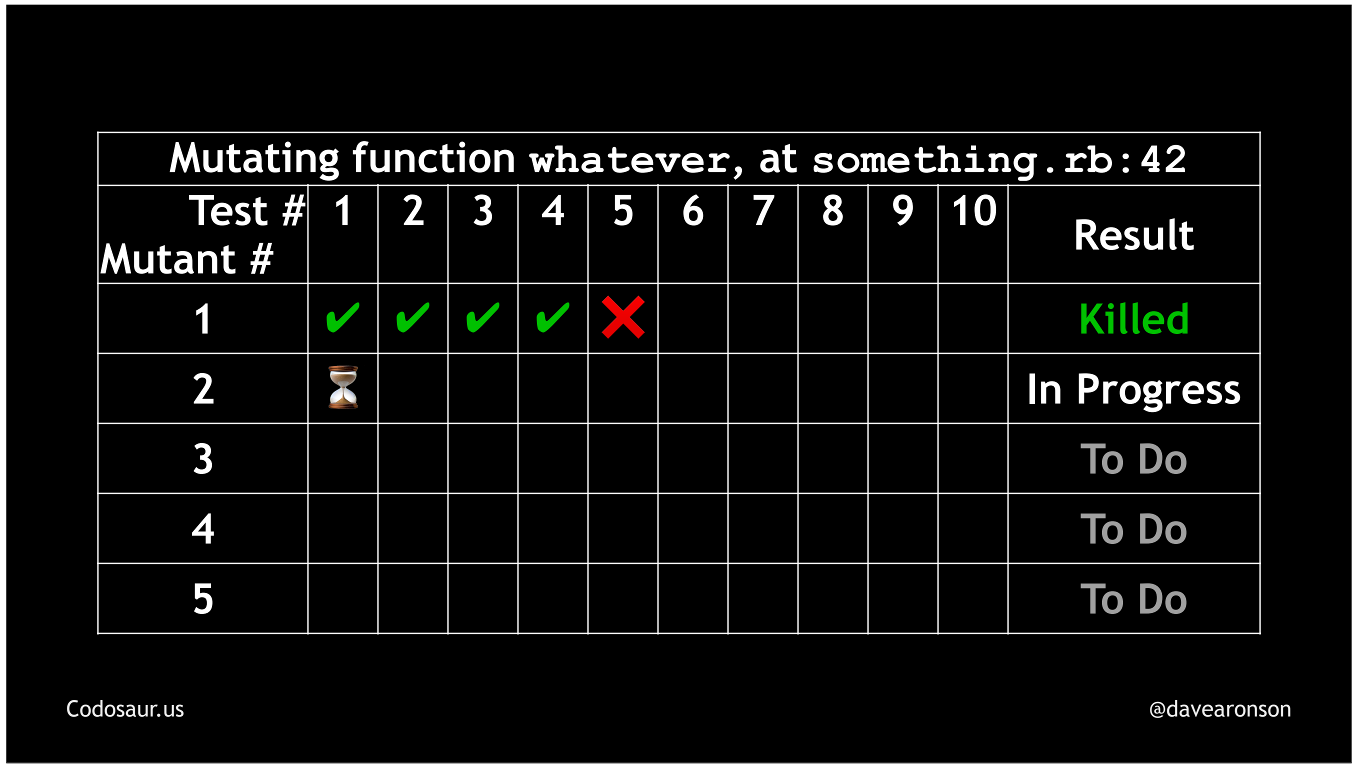


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗						Killed
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . mark that mutant killed, stop running any more tests against it, and . . .

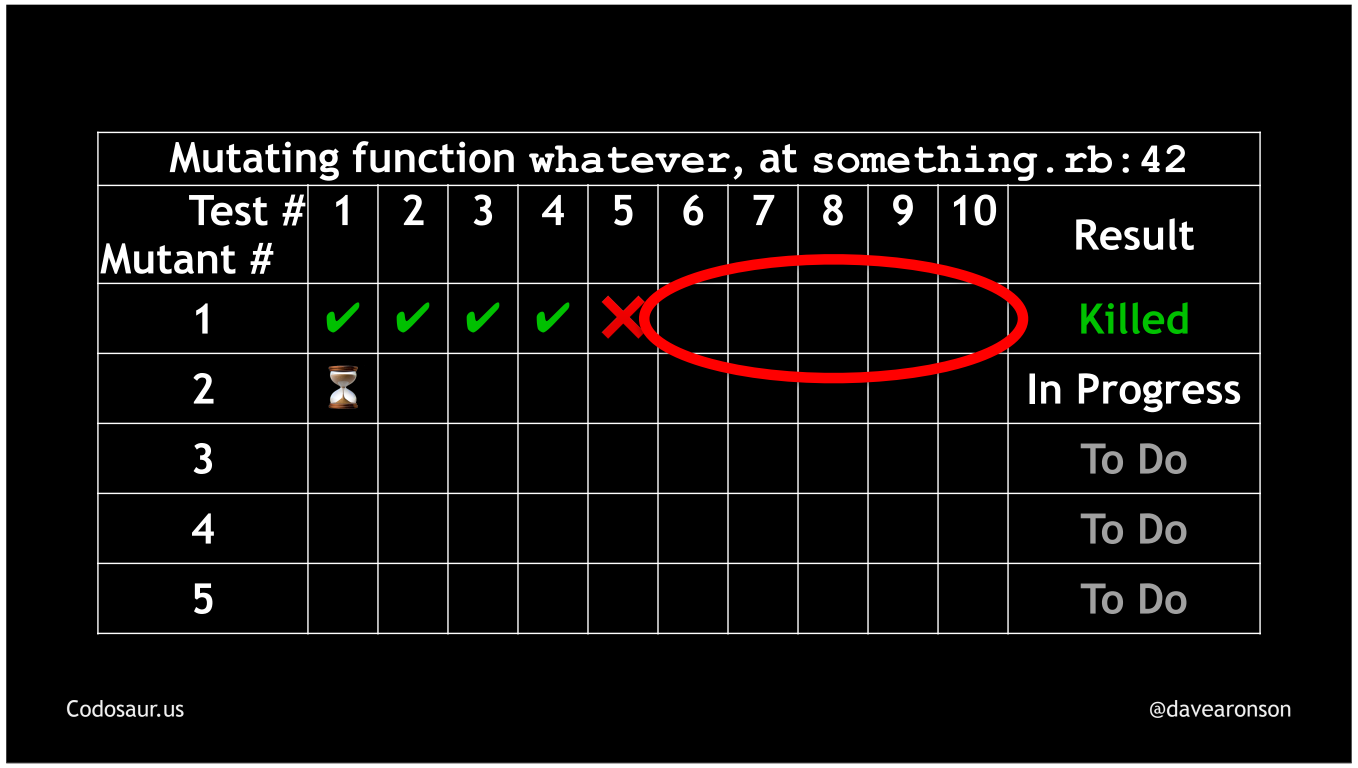


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗						Killed
2	⏳										In Progress
3											To Do
4											To Do
5											To Do

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 . . .



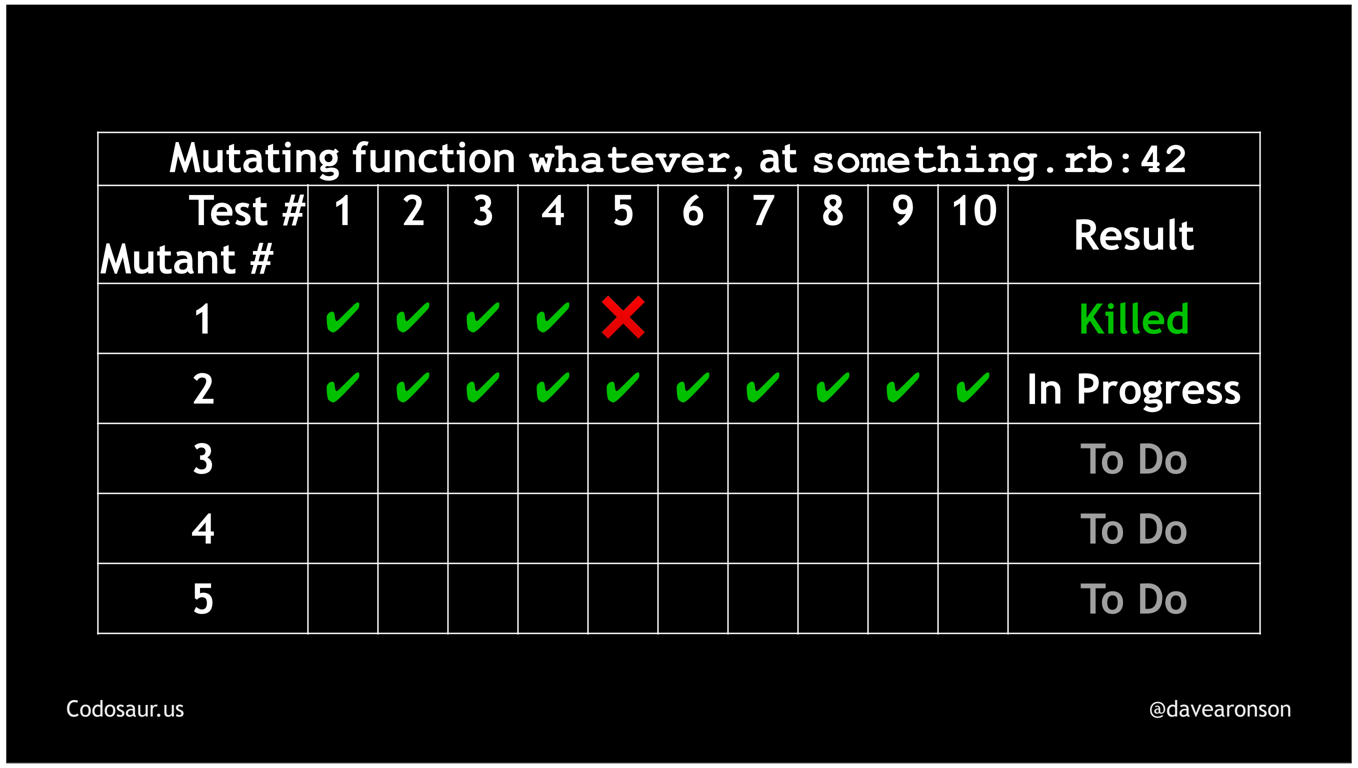
Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗						Killed
2	⏳										In Progress
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . tests six through ten for Mutant #1. Like so much in computers, we only care about ones and zeroes.

On the other claw, if a mutant . . .

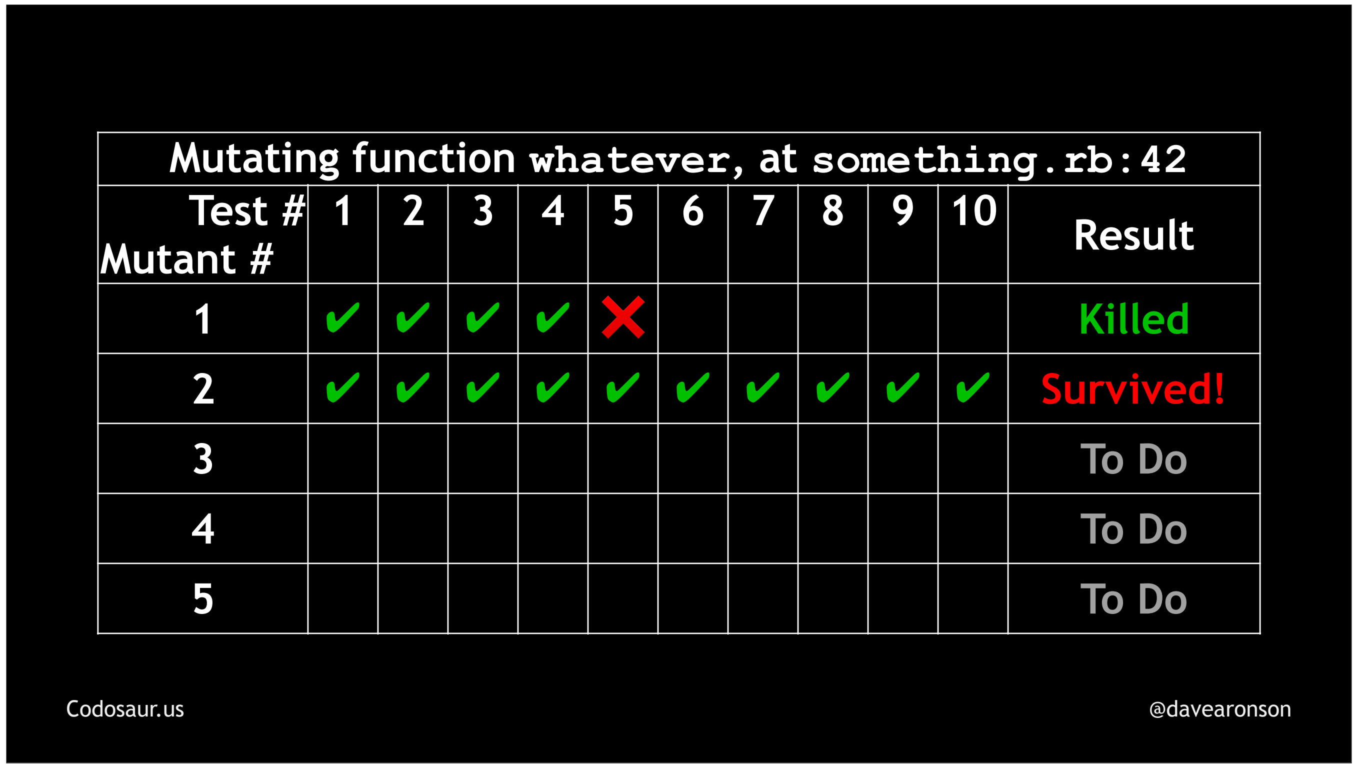


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗						Killed
2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	In Progress
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

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



Mutating function `whatever`, at `something.rb: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 . . .

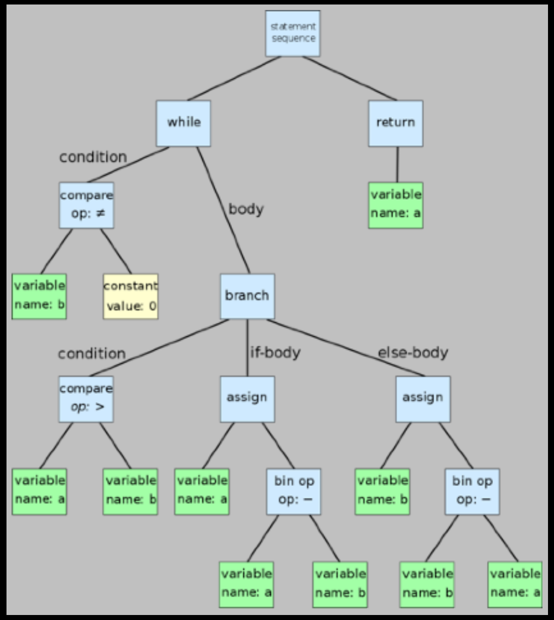
Codosaur.us

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 . . .

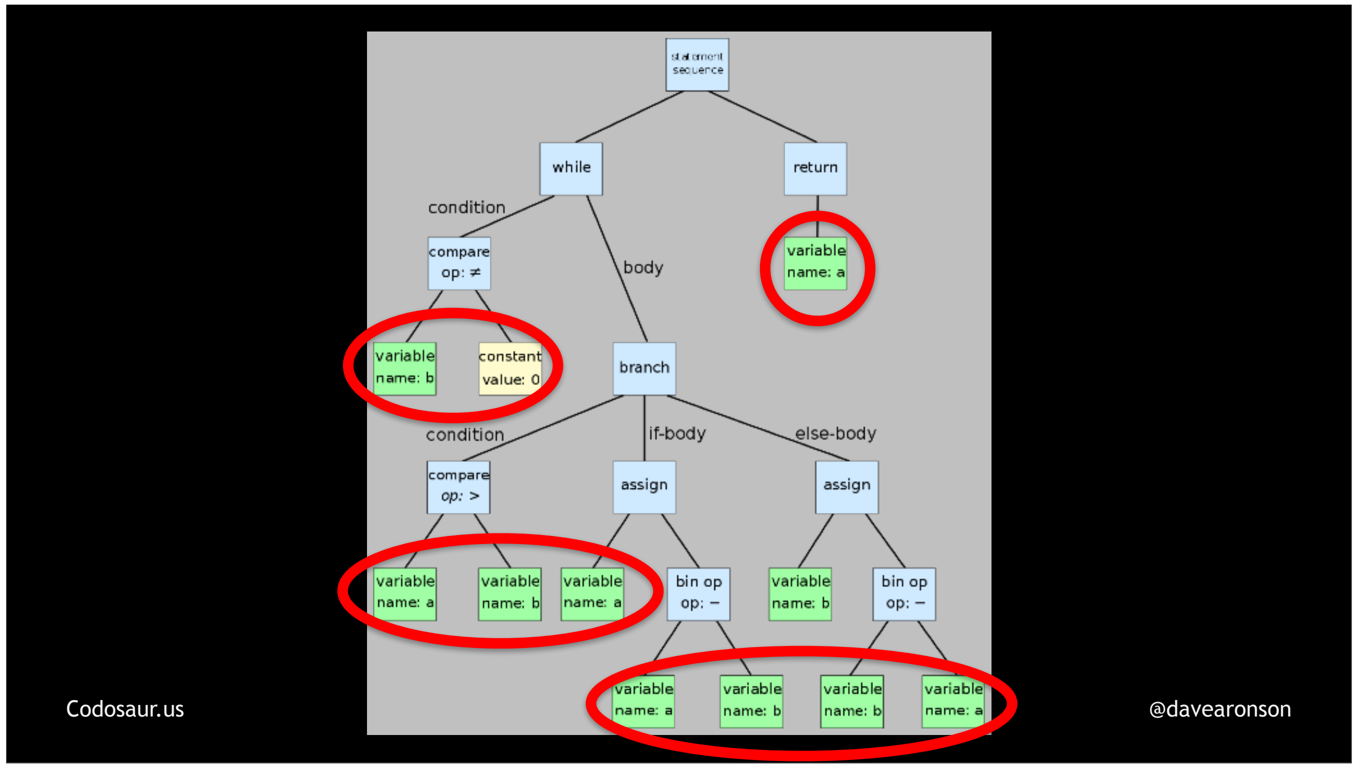
```
def euclid(a, b)
  while b != 0
    if a > b
      a -= b
    else
      b -= a
    end
  end
  a
end
```



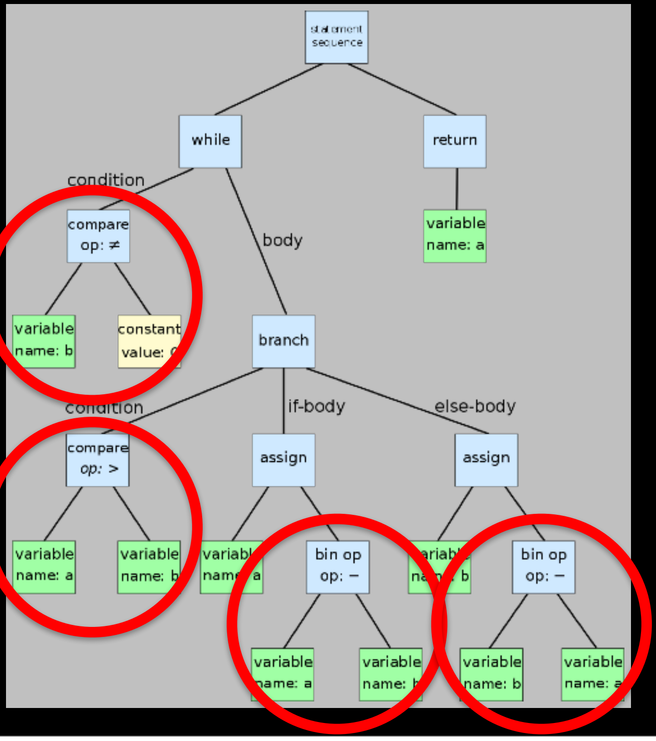
Codosaur.us Image: https://commons.wikimedia.org/wiki/File:Abstract_syntax_tree_for_Euclidean_algorithm.svg @davearonson

. . . parses our code, usually into an Abstract Syntax Tree. There are some tools that work differently, like working on bytecode, and some even work on the actual written source code, but most use an AST, so let's roll with that. (I know those boxes are too small to read easily, but we don't need to understand this one in detail.)

A brief refresher in case you haven't dealt with an AST lately: it's a tree structure that represents the semantic meaning of our code. The . . .



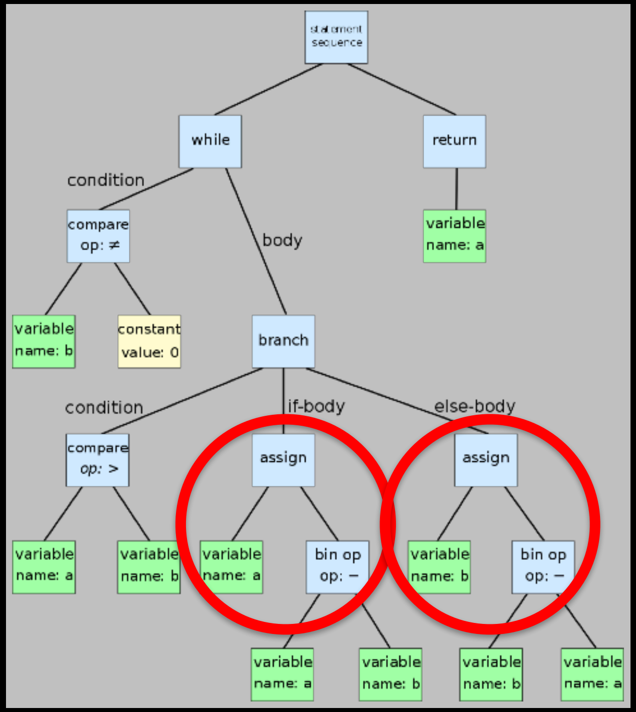
. . . leaf nodes are generally *values* such as variables, literals, or symbolic constants. The other nodes and their children are generally things such as . . .



Codosaur.us

@davearonson

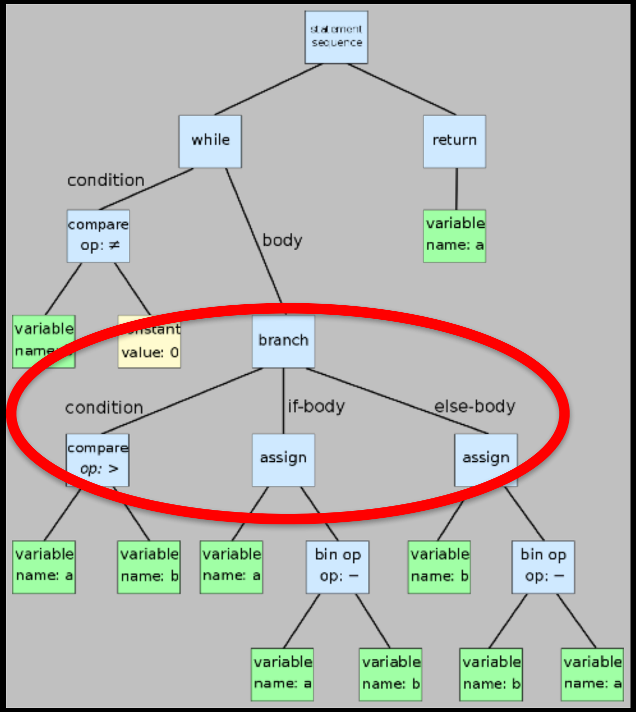
. . . an operation and its operands, such as these additions and comparisons, a function call and its arguments (though we don't have any of those on this AST), . . .



Codosaur.us

@davearonson

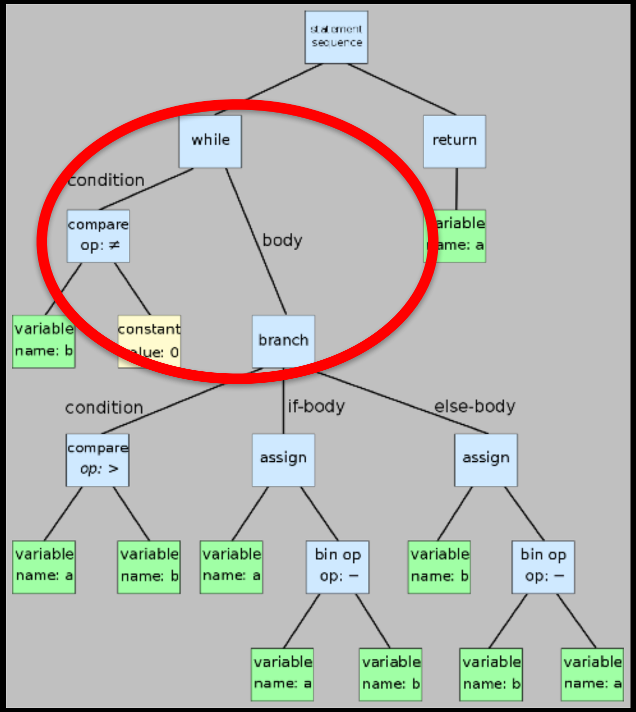
. . . an assignment and its source and destination, or a layer of structure, such as . . .



Codosaur.us

@davearonson

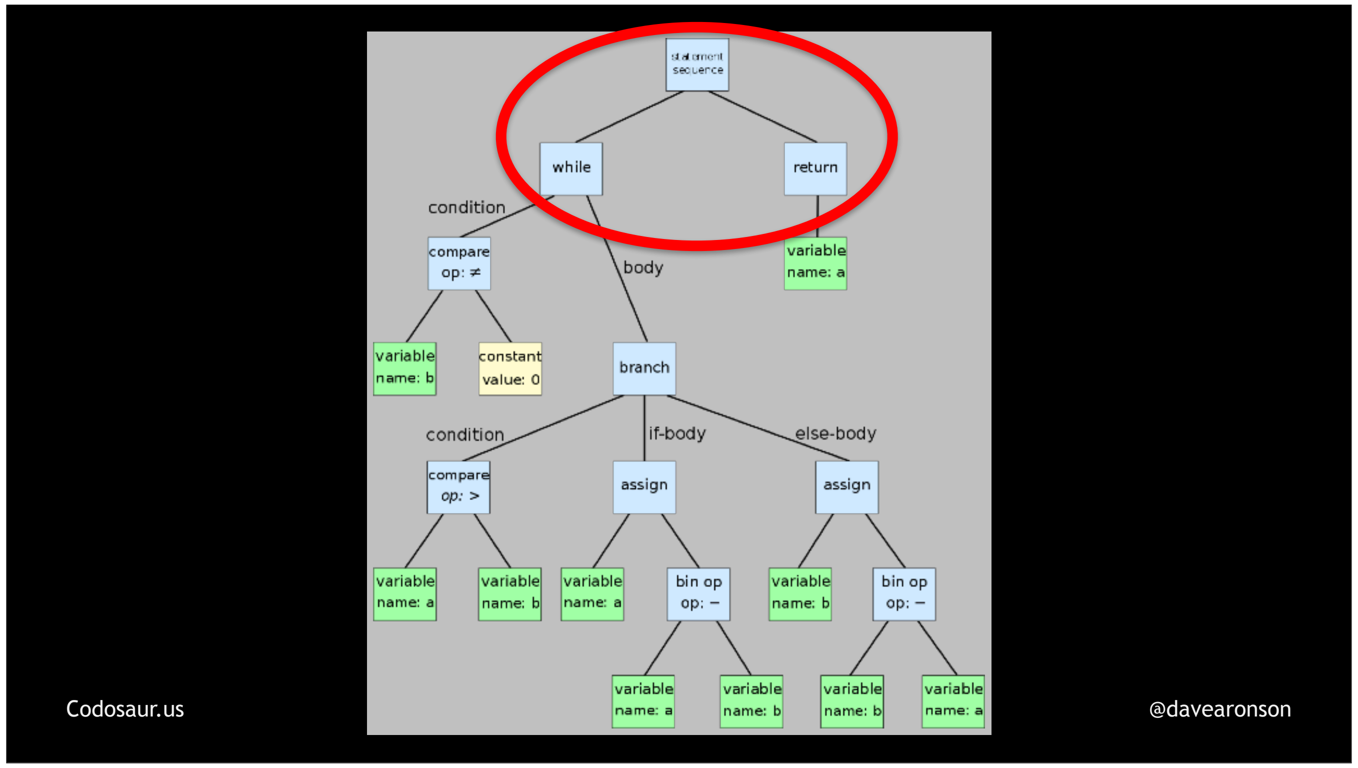
. . . a branch and its conditions and statements, like this if-else . . .



Codosaur.us

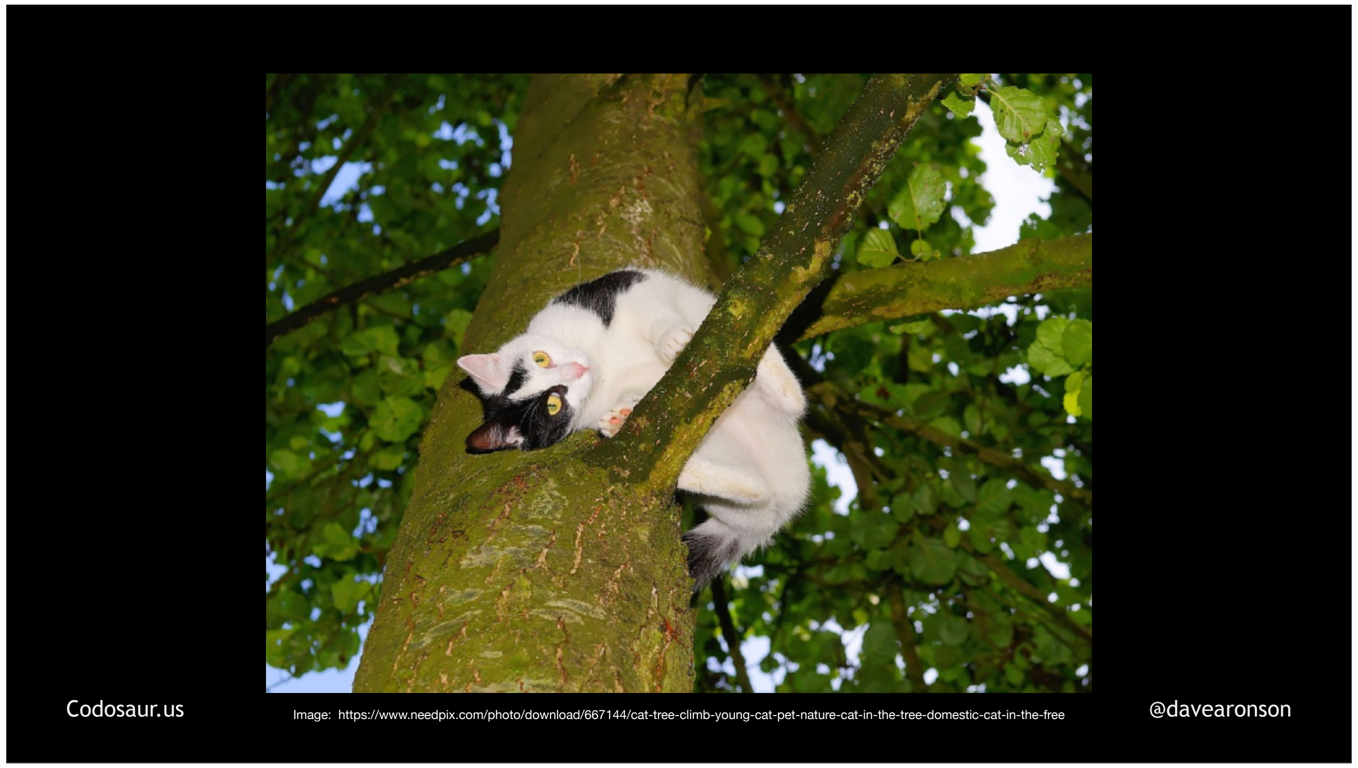
@davearonson

. . . a loop and *its* conditions and statements, like this while-loop, . . .



. . . a function declaration and its parameters (not shown on this AST) and statements, or a class or module declaration and its functions, variables, constants and so on (but we don't have an example of that on this AST).

After our mutation testing tool creates an AST out of our code, then it . . .

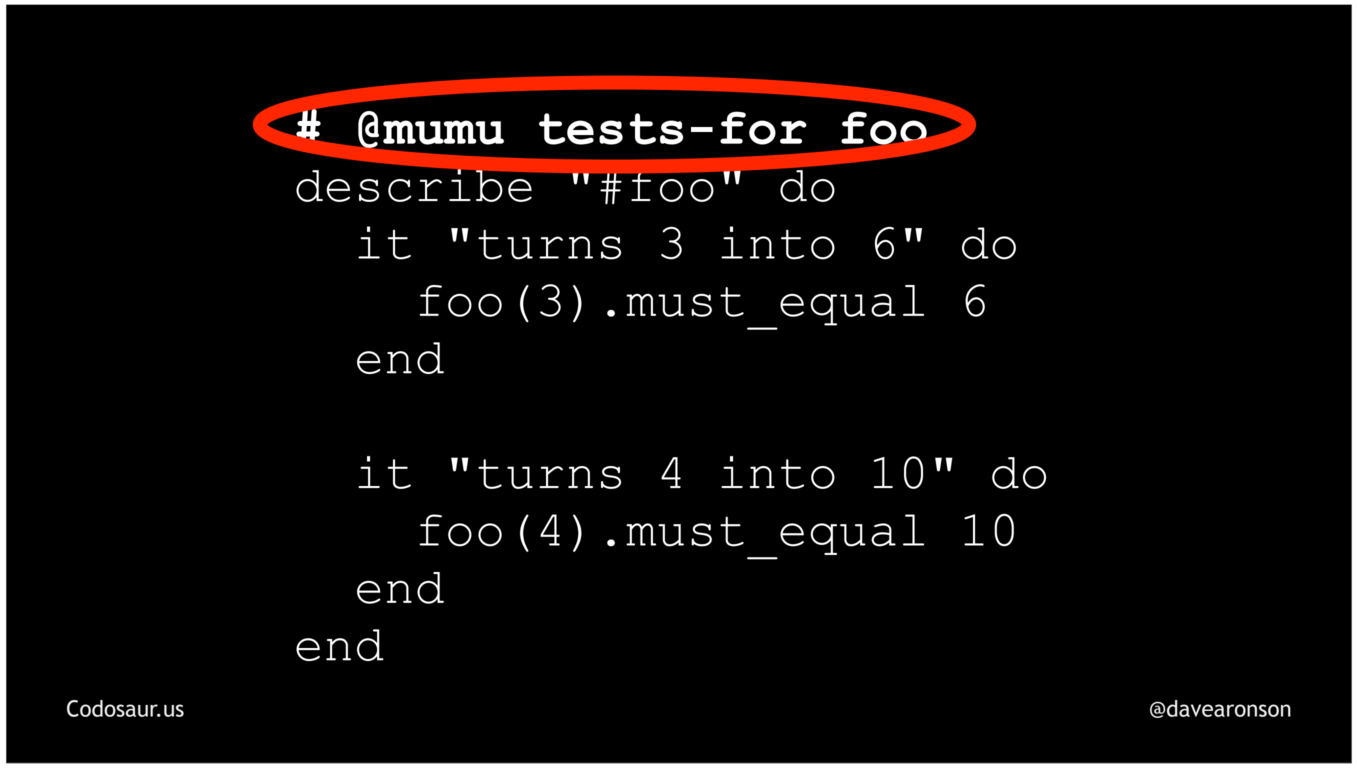


Codosaur.us

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

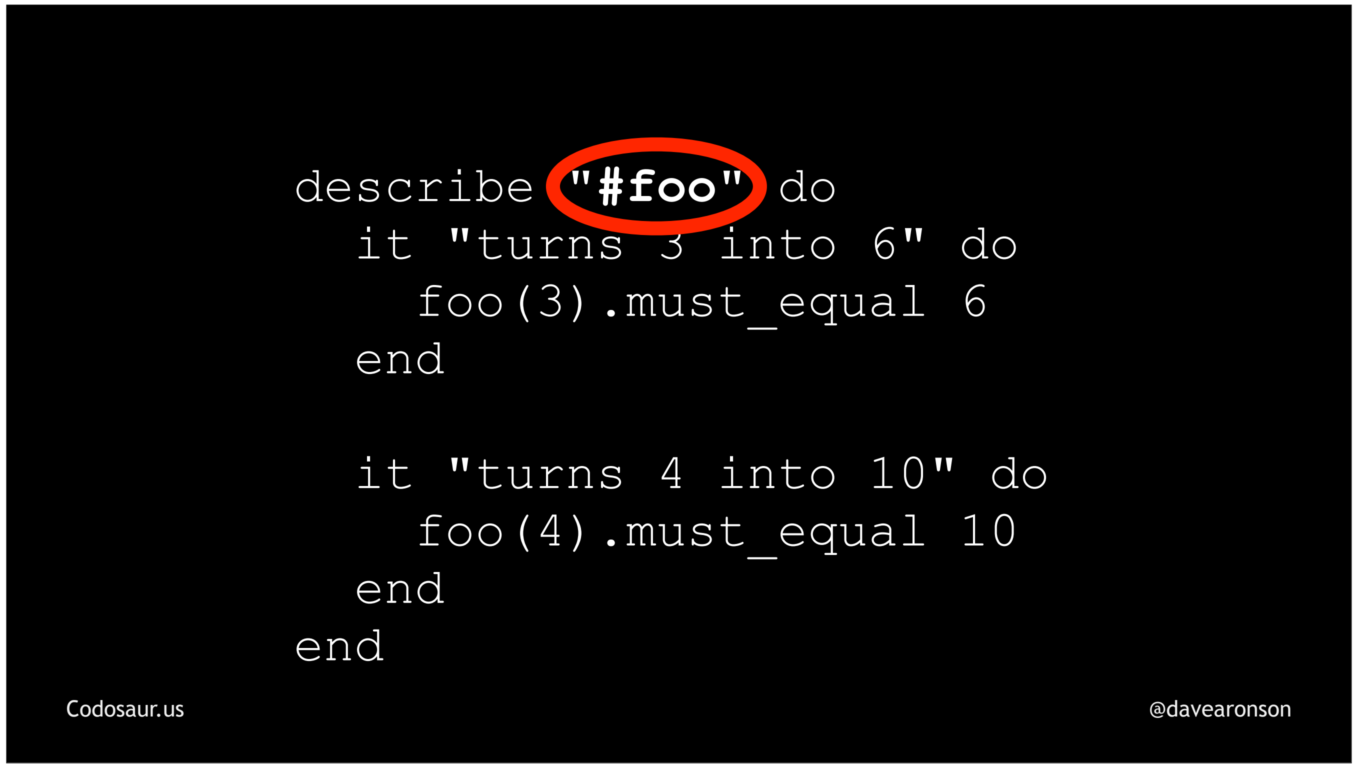
@davearonson

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



```
# @mumu tests-for foo
describe "#foo" do
  it "turns 3 into 6" do
    foo(3).must_equal 6
  end

  it "turns 4 into 10" do
    foo(4).must_equal 10
  end
end
```

. . . annotating our tests, as I hinted at earlier, or following some kind of . . .

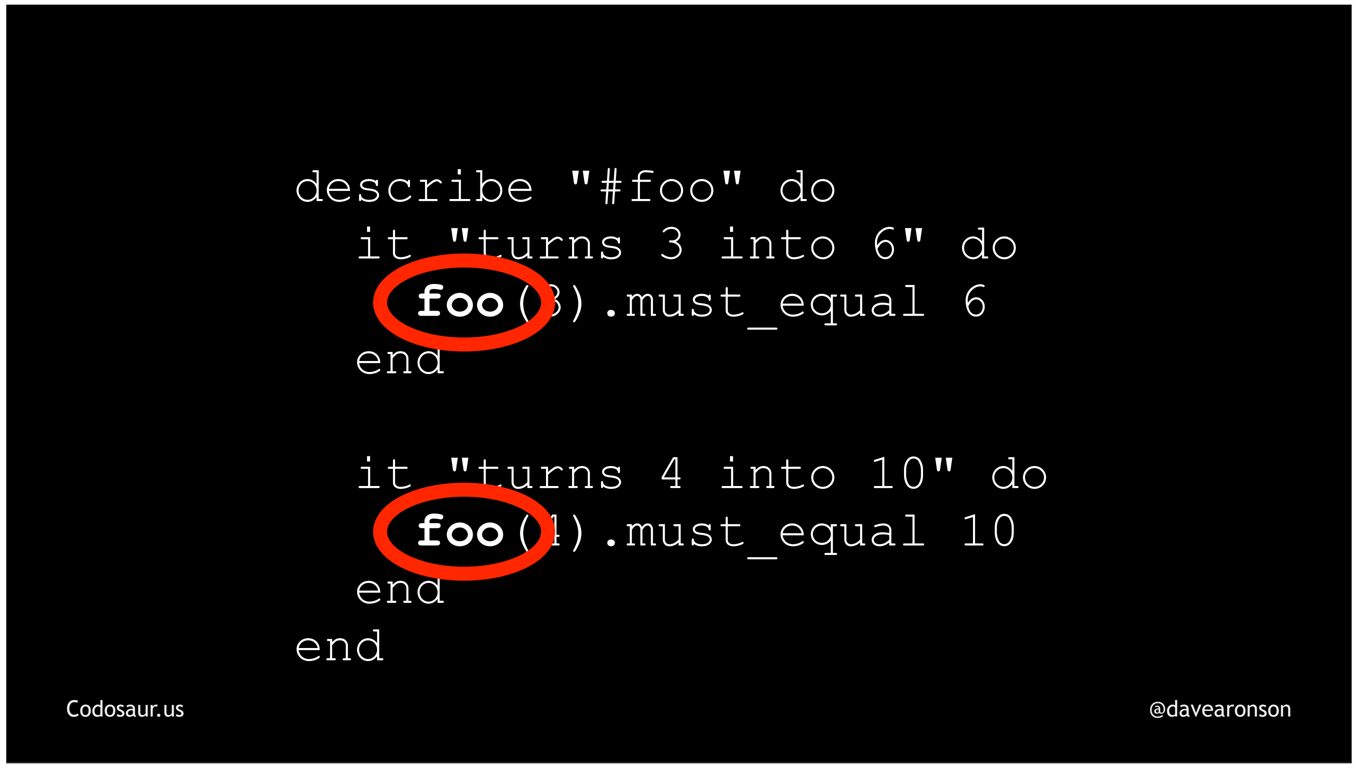


```
describe "#foo" do
  it "turns 3 into 6" do
    foo(3).must_equal 6
  end

  it "turns 4 into 10" do
    foo(4).must_equal 10
  end
end
```

Codosaur.us

@davearonson

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

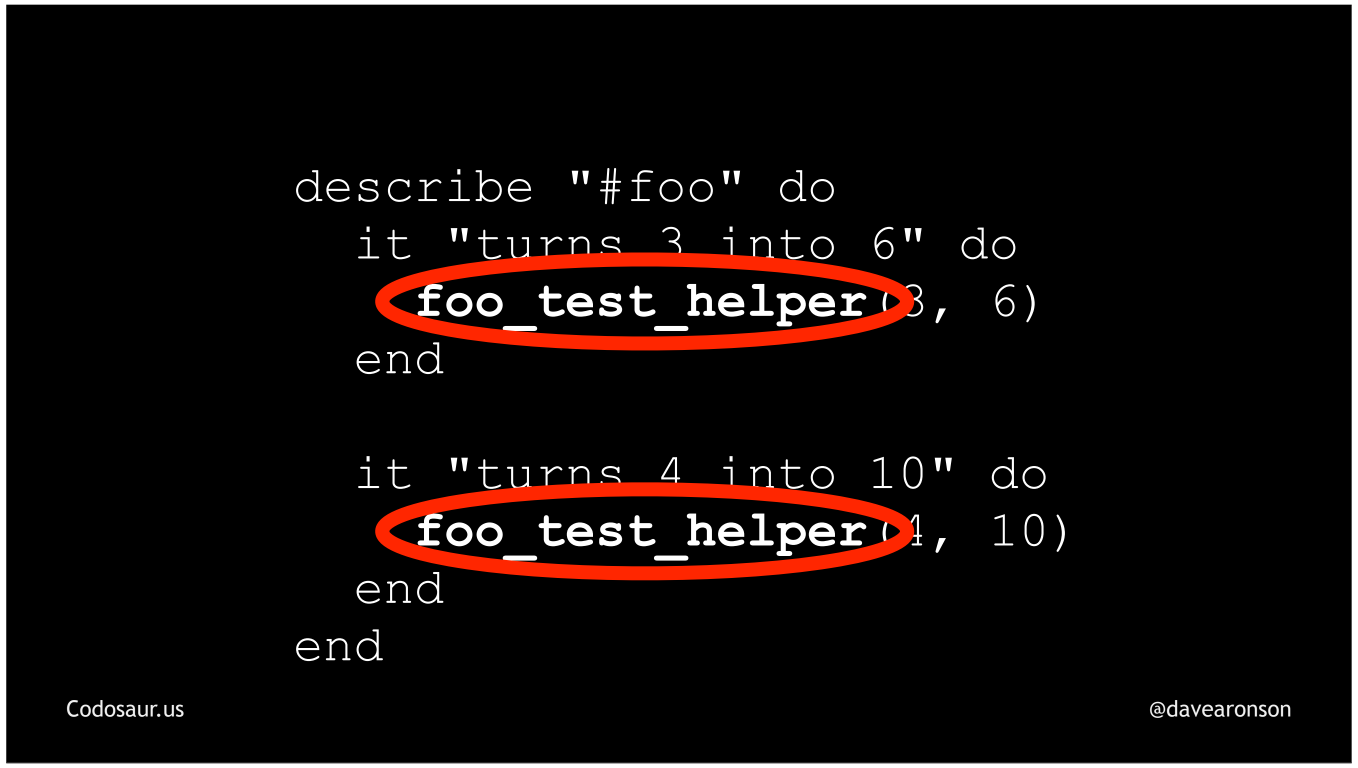


```
describe "#foo" do
  it "turns 3 into 6" do
    foo(3).must_equal 6
  end

  it "turns 4 into 10" do
    foo(4).must_equal 10
  end
end
```

Codosaur.us

@davearonson

. . . the tool looking at what tests call what functions. However, that can get tricky if . . .



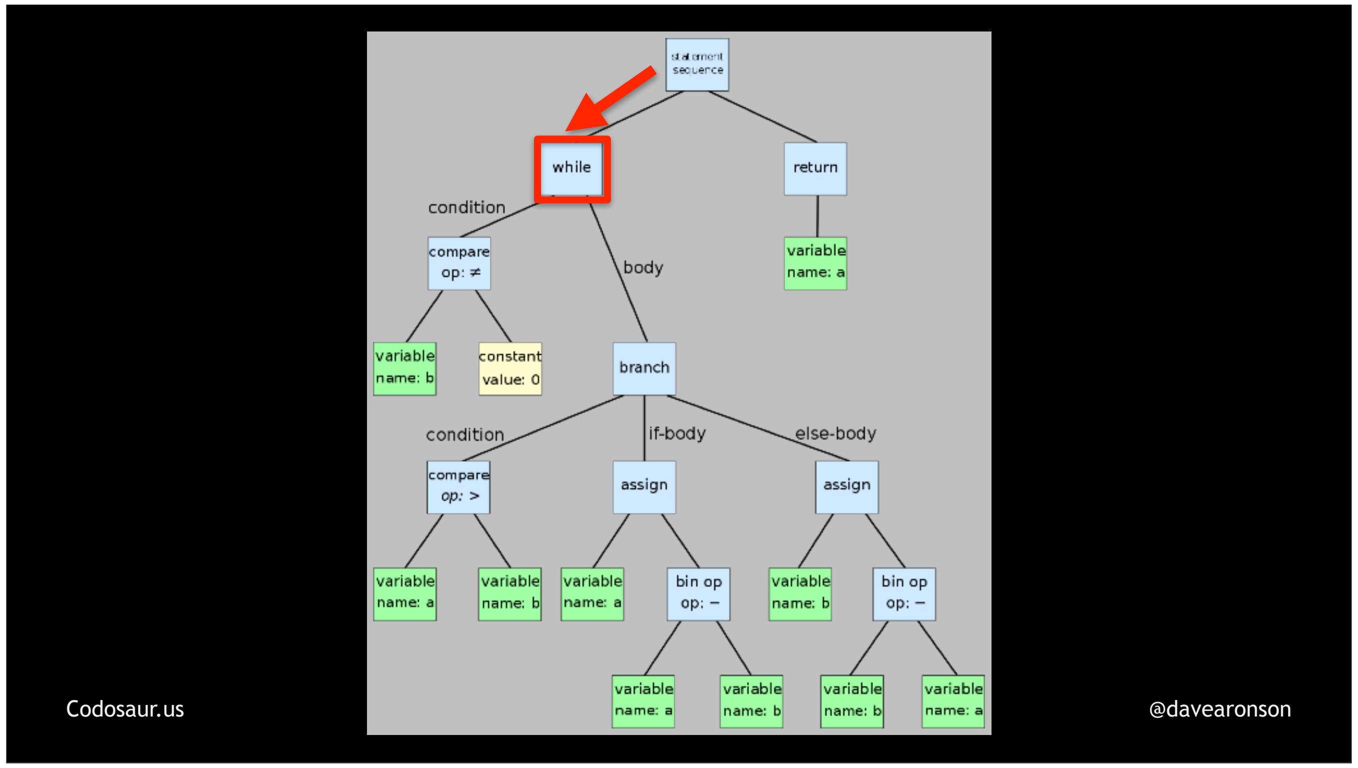
```
describe "#foo" do
  it "turns 3 into 6" do
    foo_test_helper(3, 6)
  end

  it "turns 4 into 10" do
    foo_test_helper(4, 10)
  end
end
```

Codosaur.us

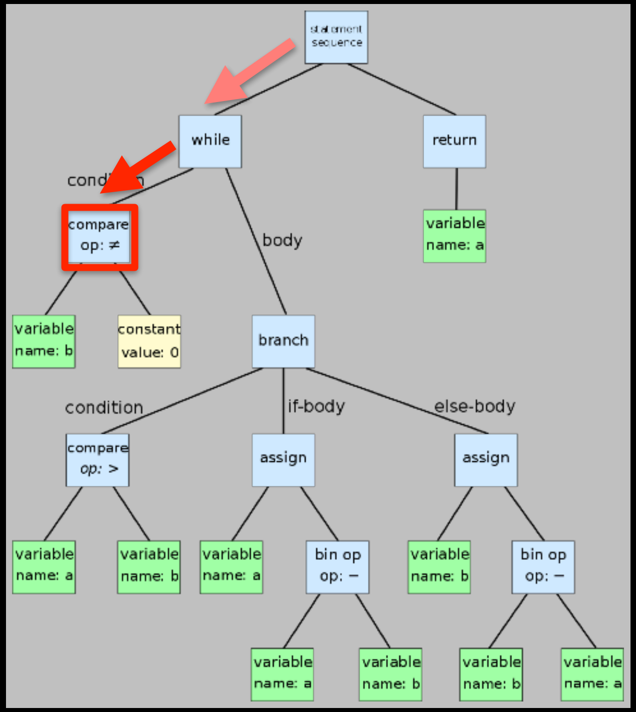
@davearonson

. . . the function isn't called *directly* from the test. In that case, it usually involves looking at the test *coverage* data or some such gathered information. (PAUSE!) 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. However, now, instead of looking for even smaller *subtrees* it can *extract*, like twigs or something, it looks 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 AST I showed earlier, and has only gotten down to . . .



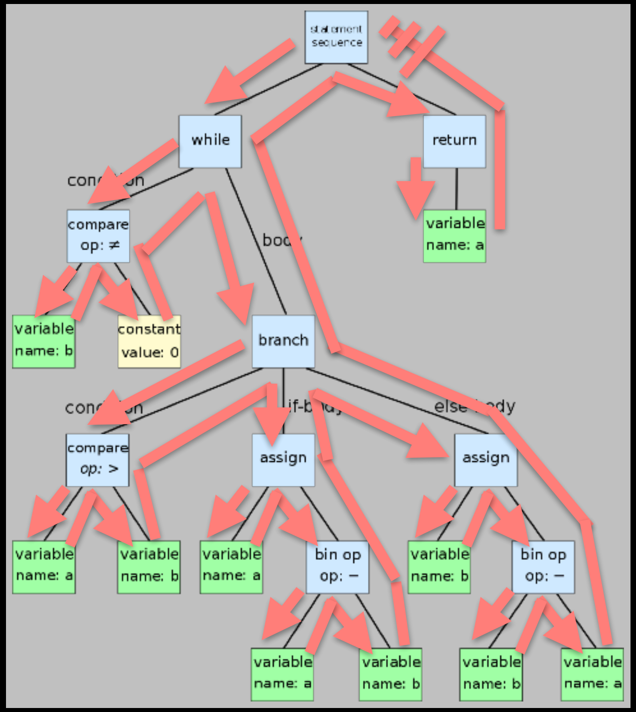
. . . this while loop, following that arrow. Some tools, in the interests in speed, will only make *one* change per node, usually the smallest one it can. But most tools will apply a much larger range of changes, and we'll see some examples in a moment. Either way, for each way this tool could change 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 from *that* node, it would continue traversing the subtree, down to . . .



Codosaur.us

@davearonson

. . . that node's first child-node, the conditional that controls it, a not-equal comparison. 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 . . .

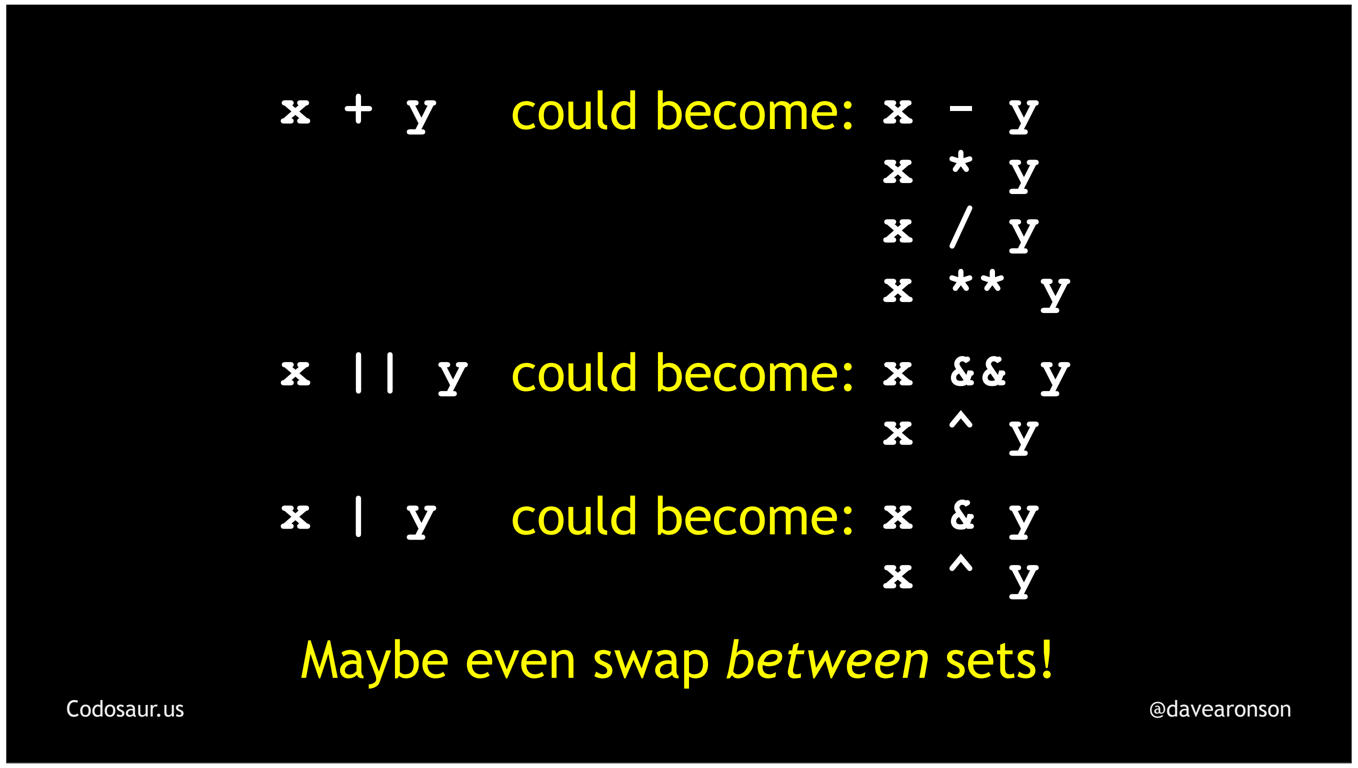


Codosaur.us

@davearonson

. . . traversed the entire subtree.

Now, I've been talking a lot about mutating things and changing them. 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!

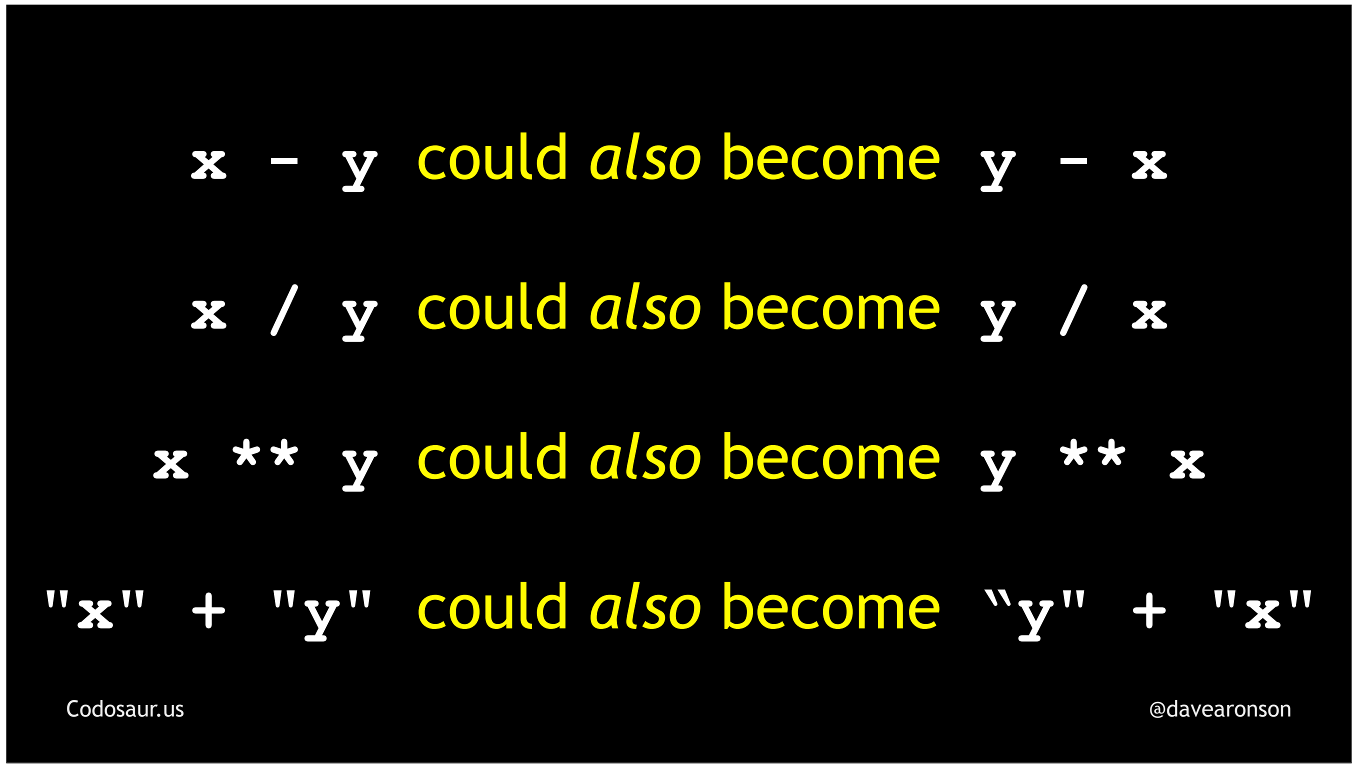
Codosaur.us

@davearonson

It could change a mathematical, logical, or bitwise operator from one to another.

In languages and situations where we can do so, it could even substitute an operator from a different category. For instance, in many languages, we can treat *anything* as *booleans*, so x *times* y could become, for instance, x *and* y, or x *exclusive-or* y.

But remember, some tools will only make one change per node, so for instance a plus will only be mutated into a minus, while with other tools it may be mutated into *all* of these, or at least some larger subset.



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

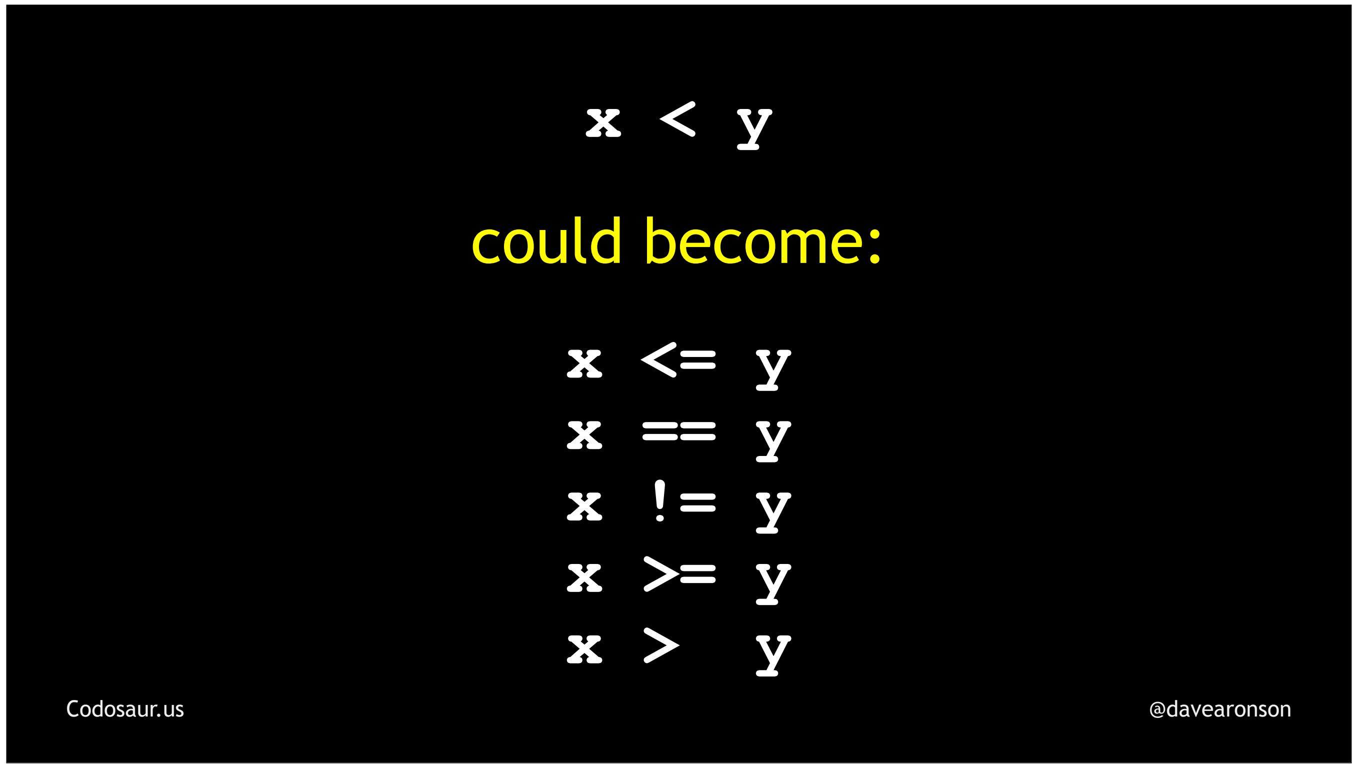
`x / y` could *also* become `y / x`

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

`"x" + "y"` could *also* become `“y" + "x"`

Codosaur.us @davearonson

When the *order* of operands matters, such as in subtraction, division, exponentiation, or string concatenation, it could *swap* them.



```
x < y
```

could become:

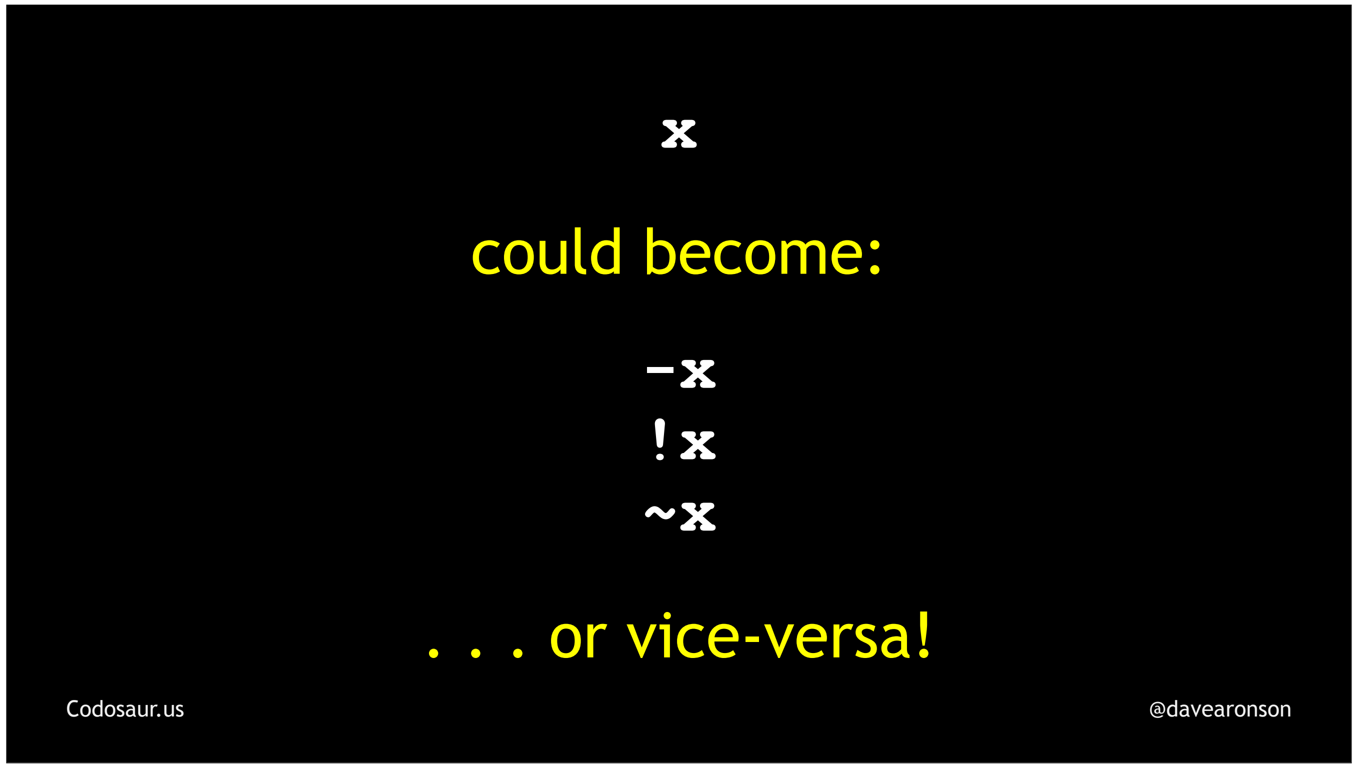
```
x <= y
x == y
x != y
x >= y
x >  y
```

Codosaur.us

@davearonson

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

Again, some tools only make one change per node, so for instance less-than will only be mutated into greater-than, or possibly less-than-*or-equal*, while with other tools it may be mutated into *any* other comparison. I think you get that idea by now, so I'm not going to repeat it. Also, some tools use only semantic substitution or reduction but never *addition*, so some might turn less-than into greater-than or into equals, but never less-or-equal or greater-or-equal, since that could be interpreted as adding semantics, in the “or” part.



```
x
```

could become:

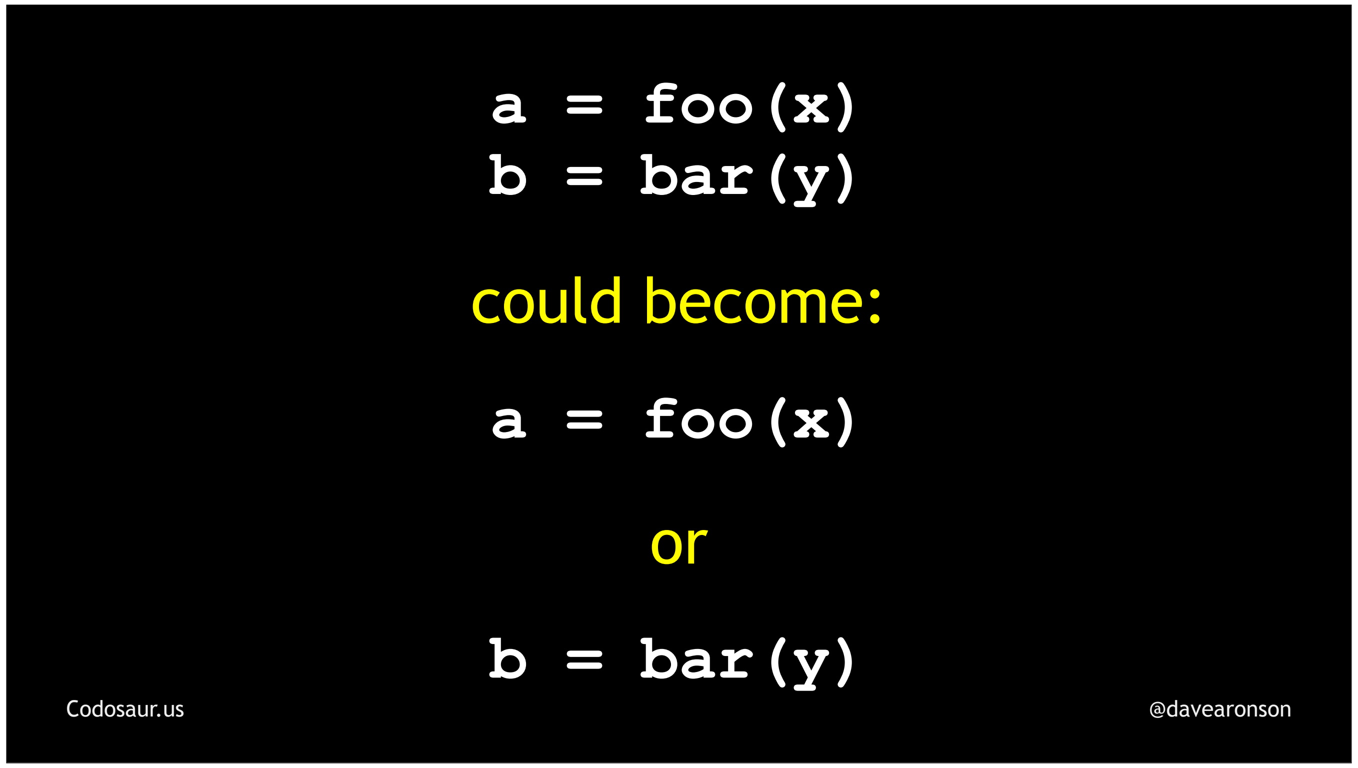
```
-x
!x
~x
```

. . . 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)
```

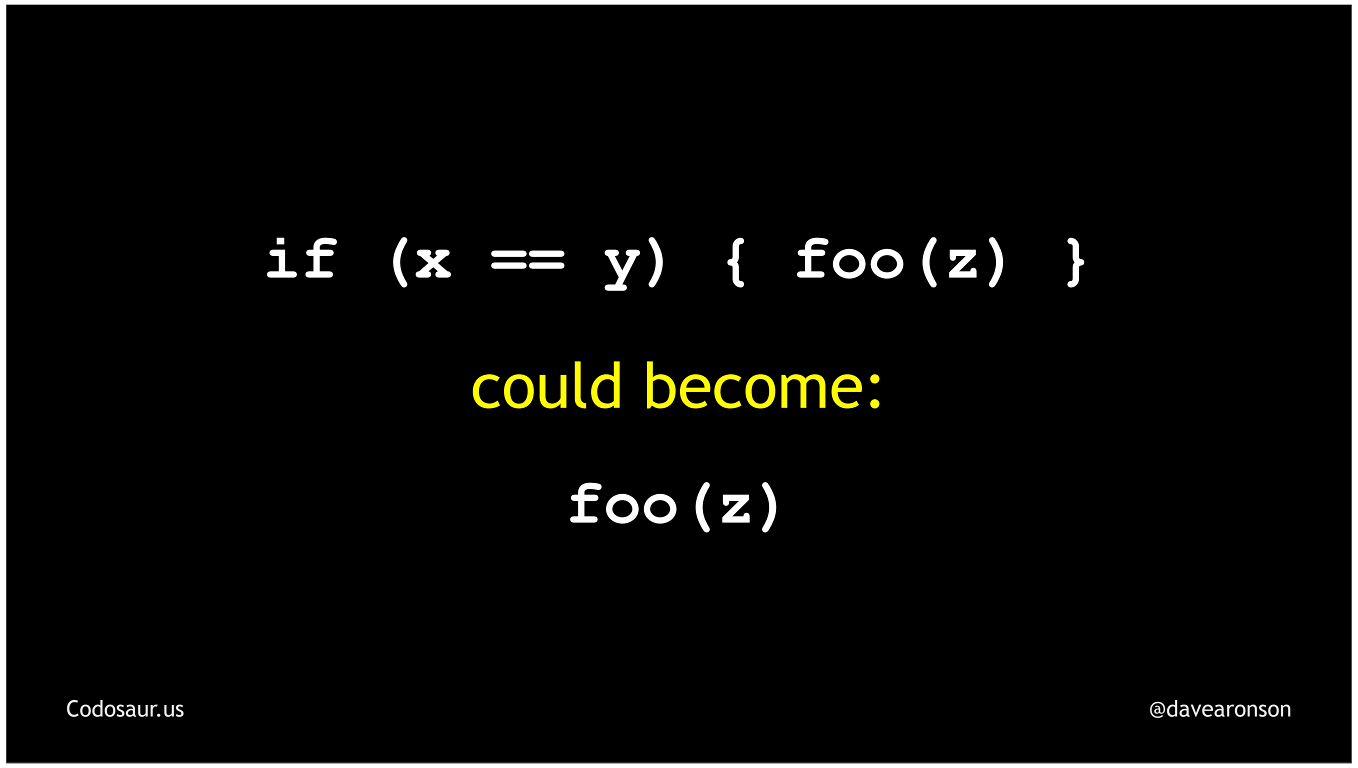
or

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

Codosaur.us

@davearonson

It can remove entire lines of code, though usually because it's a *statement*, rather than looking at the *physical written lines* of the source code.



```
if (x == y) { foo(z) }
```

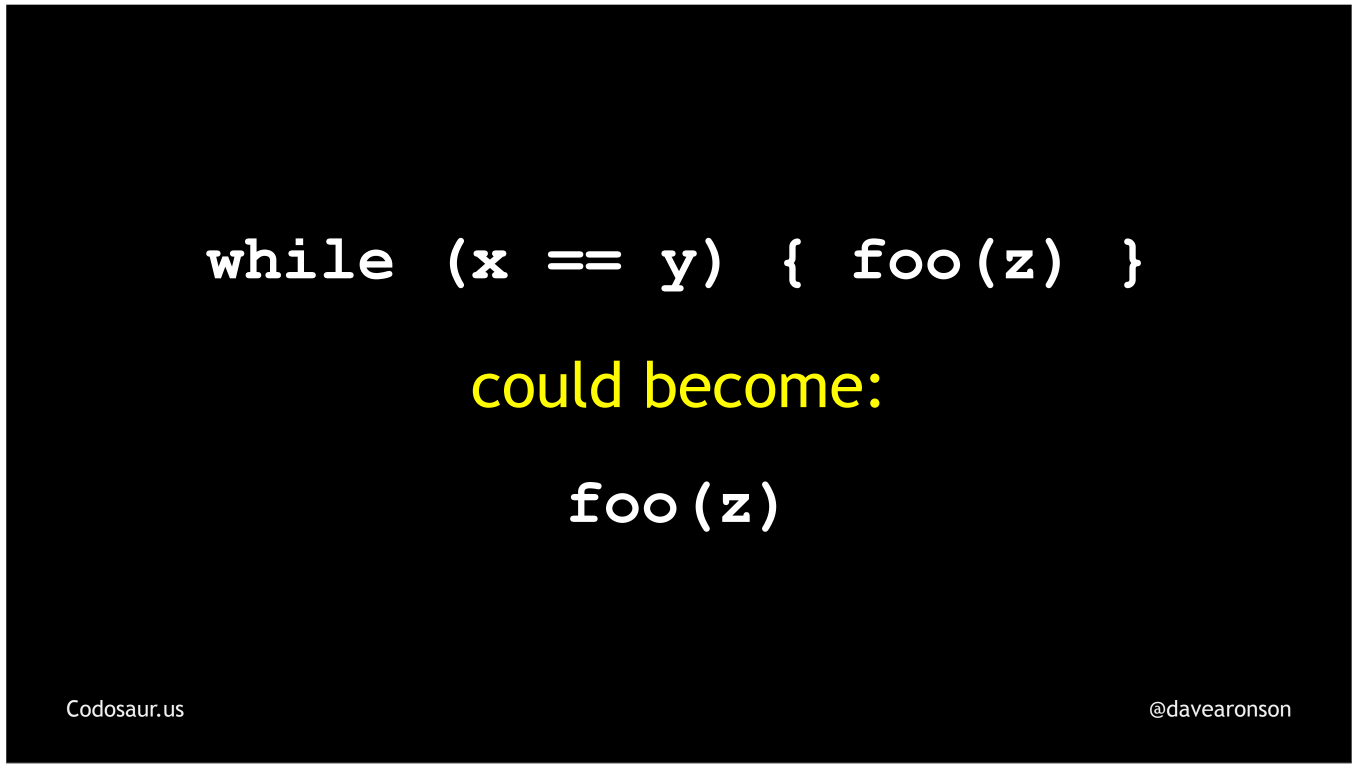
could become:

```
foo(z)
```

Codosaur.us

@davearonson

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



```
while (x == y) { foo(z) }
```

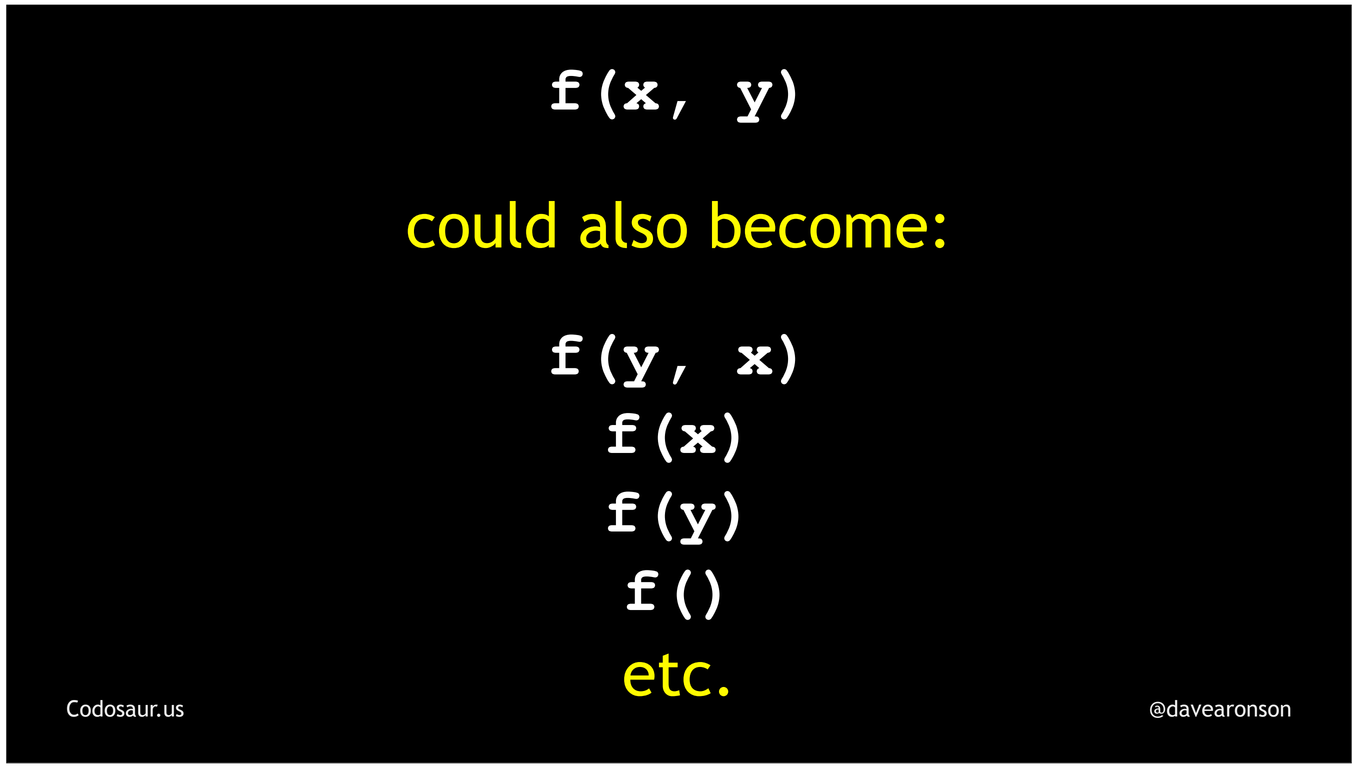
could become:

```
foo(z)
```

Codosaur.us

@davearonson

It can remove a loop control, so that something that might be skipped, done once, or done multiple times, is always done exactly once.



```
f(x, y)
```

could also become:

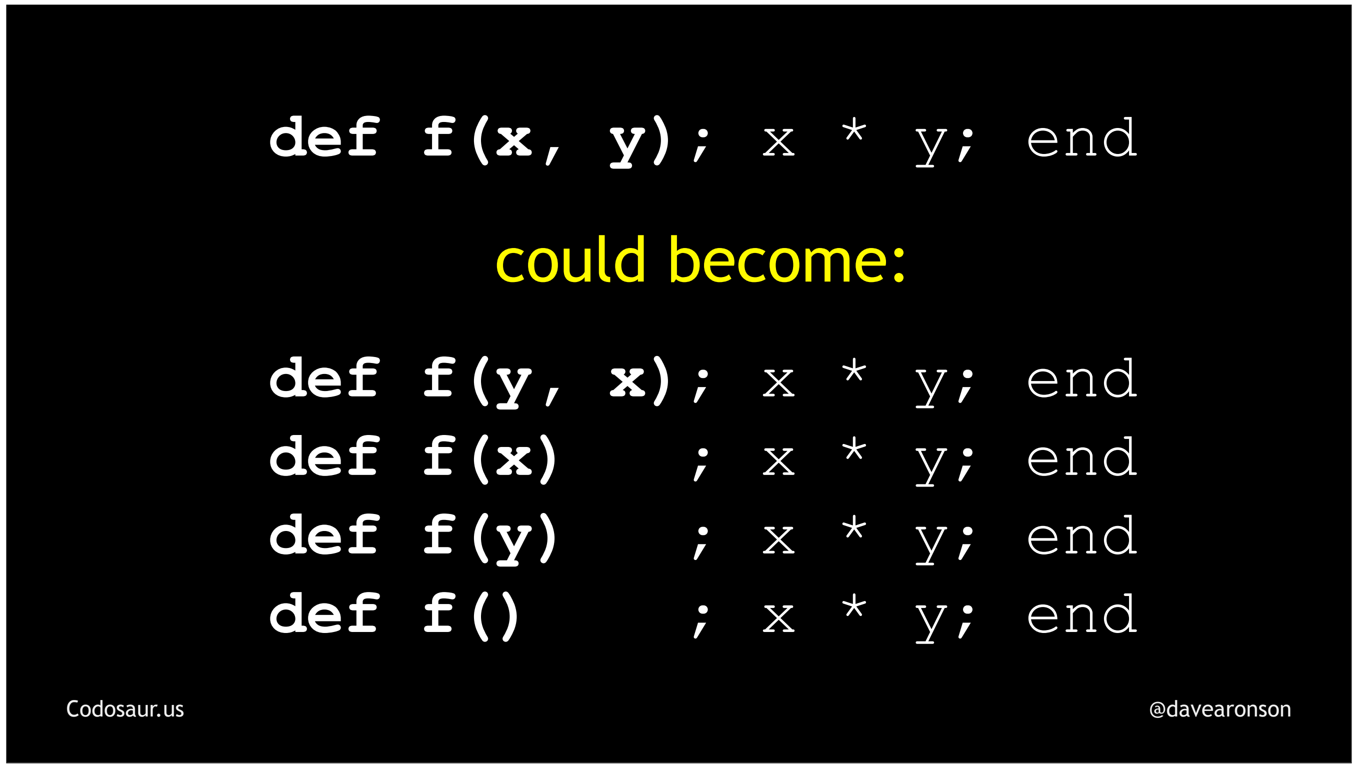
```
f(y, x)
f(x)
f(y)
f()
```

etc.

Codosaur.us

@davearonson

It could theoretically, but very few do, *scramble* or *truncate* argument lists of function calls. Truncation will often result in a syntax error, from having the wrong number of arguments. However, if our language allows default or variadic arguments, as most modern languages do, this could be perfectly fine.



```
def f(x, y); x * y; end
```

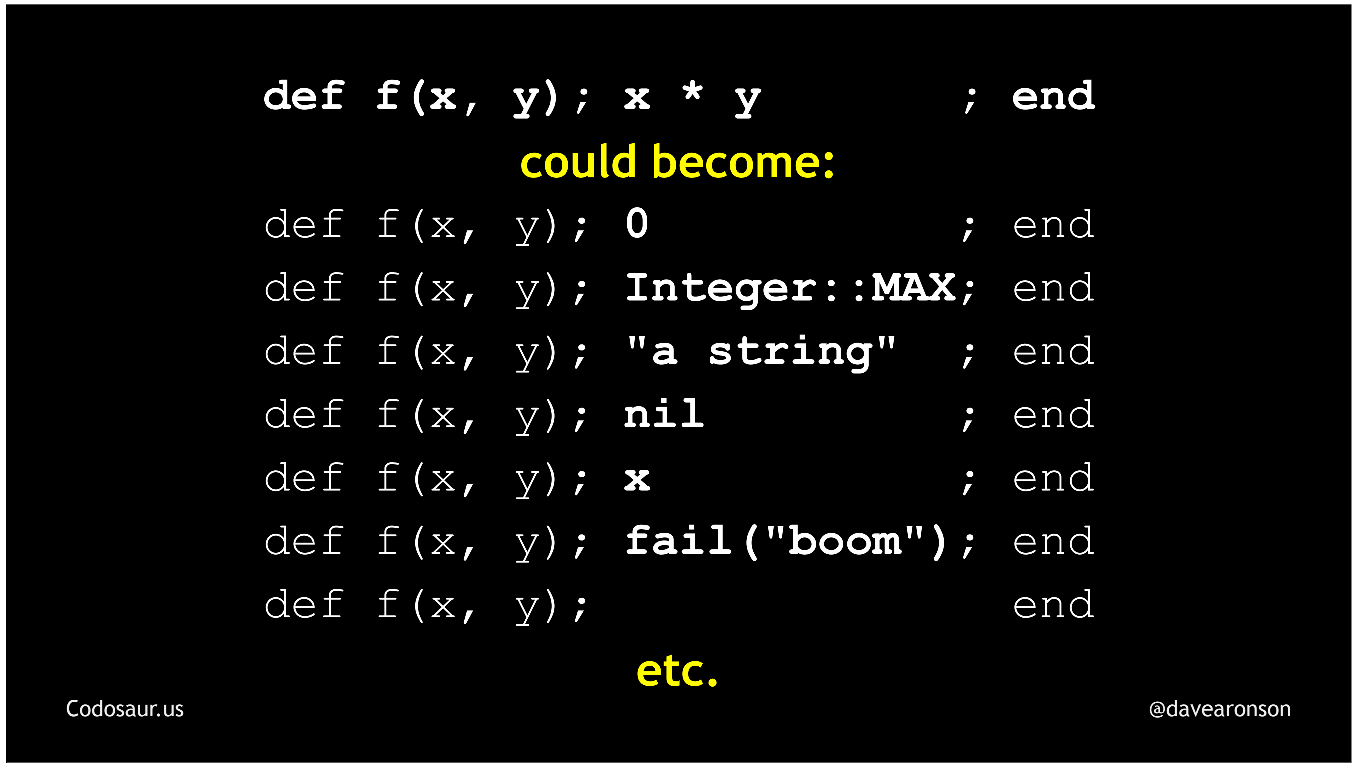
could become:

```
def f(y, x); x * y; end
def f(x)   ; x * y; end
def f(y)   ; x * y; end
def f()    ; x * y; end
```

Codosaur.us

@davearonson

It could *also* (theoretically but rarely) scramble or truncate argument lists of function *declarations*. This will *usually* result in a syntax error where it's called from, because of the call having too *many* arguments, or *inside* the function, from having an unknown variable, but sometimes not! When the call still works, that can help reveal times when we're shadowing a variable in an outer scope, IOW, using the same name. That isn't necessarily *wrong*, but it can be *dangerous*, or at least *confusing*, making for bad maintainability. Of course, any decent compiler or interpreter should warn us about that anyway, but we don't always have good tools, or pay attention to all the warnings. Moving on . . .



```
def f(x, y); x * y         ; end
```

could become:

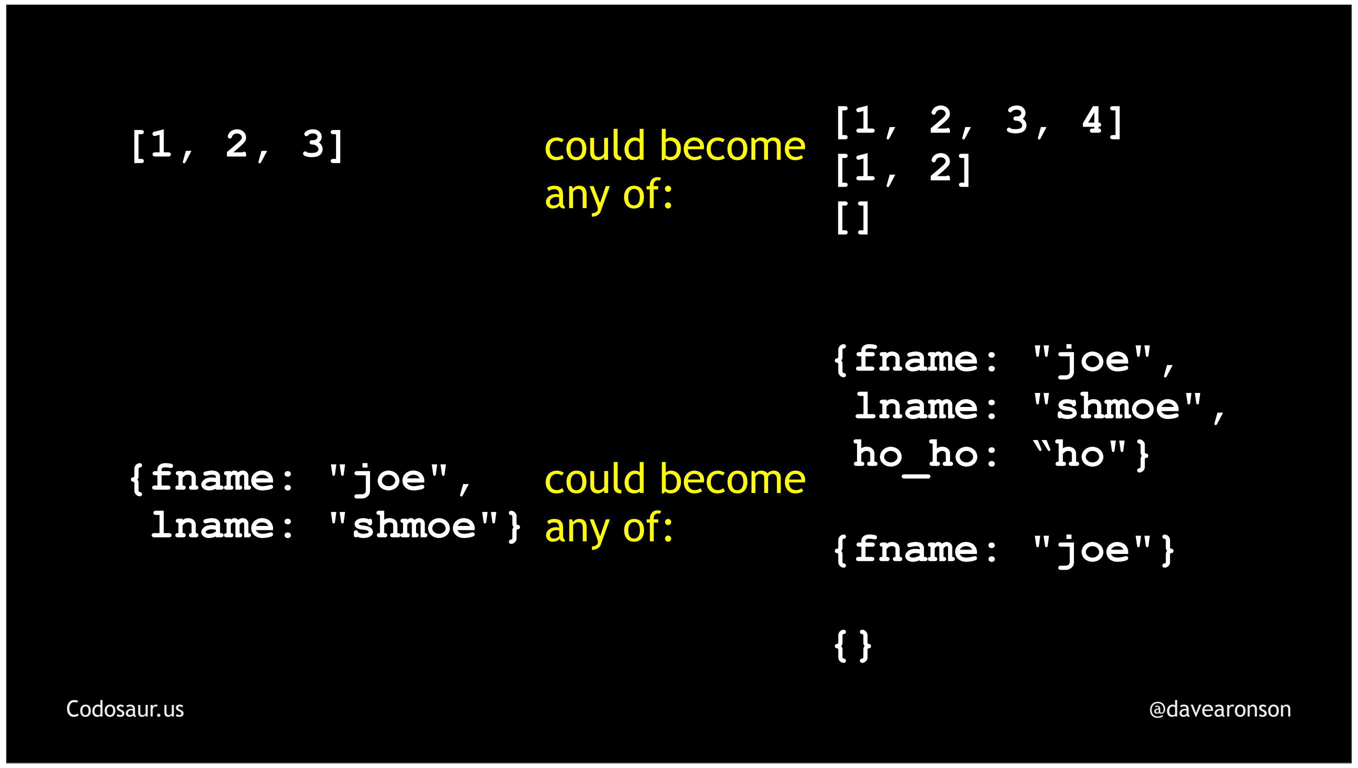
```
def f(x, y); 0             ; end
def f(x, y); Integer::MAX; end
def f(x, y); "a string"   ; end
def f(x, y); nil           ; end
def f(x, y); x             ; end
def f(x, y); fail("boom"); end
def f(x, y);                 end
```

etc.

Codosaur.us

@davearonson

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. There is even a style called *EXTREME* Mutation Testing, that *only* uses complete removal of the function body! This means a lot fewer mutants so it's faster, and it's clearer what should be done about it, and there are fewer false alarms, so it makes a great quick and dirty first pass… but of course it's nowhere near so thorough so you'll want to follow it up with regular-style.



```
[1, 2, 3]
```

could become any of:

```
[1, 2, 3, 4]
[1, 2]
[]
```

```
{fname: "joe",
 lname: "shmoe"}
```

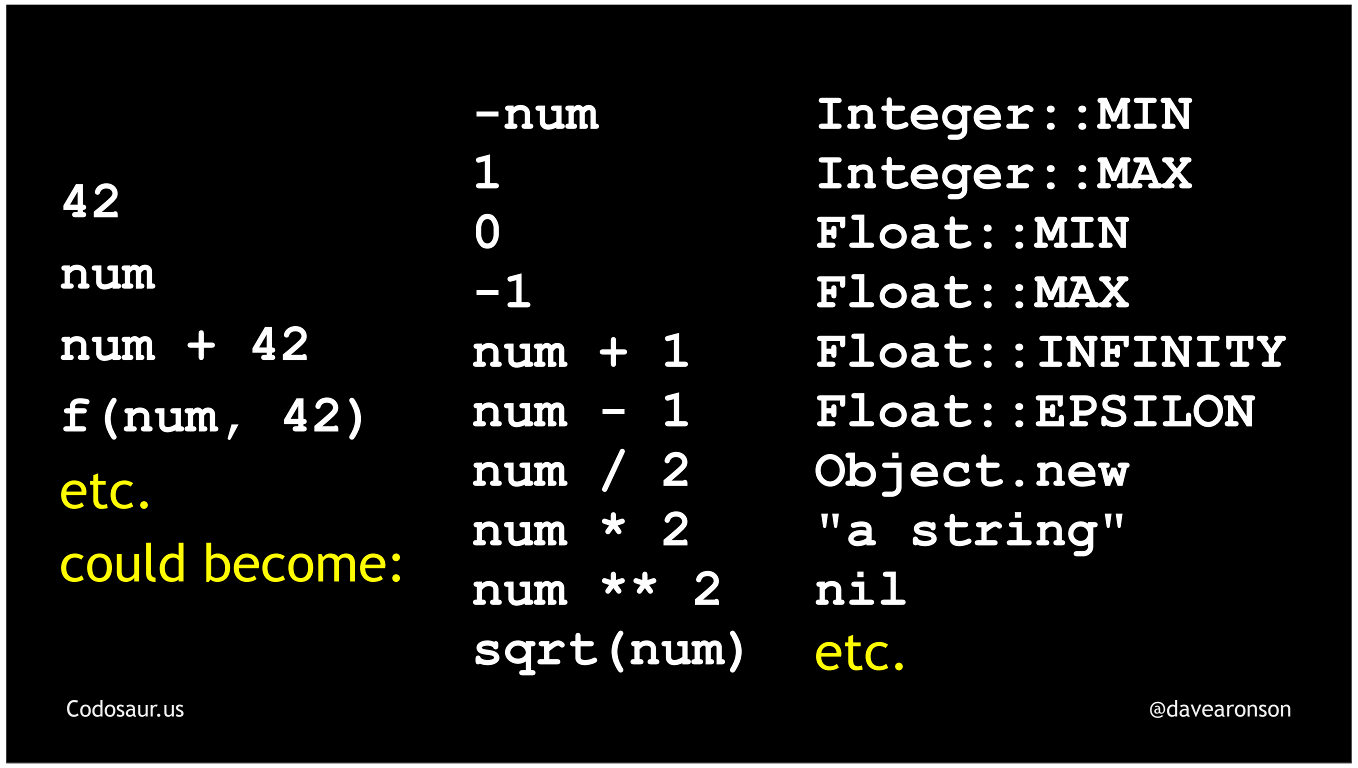
could become any of:

```
{fname: "joe",
 lname: "shmoe",
 ho_ho: "ho"}

{fname: "joe"}

{}
```

Codosaur.us

@davearonson

It could add things to, remove things from, or completely empty out, collections such as lists, tuples, maps, vectors, and so on. And of course it could mutate each item in the contents. For instance: . . .



```
42
num
num + 42
f(num, 42)
```
etc.
could become:

```
-num
1
0
-1
num + 1
num - 1
num / 2
num * 2
num ** 2
sqrt(num)
```

```
Integer::MIN
Integer::MAX
Float::MIN
Float::MAX
Float::INFINITY
Float::EPSILON
Object.new
"a string"
nil
```
etc.

Codosaur.us

@davearonson

It could change a constant or variable or expression or function call to some other value. It could even change it to something of an entirely different and incompatible type, such as changing a number into a, if I may quote . . .

Codosaur.us

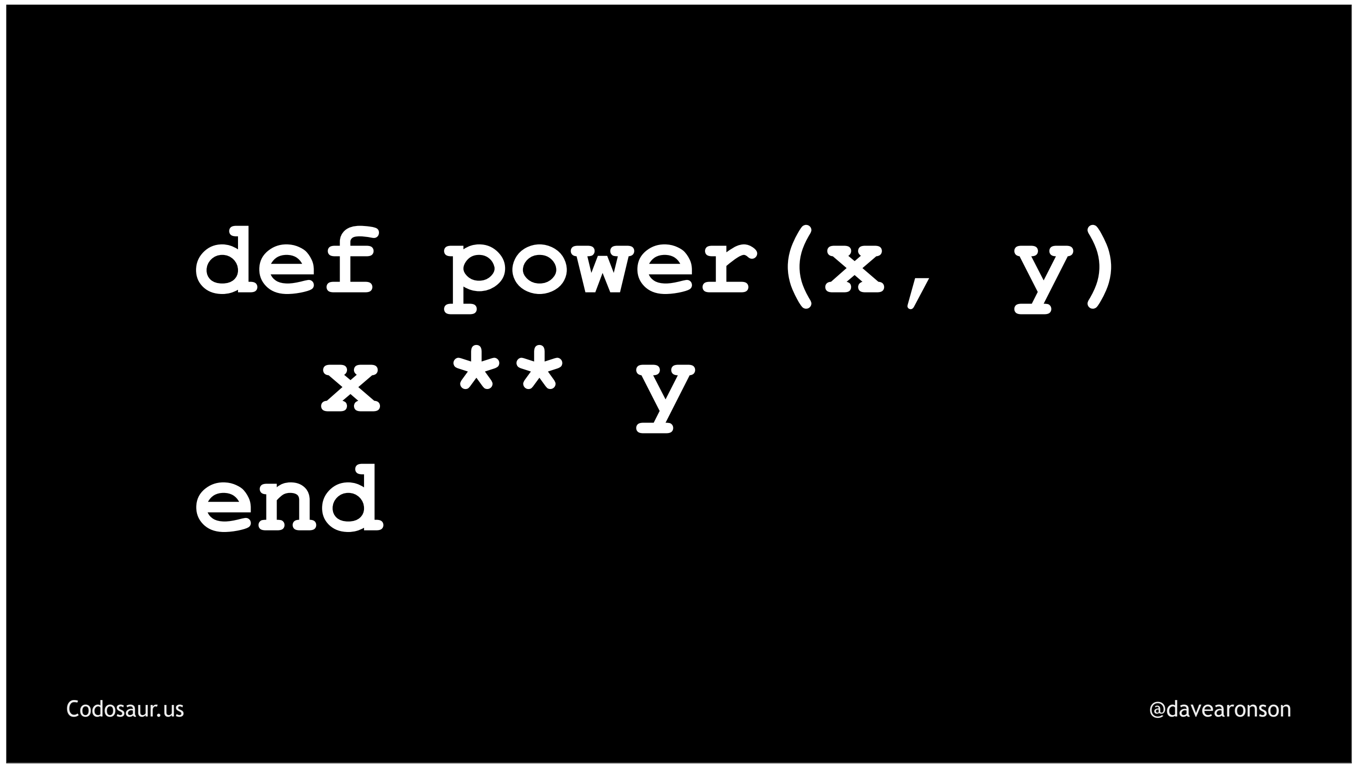
Image: https://www.flickr.com/photos/herval/50674160

@davearonson

. . . Smeagol, “string, or nothing!”

There are *many* many more, 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 . . .



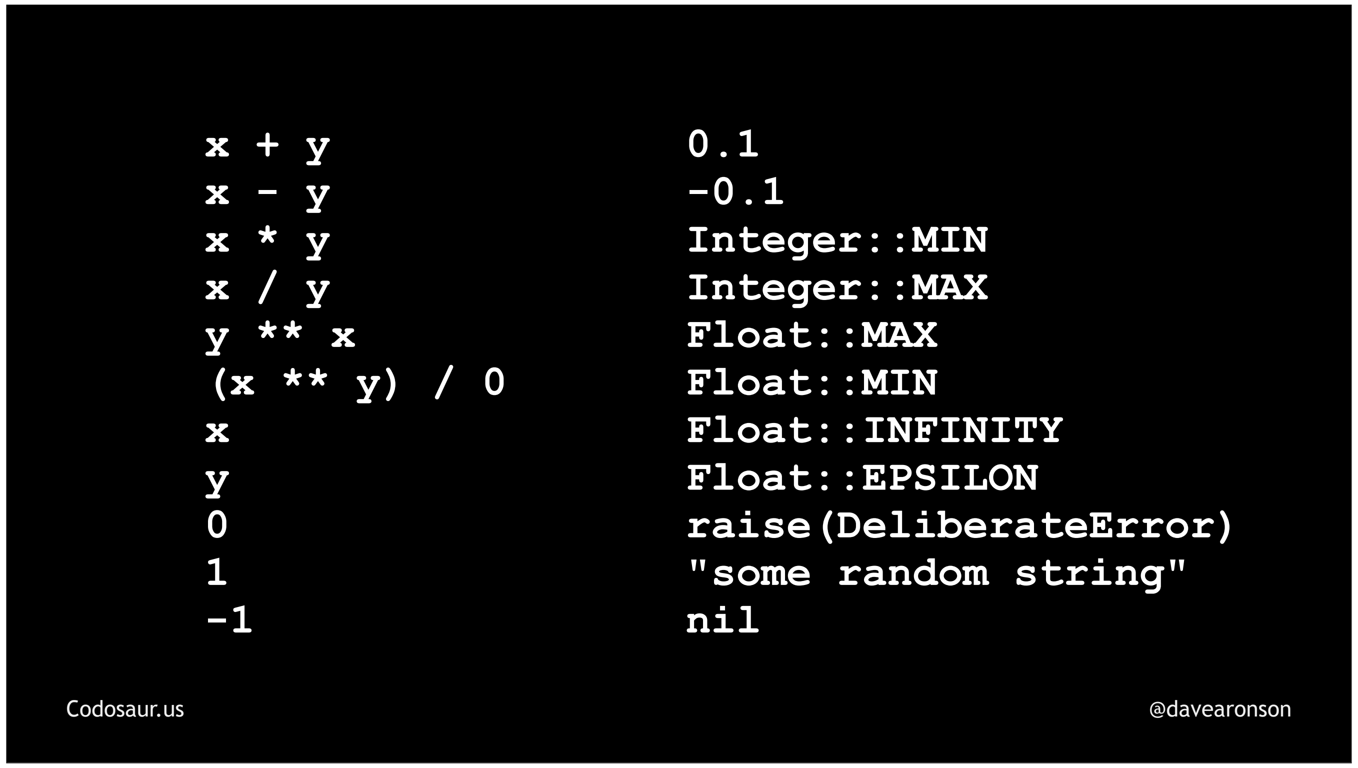
```
def power(x, y)
  x ** y
end
```

Codosaur.us @davearonson

. . . like so.

Think about what a mutant made from this might *return*, since that's what our tests would probably be looking at, as this doesn't have any side-effects.

Mainly it could return results such as . . .



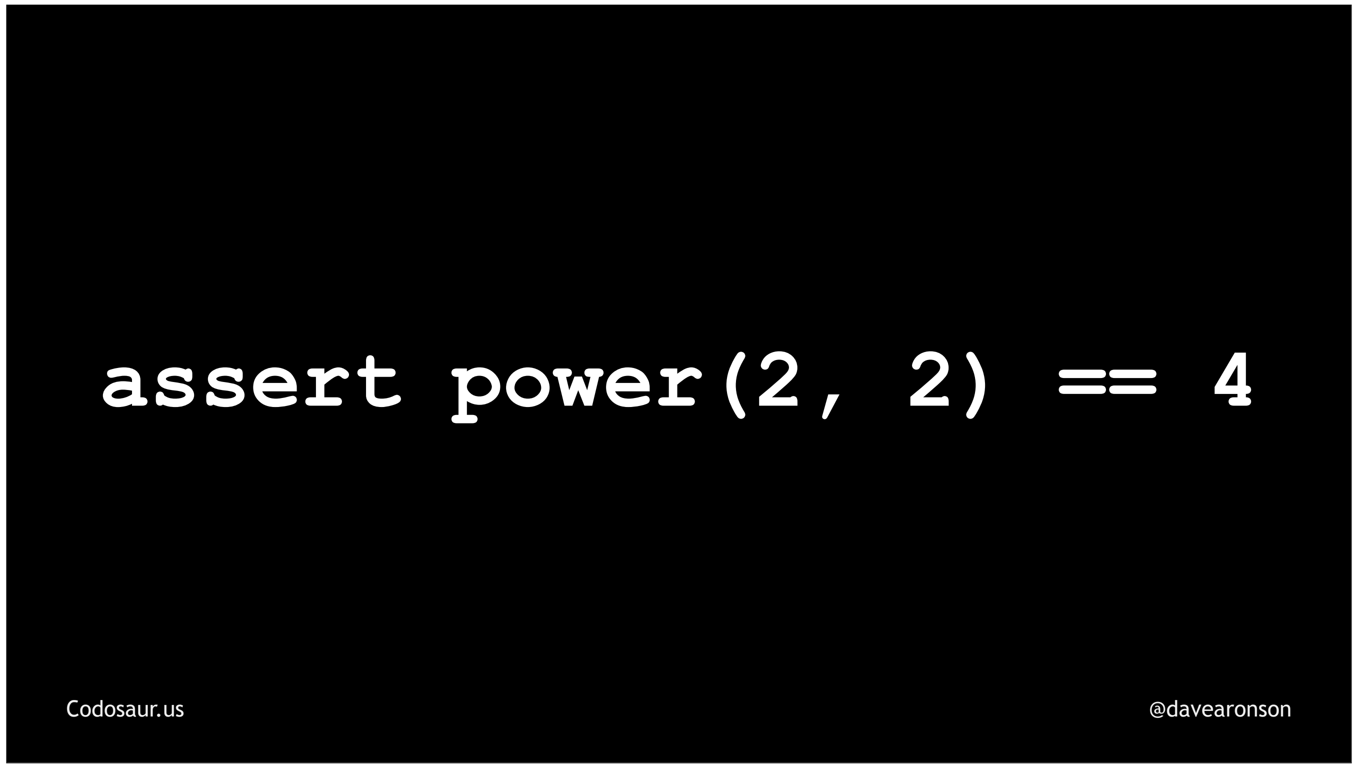
```
x + y              0.1
x - y              -0.1
x * y              Integer::MIN
x / y              Integer::MAX
y ** x             Float::MAX
(x ** y) / 0       Float::MIN
x                  Float::INFINITY
y                  Float::EPSILON
0                  raise(DeliberateError)
1                  "some random string"
-1                 nil
```

Codosaur.us

@davearonson

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

Now suppose we had only one test . . .

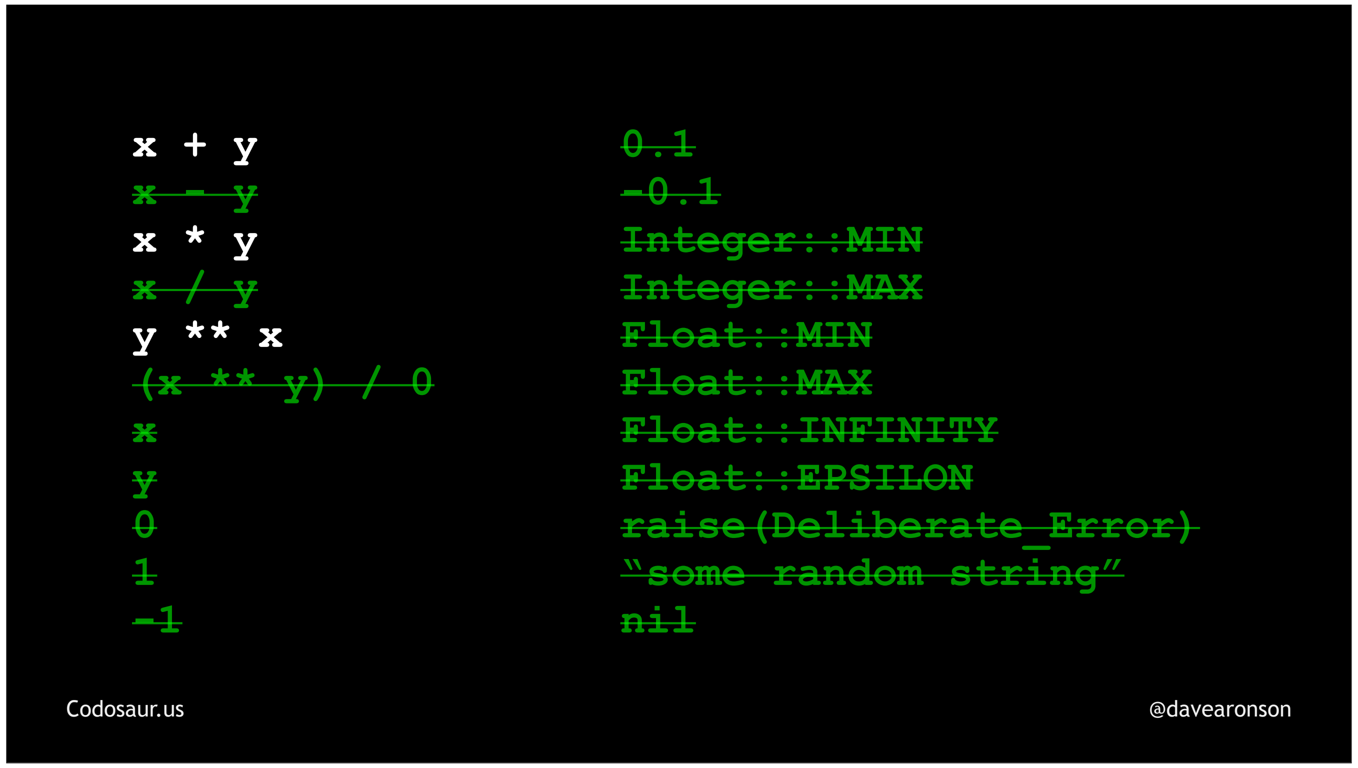


```
assert power(2, 2) == 4
```

Codosaur.us

@davearonson

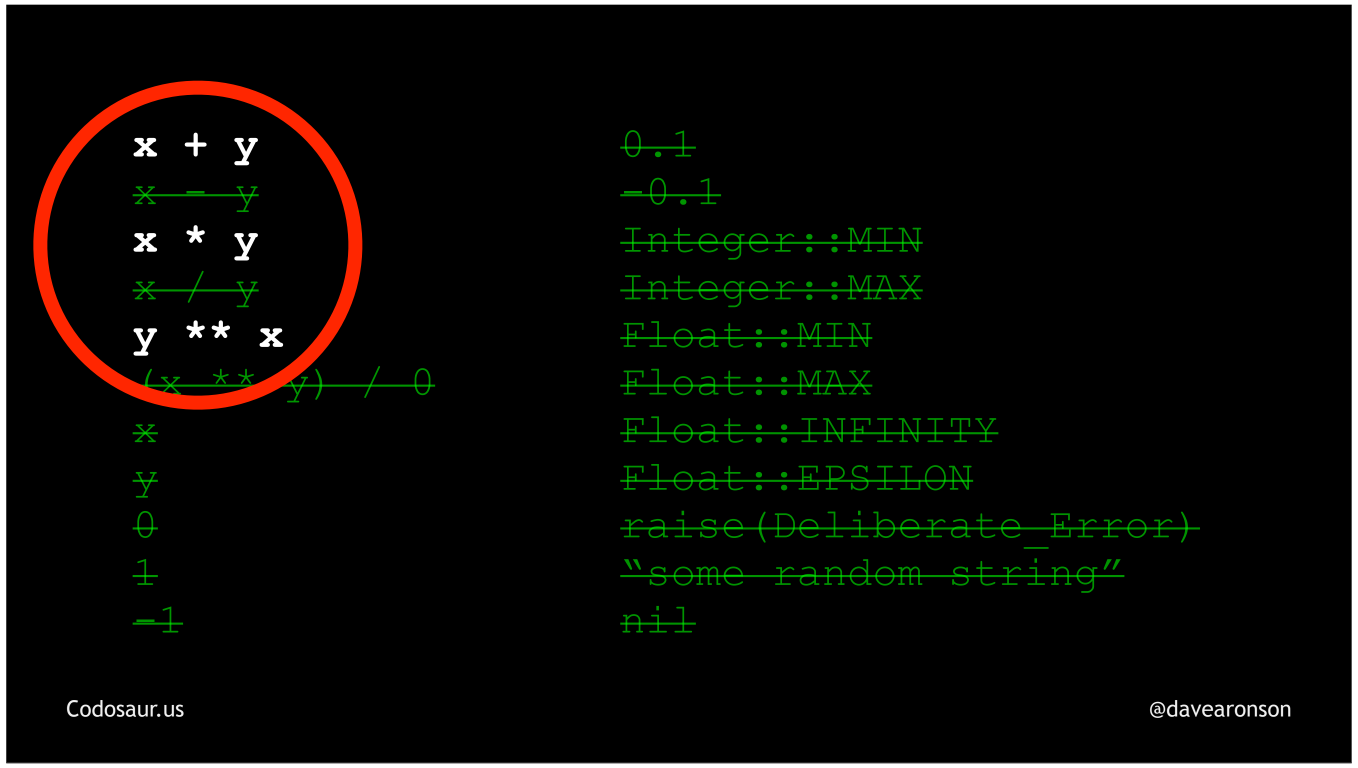
. . . like so. This is a rather poor test, and I think why is immediately obvious to most of you, but even so, *most* of those mutants on the previous slide *would get killed* by this test, the ones shown . . .



```
x + y                  ~~0.1~~
~~x - y~~                  ~~-0.1~~
x * y                  ~~Integer::MIN~~
~~x / y~~                  ~~Integer::MAX~~
y ** x                 ~~Float::MIN~~
~~(x ** y) / 0~~           ~~Float::MAX~~
~~x~~                      ~~Float::INFINITY~~
~~y~~                      ~~Float::EPSILON~~
~~0~~                      ~~raise(Deliberate_Error)~~
~~1~~                      ~~“some random string”~~
~~-1~~                     ~~nil~~
```

Codosaur.us @davearonson

. . . 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, and other significant numbers. Subtracting gets us zero, dividing gets us one, returning either argument alone gets us two, and the error conditions will at *least* make the test not pass. But . . .



```
x + y
~~x - y~~
x * y
~~x / y~~
y ** x
~~(x ** y) / 0~~
~~x~~
~~y~~
~~0~~
~~1~~
~~-1~~
```

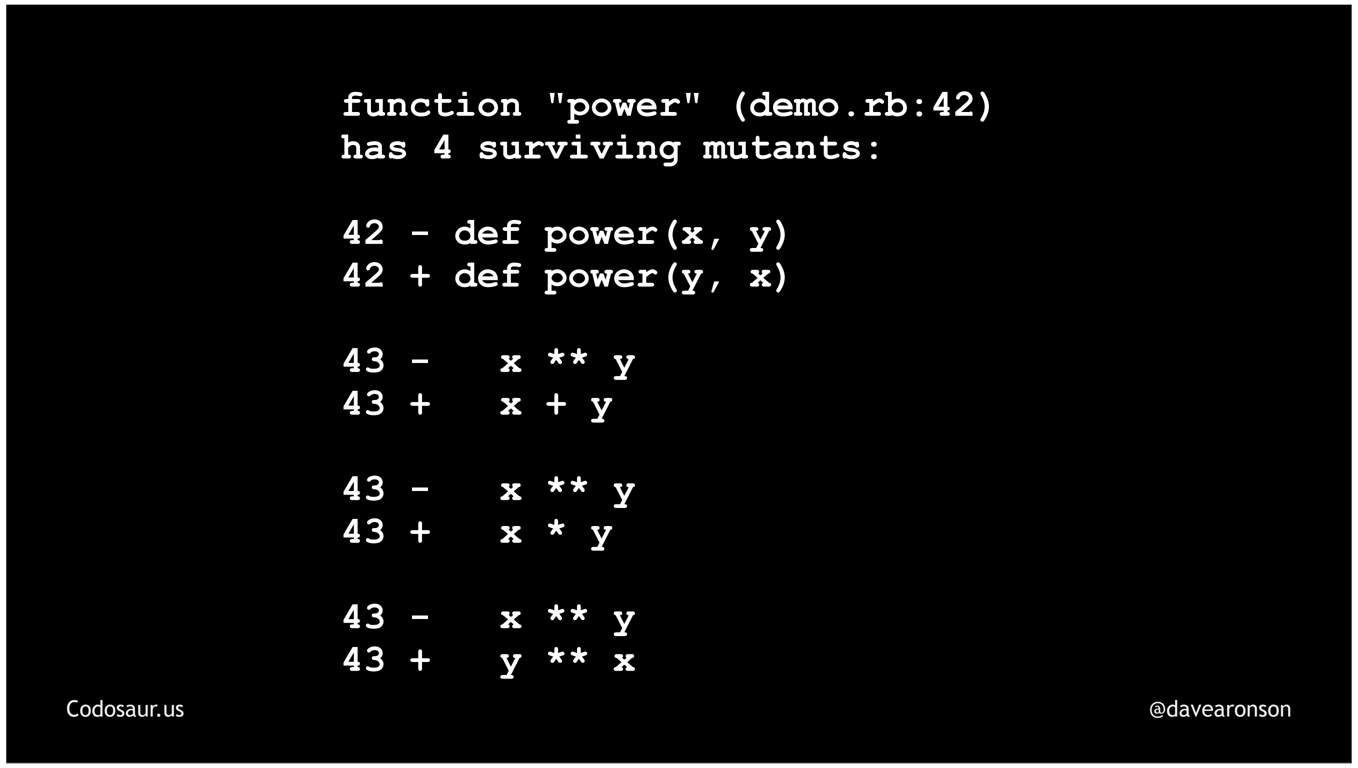
```
~~0.1~~
~~-0.1~~
~~Integer::MIN~~
~~Integer::MAX~~
~~Float::MIN~~
~~Float::MAX~~
~~Float::INFINITY~~
~~Float::EPSILON~~
~~raise(Deliberate_Error)~~
~~“some random string”~~
~~nil~~
```

Codosaur.us

@davearonson

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

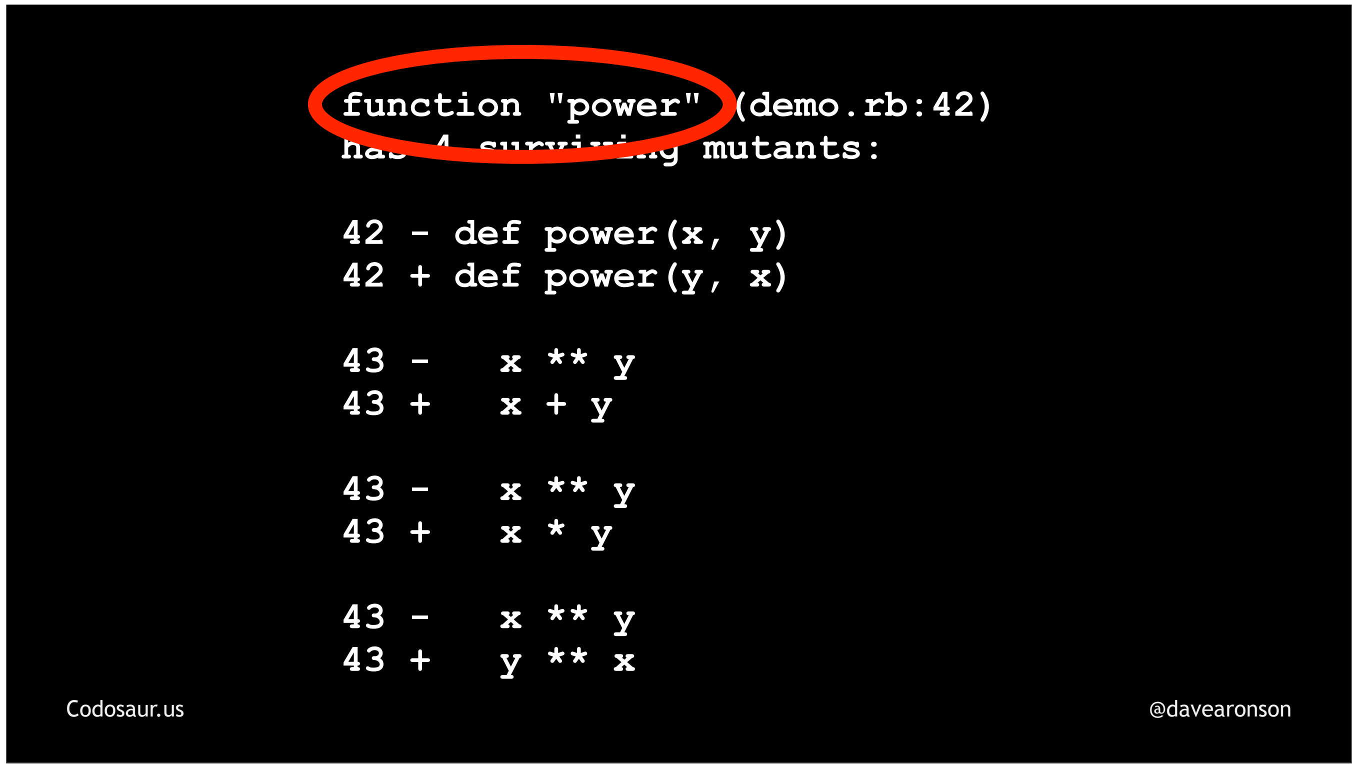
So how do we see that happening? When we run our tool, it gives us a report, that looks roughly like . . .



```
function "power" (demo.rb: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
```

. . . this. The exact words, format, amount of context, etc., will depend on exactly which tool we use, but the information should be pretty much the same. The minus or plus signs to the left of the actual code, denote lines removed or added, so together they mean a change. You may have seen this notation in some diff tools. Others may use less than and greater than signs, but the meaning is the same.

To fully unpack this, it's saying that if we changed . . .

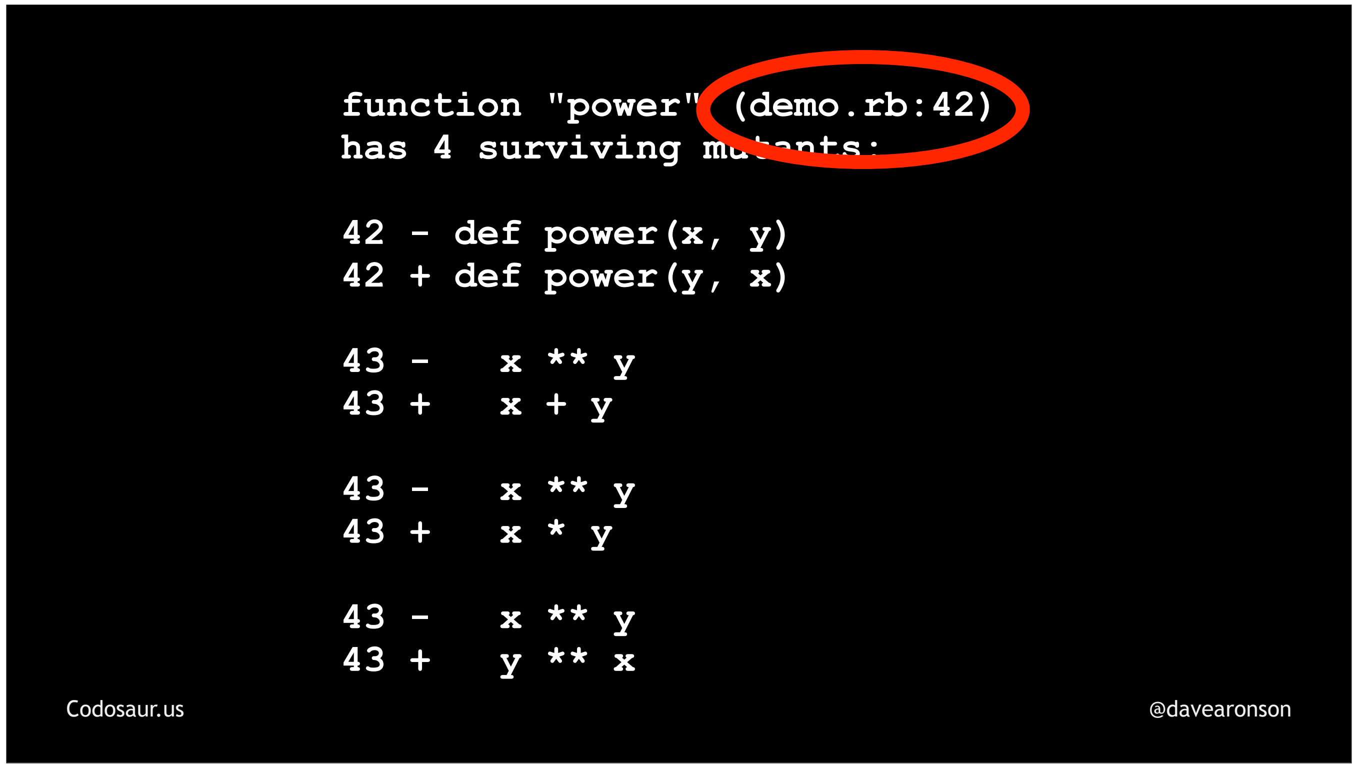


```
function "power" (demo.rb:42)
has 4 surviving mutants:

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, which is in . . .

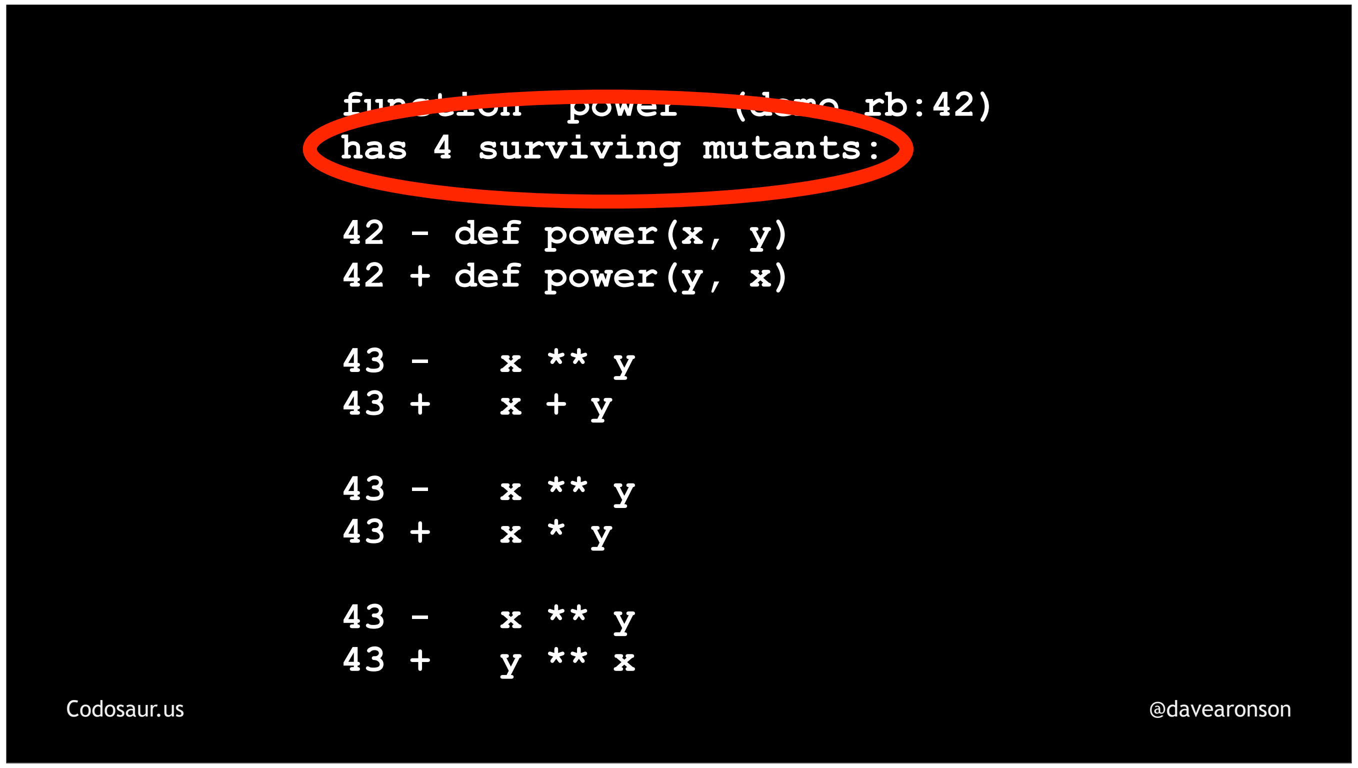


```
function "power" (demo.rb:42)
has 4 surviving mutants:

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.rb, and starts at line 42 . . .

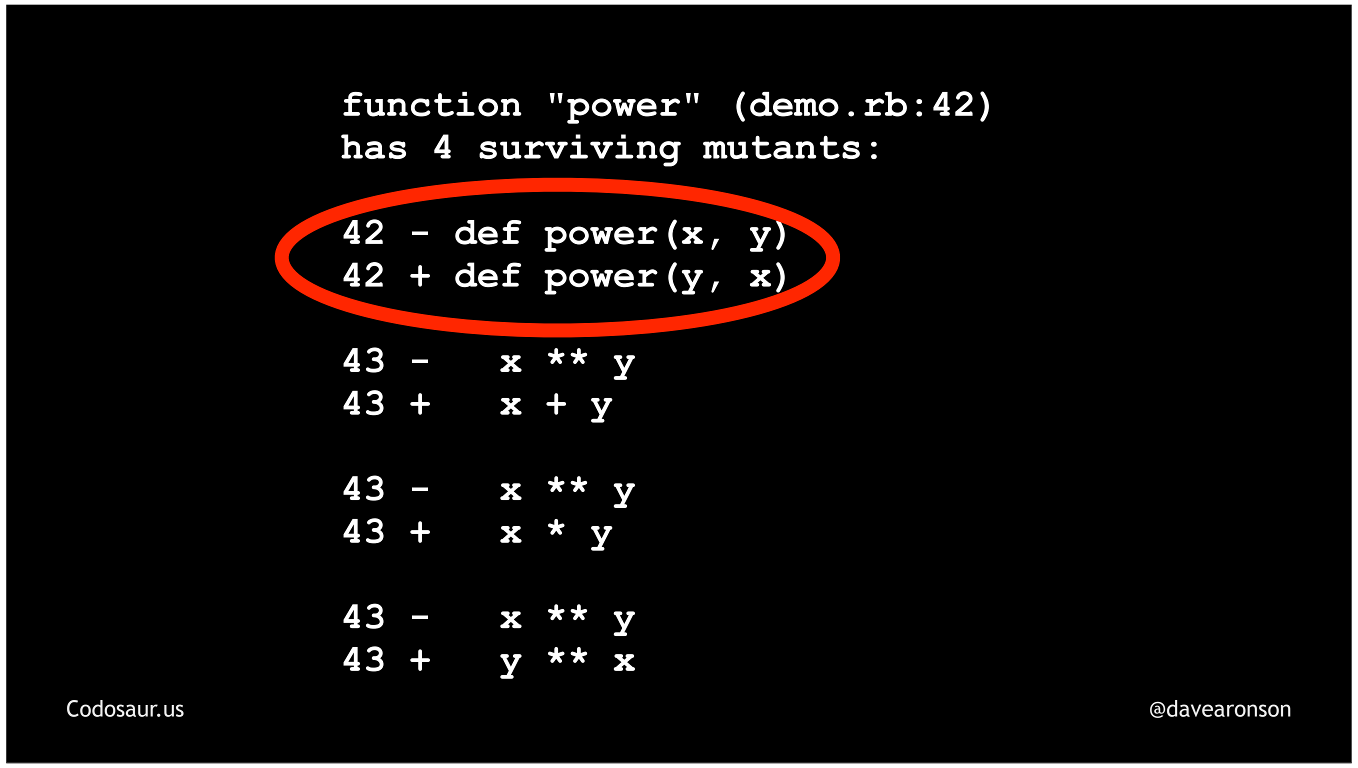


```
function power (demo.rb:42)
has 4 surviving mutants:

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 different ways, then all its tests would still pass, and those four ways are: . . .

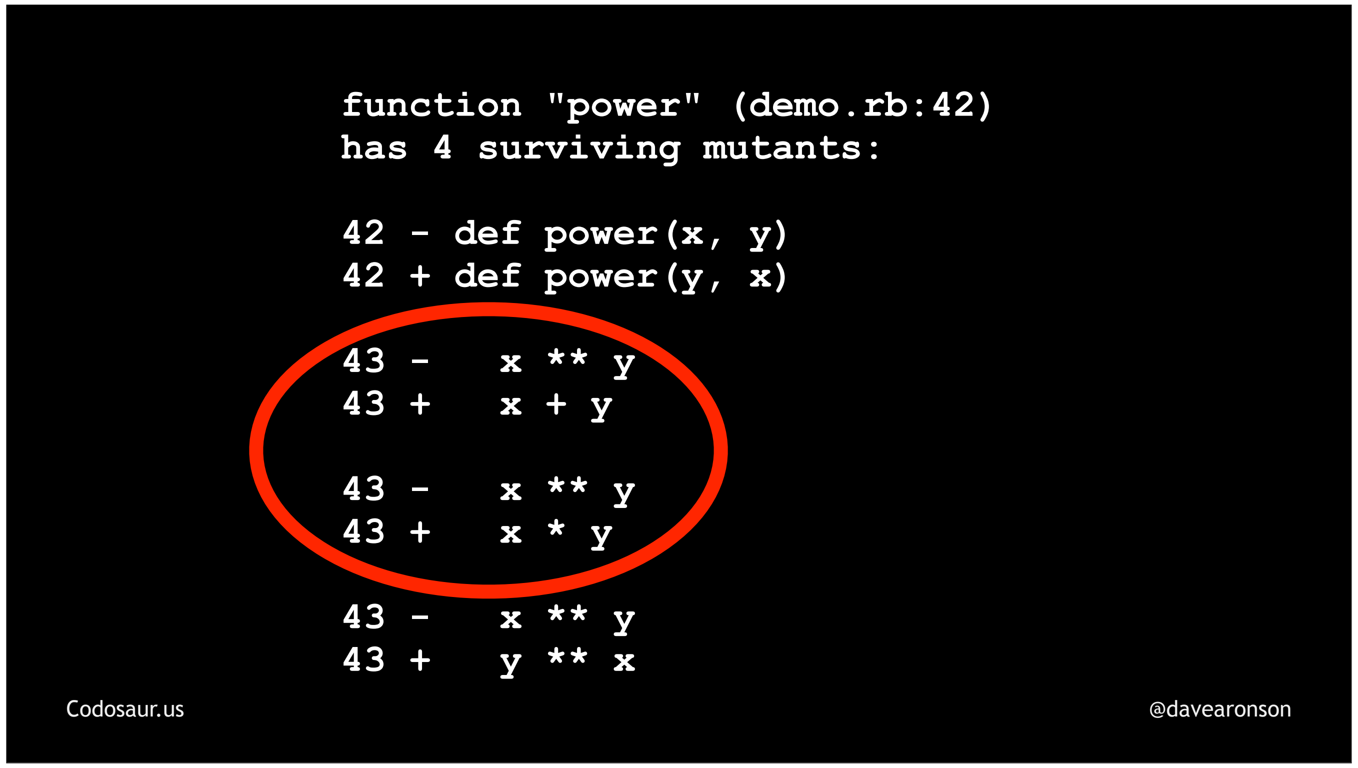


```
function "power" (demo.rb:42)
has 4 surviving mutants:

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

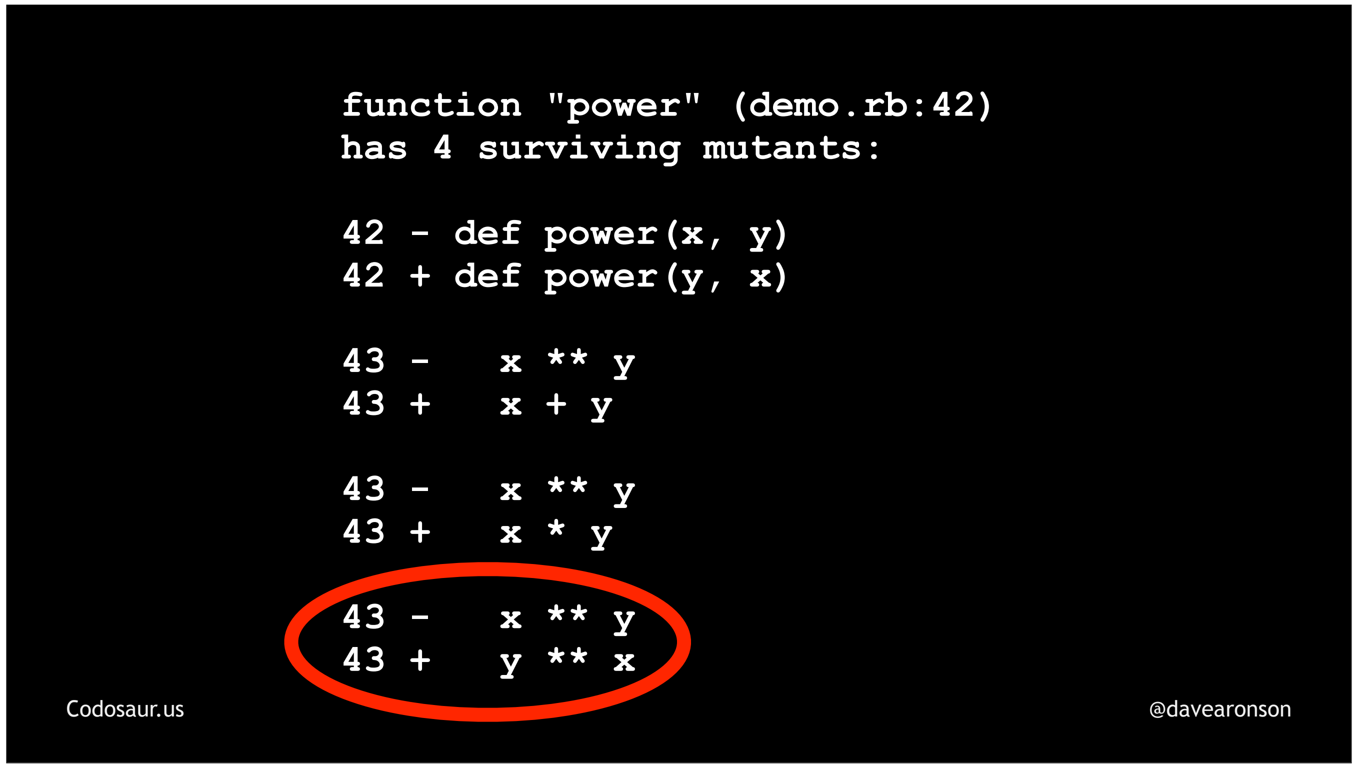
. . . to change line 42 to swap the arguments, or . . .



```
function "power" (demo.rb: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 line 43 to change the exponentiation into addition or multiplication, or . . .



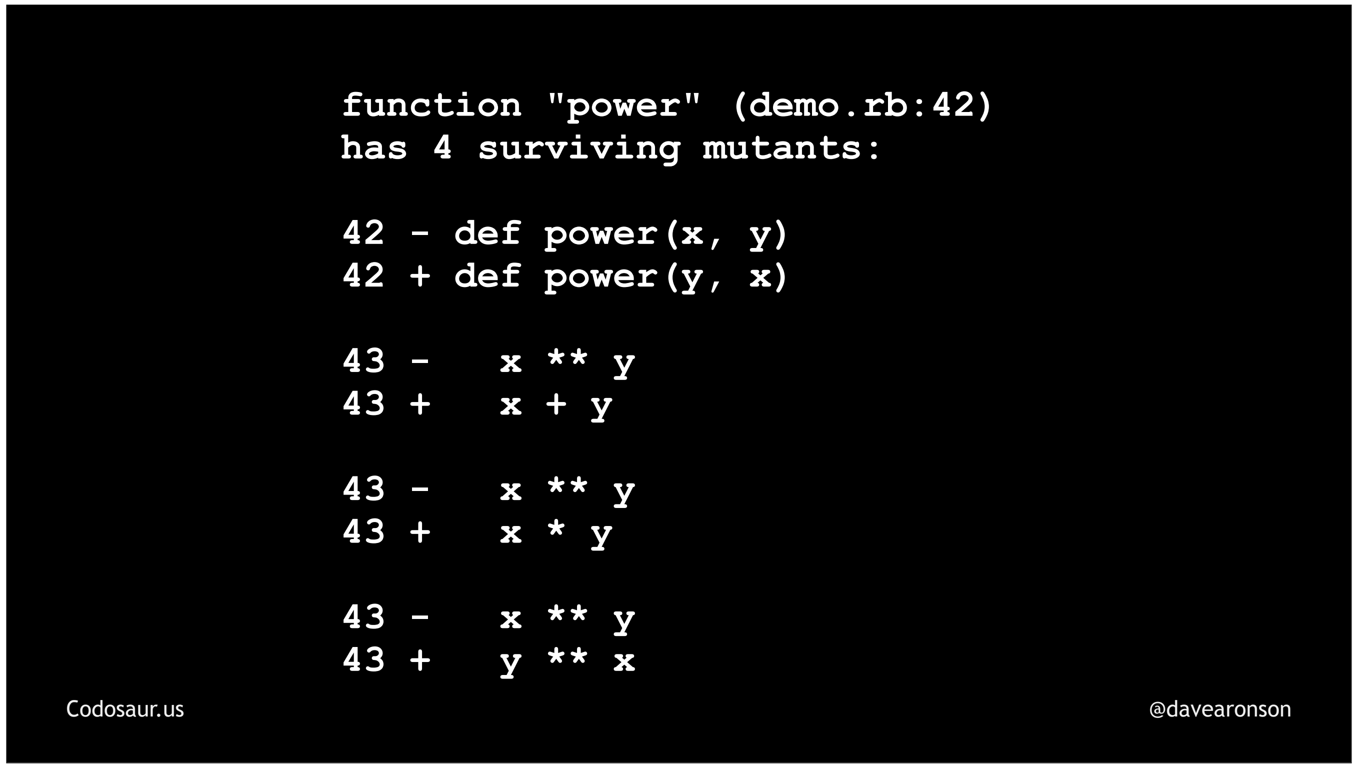
```
function "power" (demo.rb:42)
has 4 surviving mutants:

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 line 43 to to swap the operands.

So what is . . .

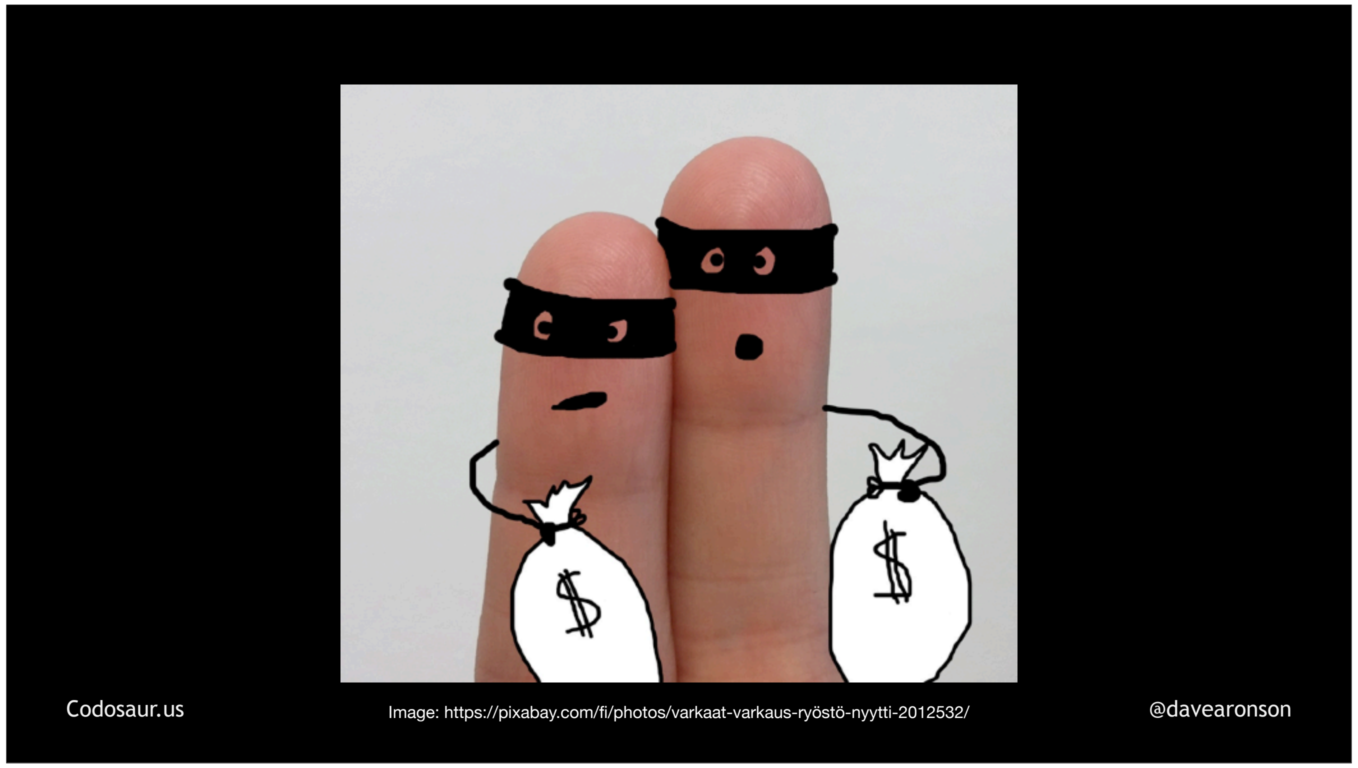


```
function "power" (demo.rb:42)
has 4 surviving mutants:

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? The very high level message is that our test suite is not sufficient, either because there aren't enough tests, or the ones we have just aren't very good, or both. But we knew that!

The question boils down to, how are these mutants surviving? Are they . . .



Codosaur.us

Image: https://pixabay.com/fi/photos/varkaat-varkaus-ryöstö-nyytti-2012532/

@davearonson

. . . pulling heists? Are they living at the . . .

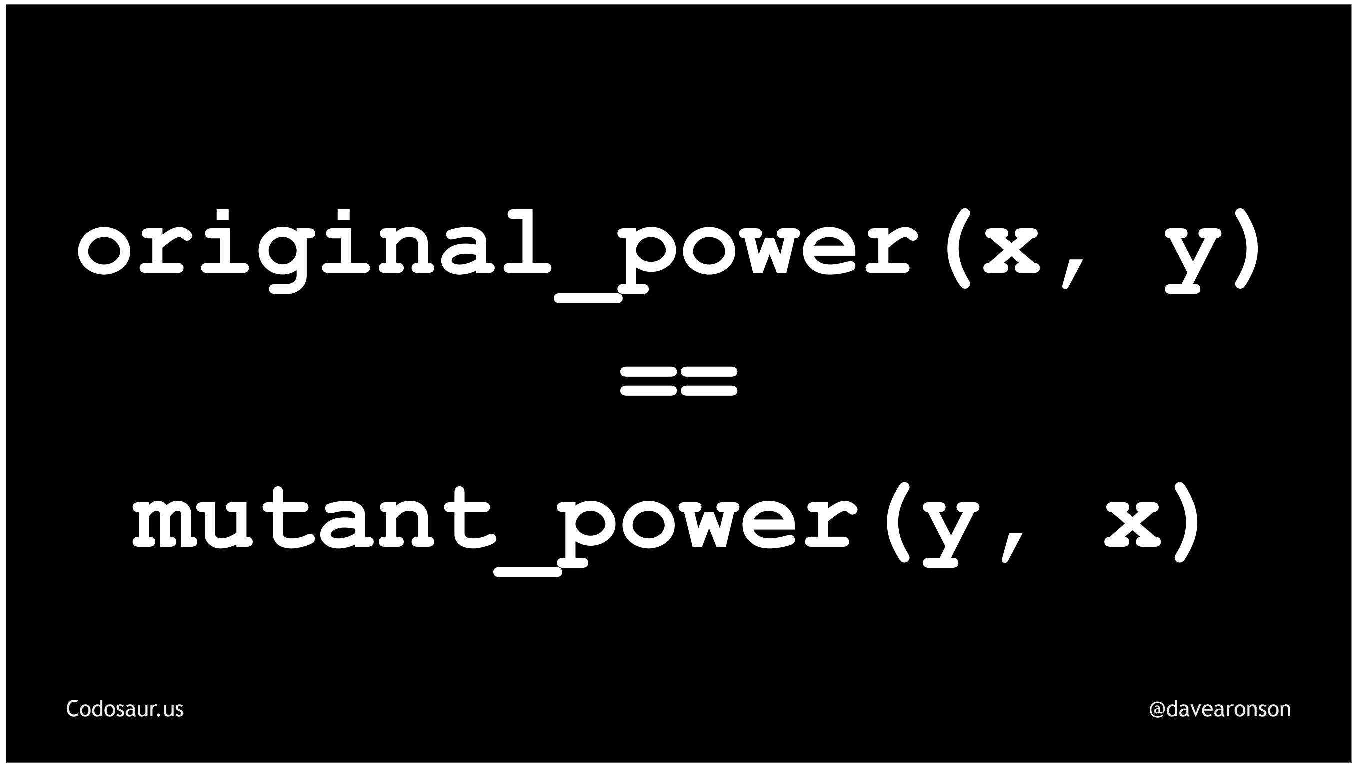
Codosaur.us

Image: https://pixabay.com/fi/photos/varkaat-varkaus-ryöstö-nyytti-2012532/

@davearonson

. . . Xavier Institute? Or what?

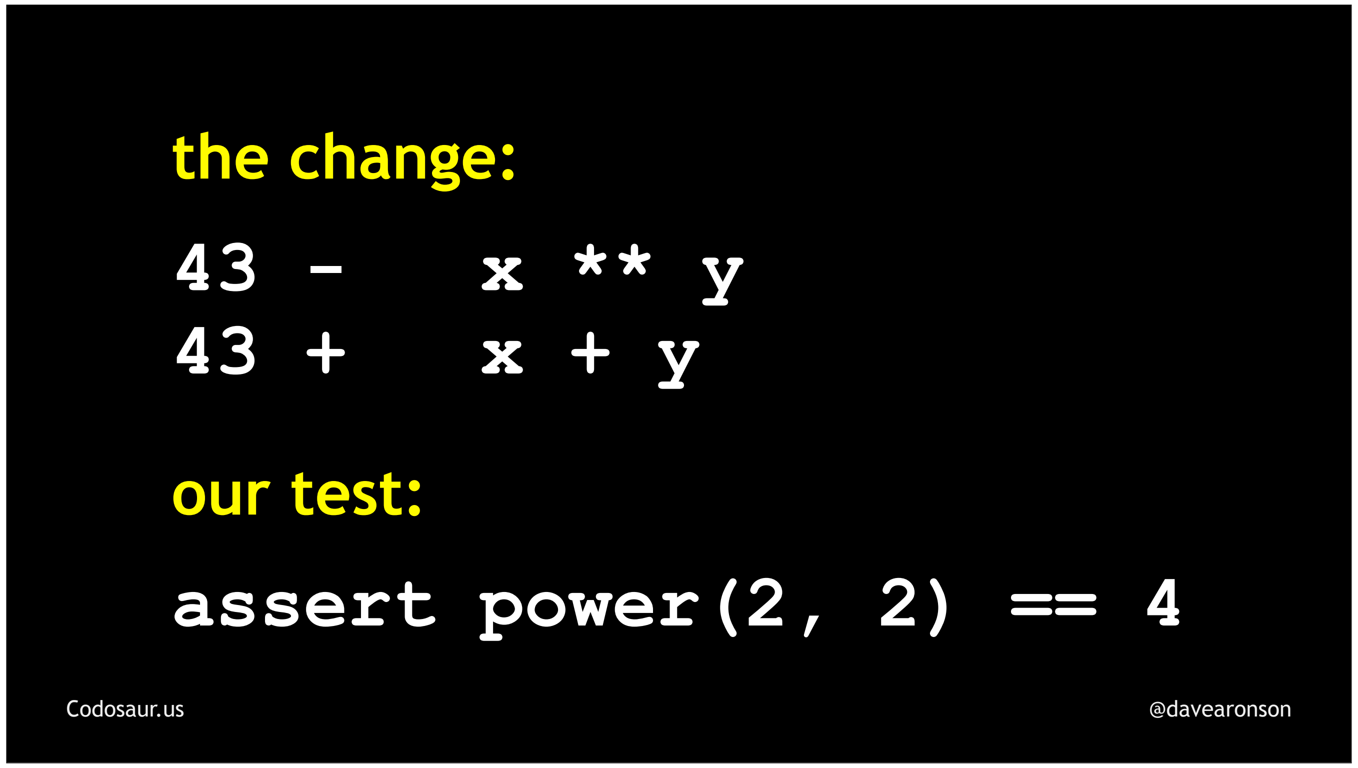
The usual answer is that . . .



```
original_power(x, y)
==
mutant_power(y, x)
```

Codosaur.us @davearonson

. . . they give the same result as the original function. Or if it's not a "pure function", maybe they have the same side effect — whatever it is that our tests are looking at. To determine how *that* happens, we can take a closer look, at one mutant, and a test it passes. Let's start with . . .



the change:

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

our test:

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

Codosaur.us

@davearonson

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

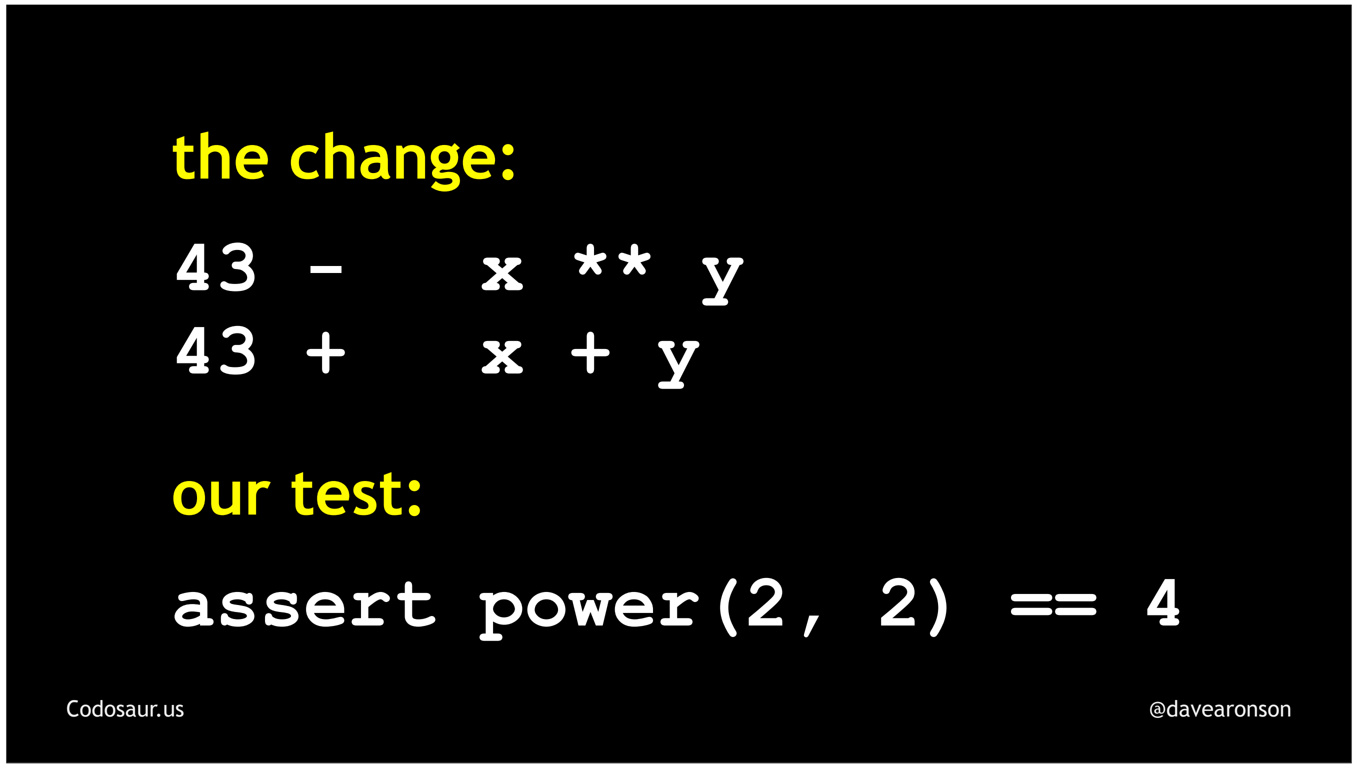
Codosaur.us

Image: meme going around, original source unfindable, sorry

@davearonson

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

So how can we *kill* . . .



the change:

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

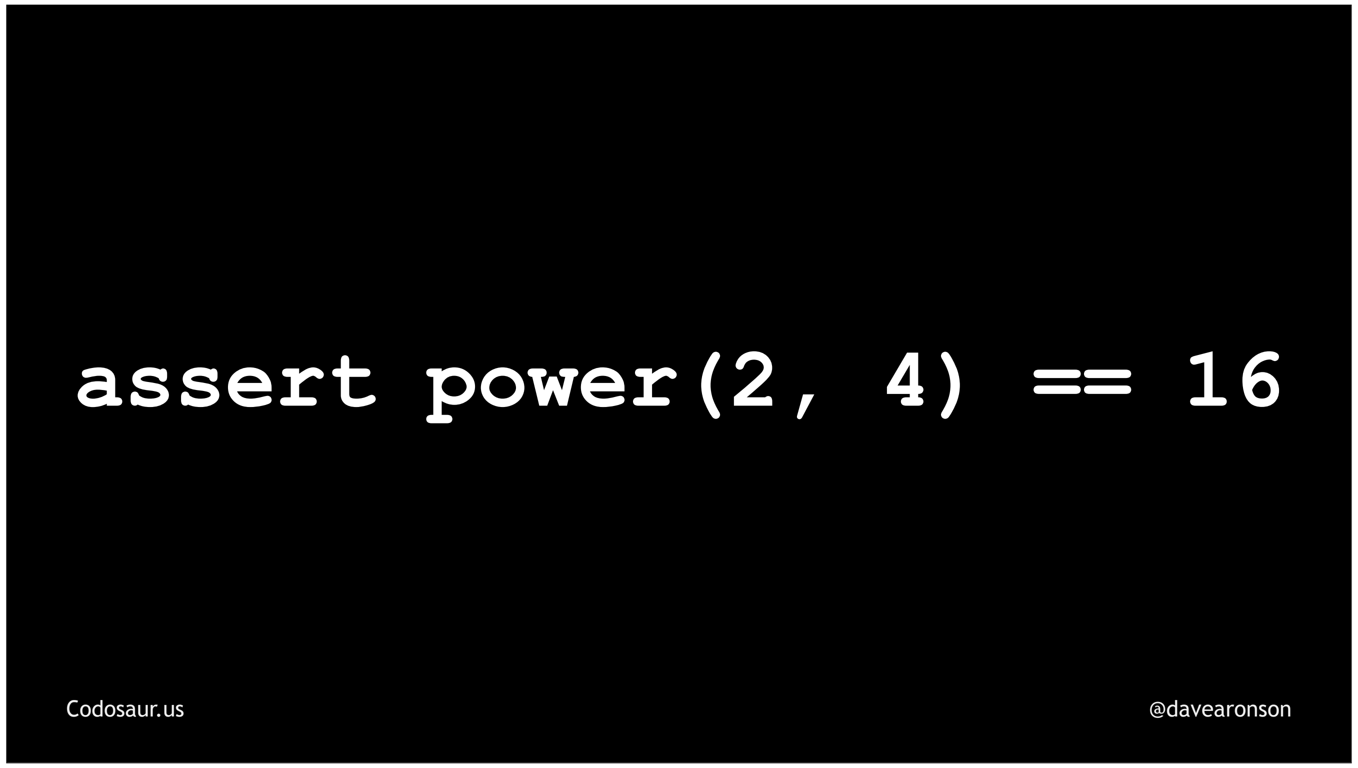
our test:

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

Codosaur.us

@davearonson

. . . this mutant, in other words, make it return a different answer, than the original code, given some set of inputs? It's quite simple in this case. We need to make at least one test use arguments such that x *to the* y is different from x *plus* y. For instance, we could add a test or change our test to . . .



```
assert power(2, 4) == 16
```

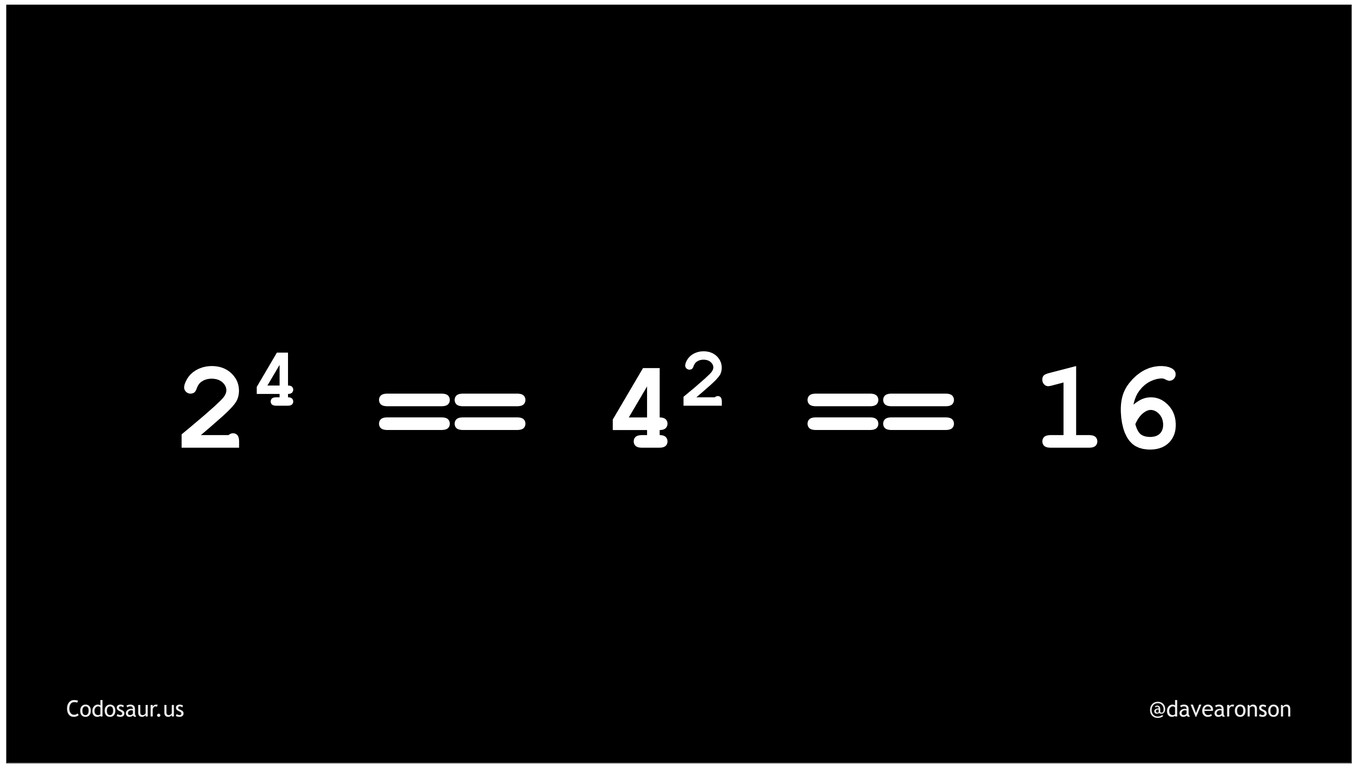
Codosaur.us @davearonson

. . . assert that two to the *fourth* power is *sixteen*. All the mutants that our original test killed, this would still kill. Two *plus* four is six, not six*teen*, so this should kill the plus mutant just fine. For that matter, two *times* four is eight, which is *also* not sixteen, so this should kill the "times" mutant as well.

However, . . .

Codosaur.us
Image: https://pixabay.com/vectors/teenage-mutant-ninja-turtles-turtle-151715/
@davearonson

. . . the (ahem) pair of argument-swapping mutants survive! How can that be? It's because . . .

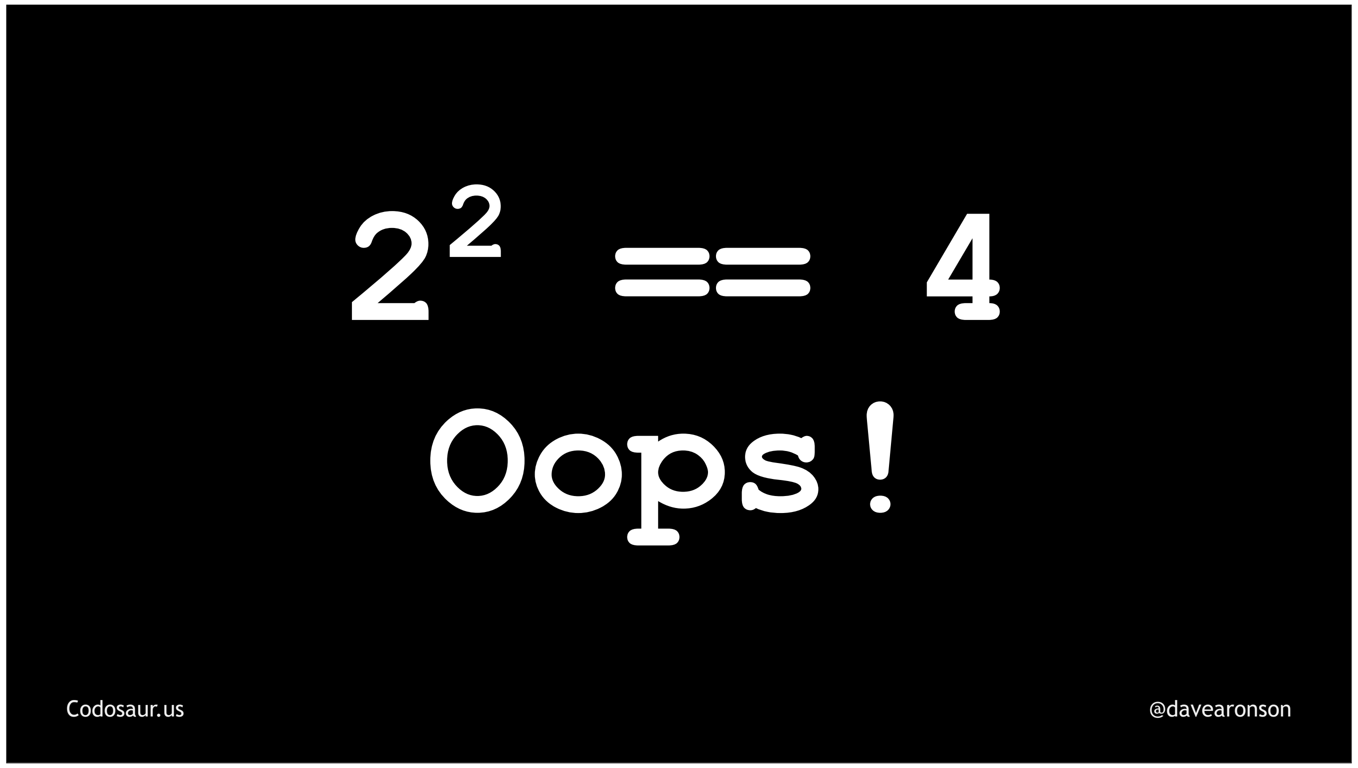


$$2^4 == 4^2 == 16$$

Codosaur.us

@davearonson

. . . two to the fourth power, and four to the second, are both sixteen! Since the function deals with powers, it probably would have been smarter to avoid argument pairs where . . .



2^2 == 4

Oops!

Codosaur.us

@davearonson

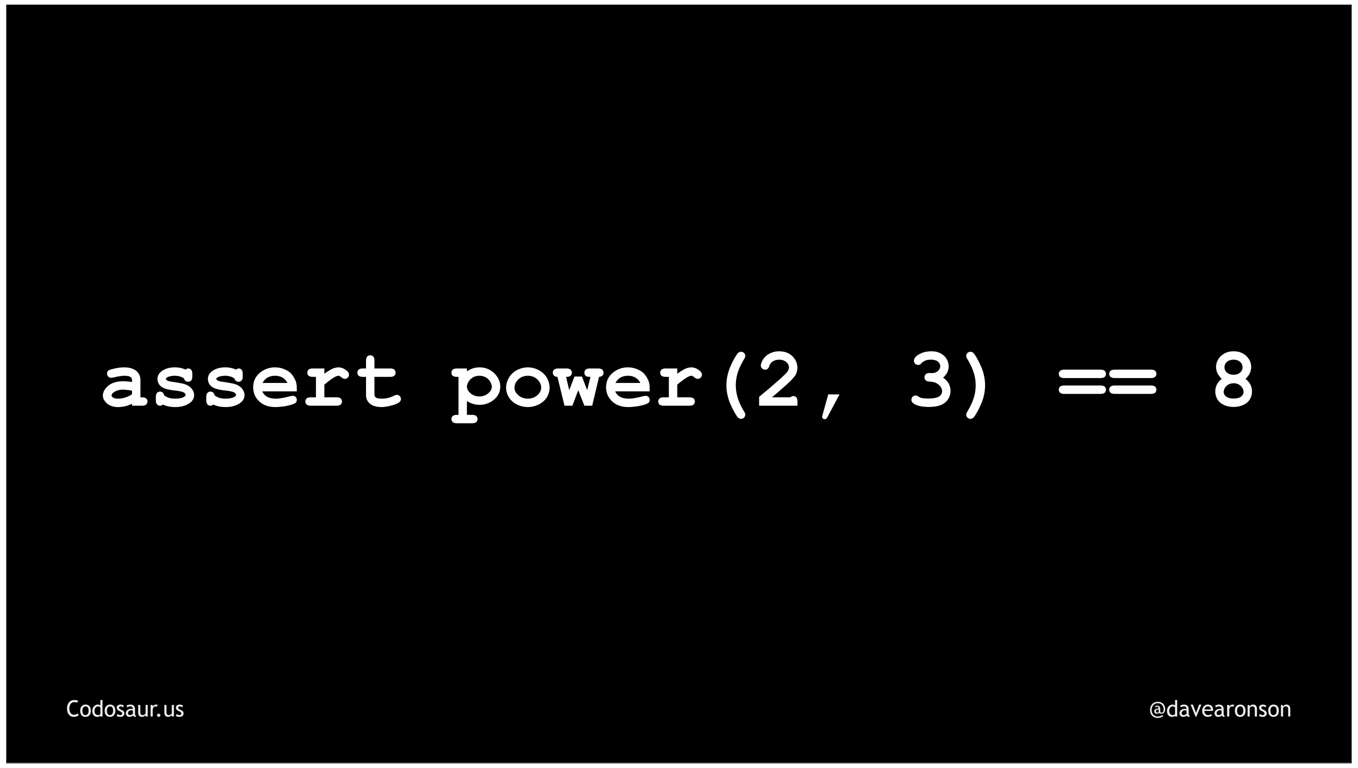
. . . one of them is a power of the other. But anyway, we can . . .

Codosaur.us

Image: https://publicdomainvectors.org/en/free-clipart/Superhero-in-red-uniform/45131.html

@davearonson

. . . attack the argument-swapping mutants separately, no need to kill all the mutants at once and be some kind of superhero about it. To do that, again, we can either add a test, or adjust an existing test, such as . . .



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

Codosaur.us

@davearonson

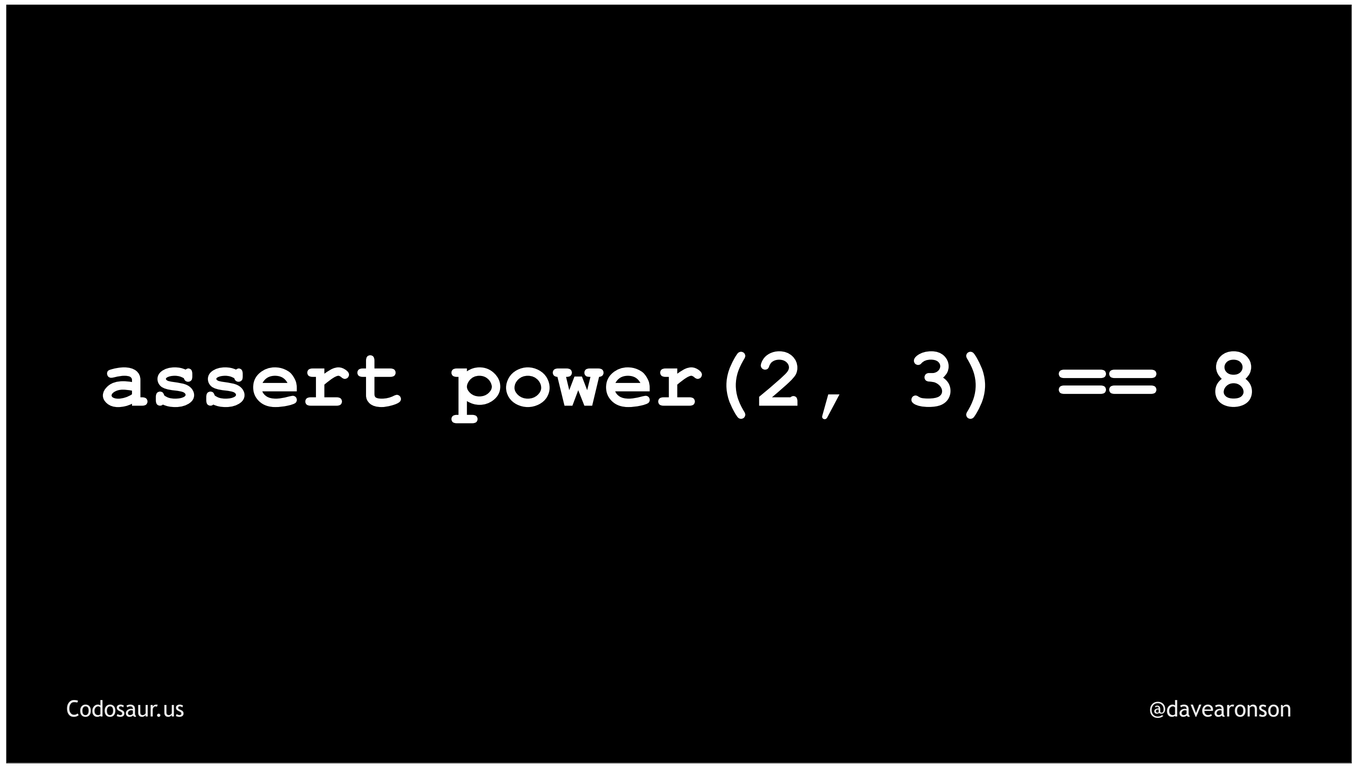
. . . this, to assert that two to the *third* power is *eight*.   Three squared is nine, not eight, so this kills the argument-swapping mutants. Two *plus* three is five, two *times* three is six, and both of those are, guess what, not eight, so the "plus" and "times" mutants *stay* dead, and we don't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

Codosaur.us

Image: https://pixabay.com/vectors/zombie-undead-monster-living-dead-156138/ (modified by me)

@davearonson

. . . zombie mutants wandering around. (PAUSE!) With . . .



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

Codosaur.us

@davearonson

. . . these inputs, the correct operation is the only simple common one that yields the correct answer. This isn't the *only* solution, though; we could have used two to the *fifth*, *three* squared, three to the fifth, vice-versa, and many many more. There are *lots* of ways to skin . . .

Codosaur.us

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

@davearonson

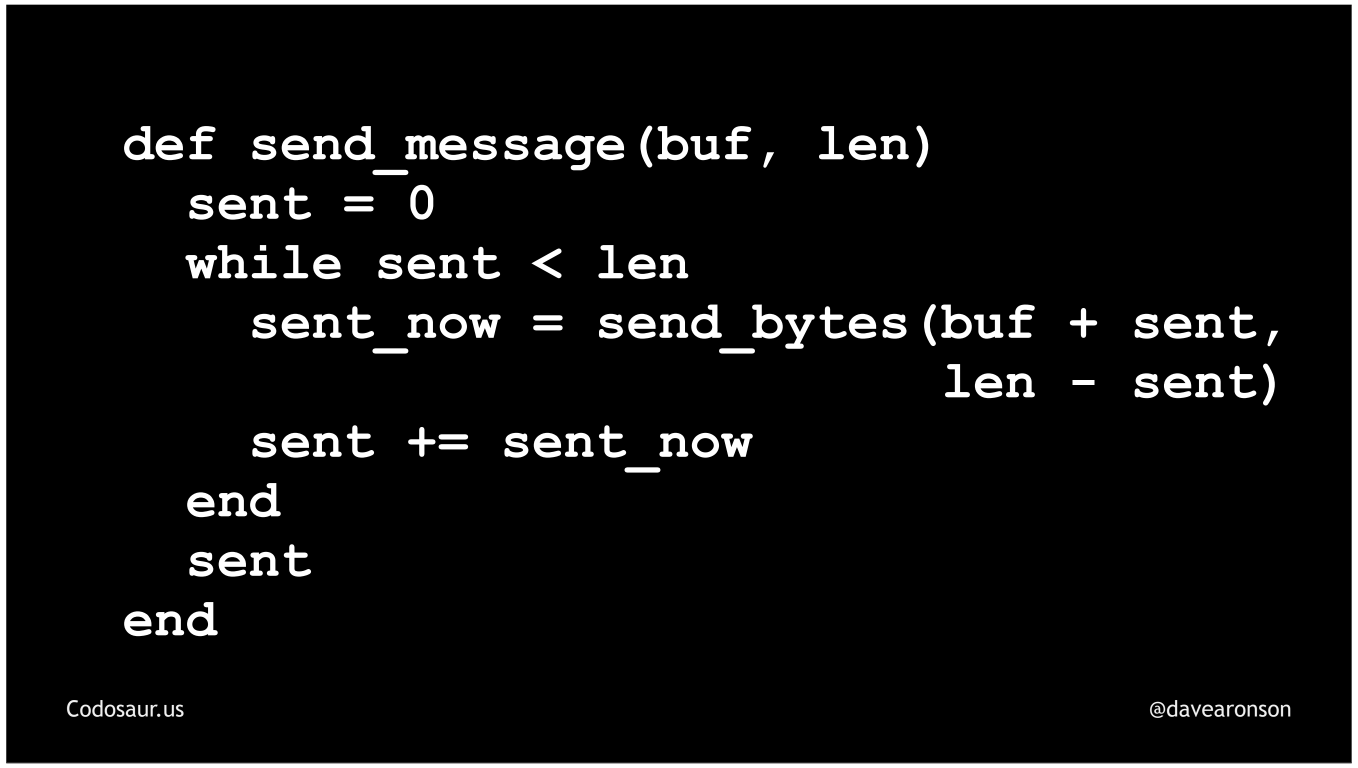
. . . *that* flerken!

This may make mutation testing sound . . .

SIMPLE SIMON
Codosaur.us
Image: https://commons.wikimedia.org/wiki/File:Simple_Simon_LCCN2003677693.jpg
@davearonson

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

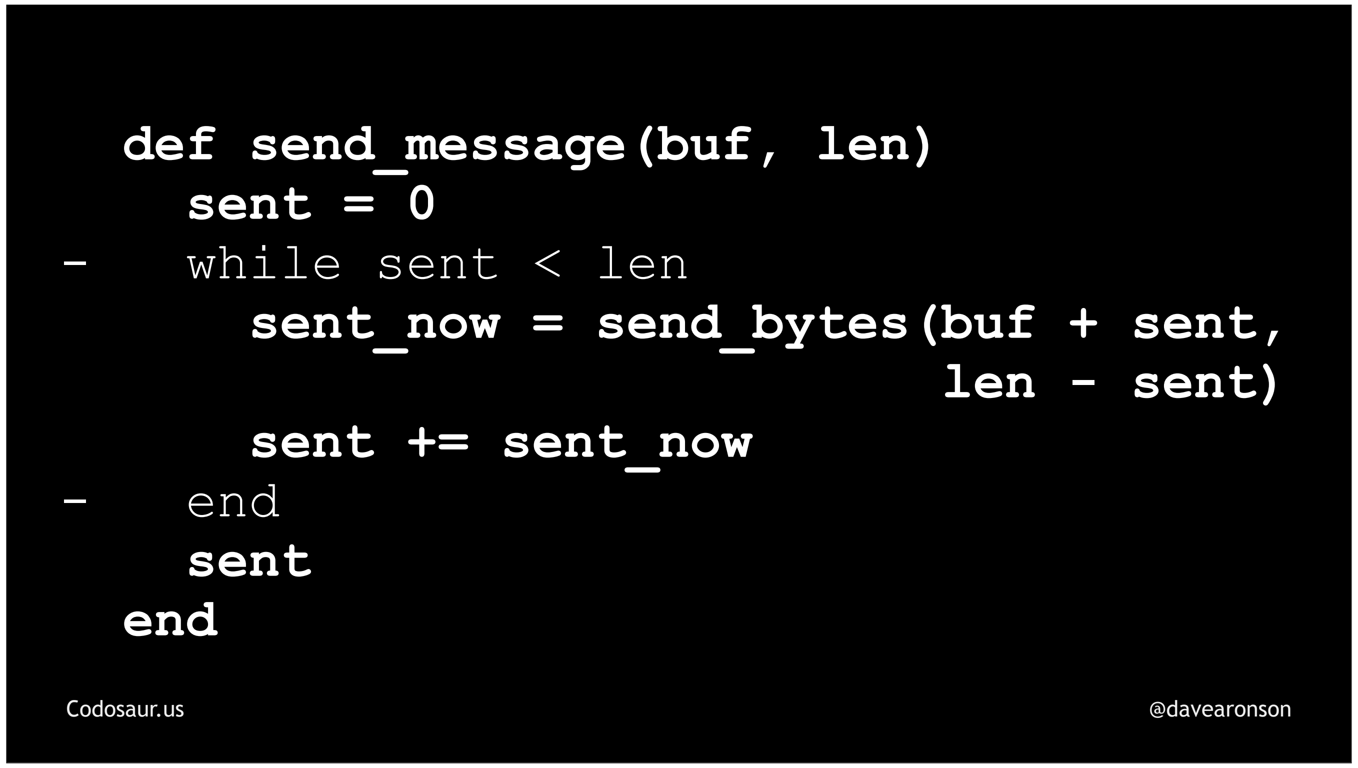
So let’s look at a more *complex* example! Suppose we have a function to send a message, . . .



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

. . . like so. This function, send_*message*, uses a loop that sends as much data as send_*bytes* can handle in one chunk, over and over, picking up where it left off, until the message is all sent. This is a very common pattern in communication software.

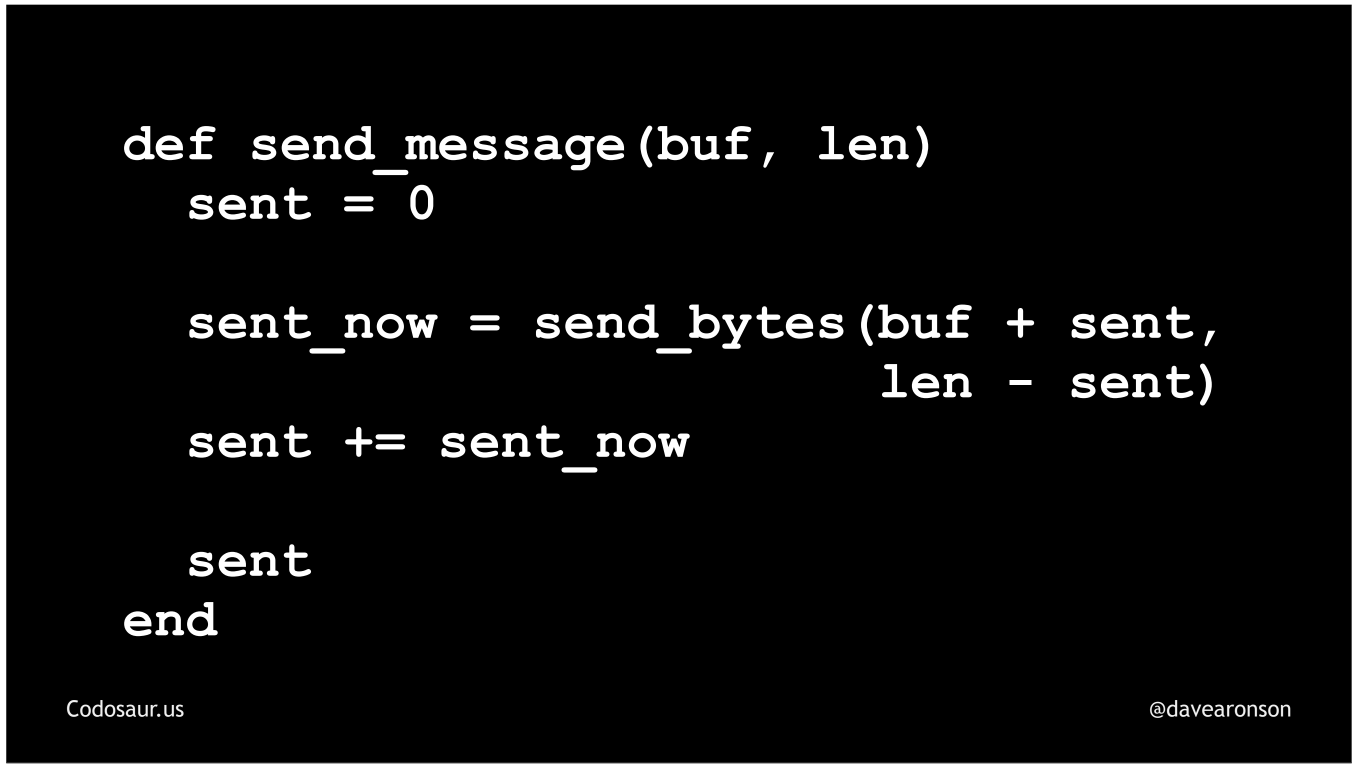
A mutation testing tool could make lots and lots of mutants from this, but one of them, of particular interest, would be . . .



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

Codosaur.us @davearonson

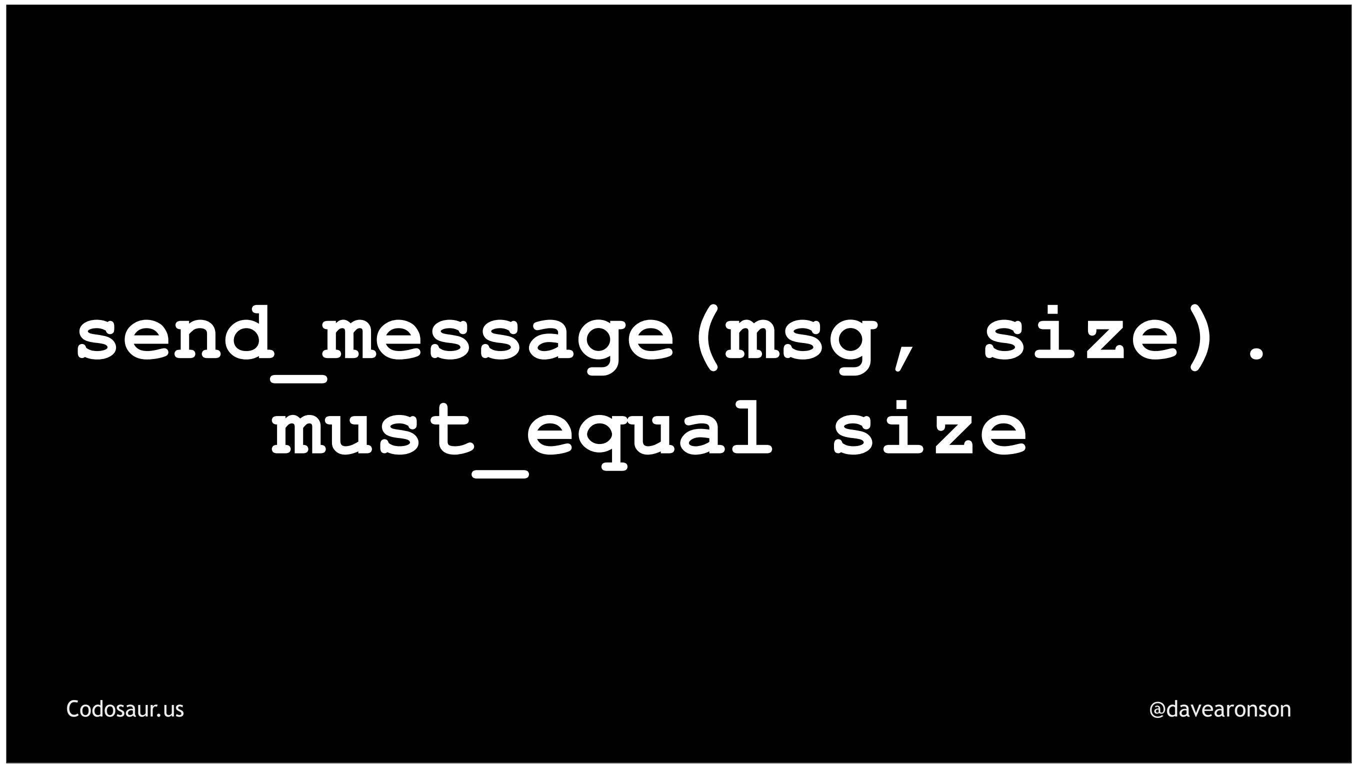
. . . this, removing those lines with the minus signs, an example of removing a loop control. That would make it effectively read like . . .



```
def send_message(buf, len)
  sent = 0

  sent_now = send_bytes(buf + sent,
                        len - sent)
  sent += sent_now

  sent
end
```

Codosaur.us @davearonson

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



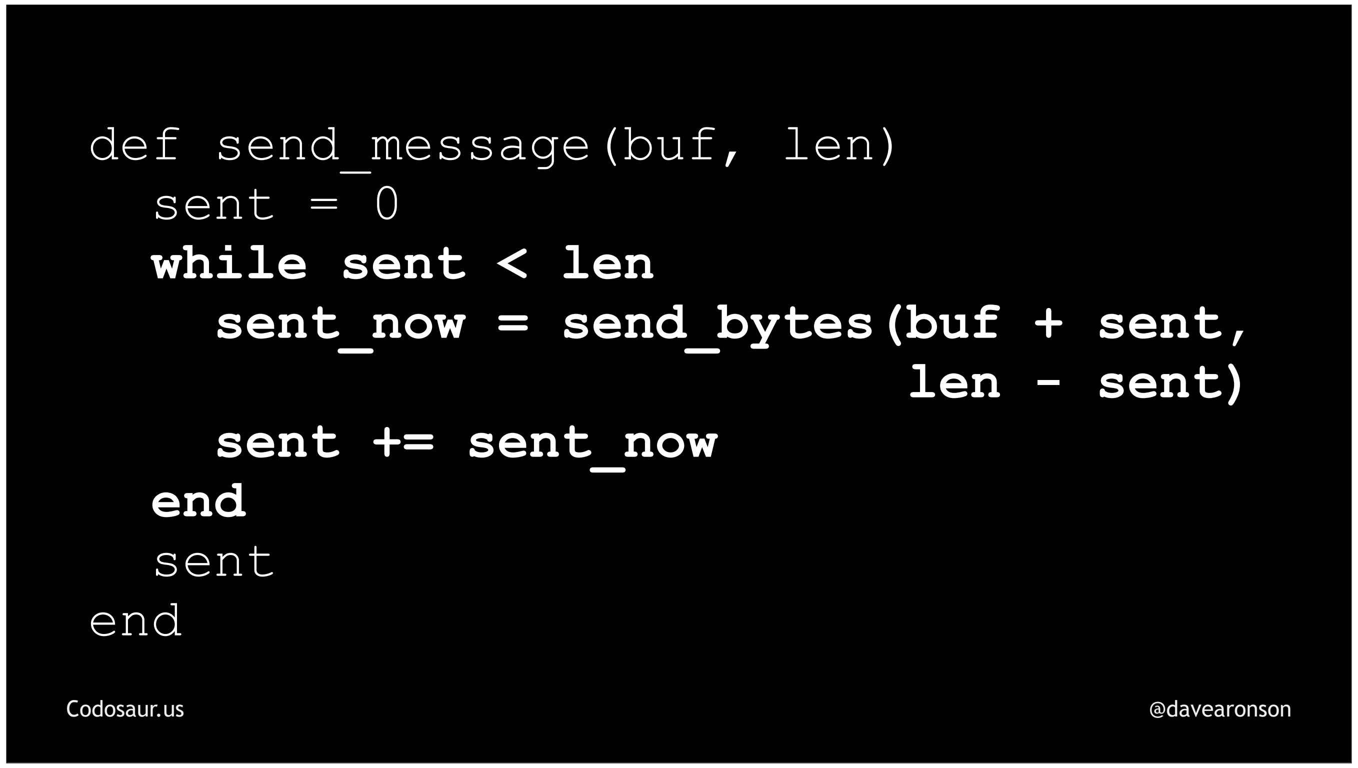
```
send_message(msg, size).
    must_equal size
```

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 creating the message. Even without seeing that test code though, what does the survival of that non-looping mutant tell us? (PAUSE!)

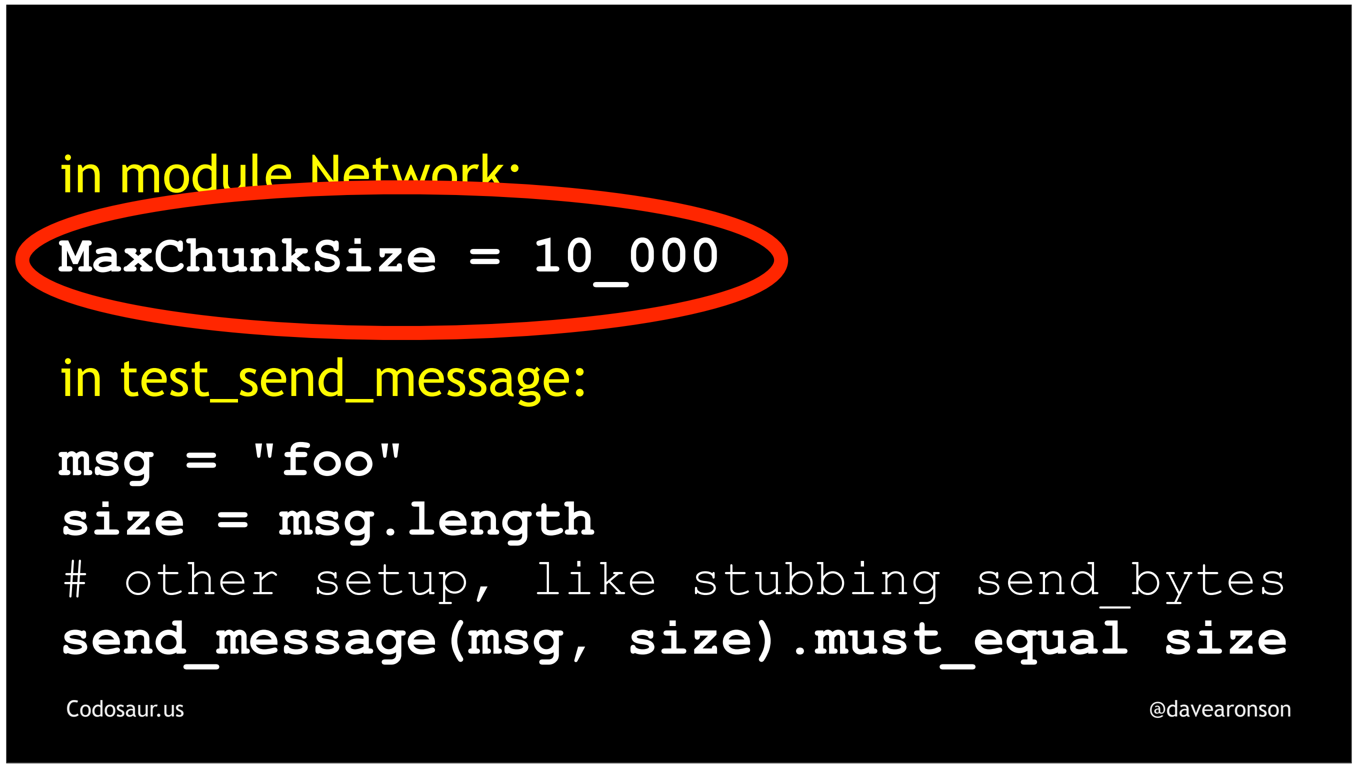
If a mutant that only goes through . . .



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

. . . that while-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 code go through that while-loop once. So what does that mean? (PAUSE!) By the way, you'll find that interpreting mutants involves a lot of asking *"so what does that mean"*, often recursively!

In this case, it means that we're not testing sending a message larger than send_*bytes* can handle in one chunk! The most likely cause of *that*, is that we simply didn't test with a big enough message. For instance, . . .



in module Network:

```
MaxChunkSize = 10_000
```

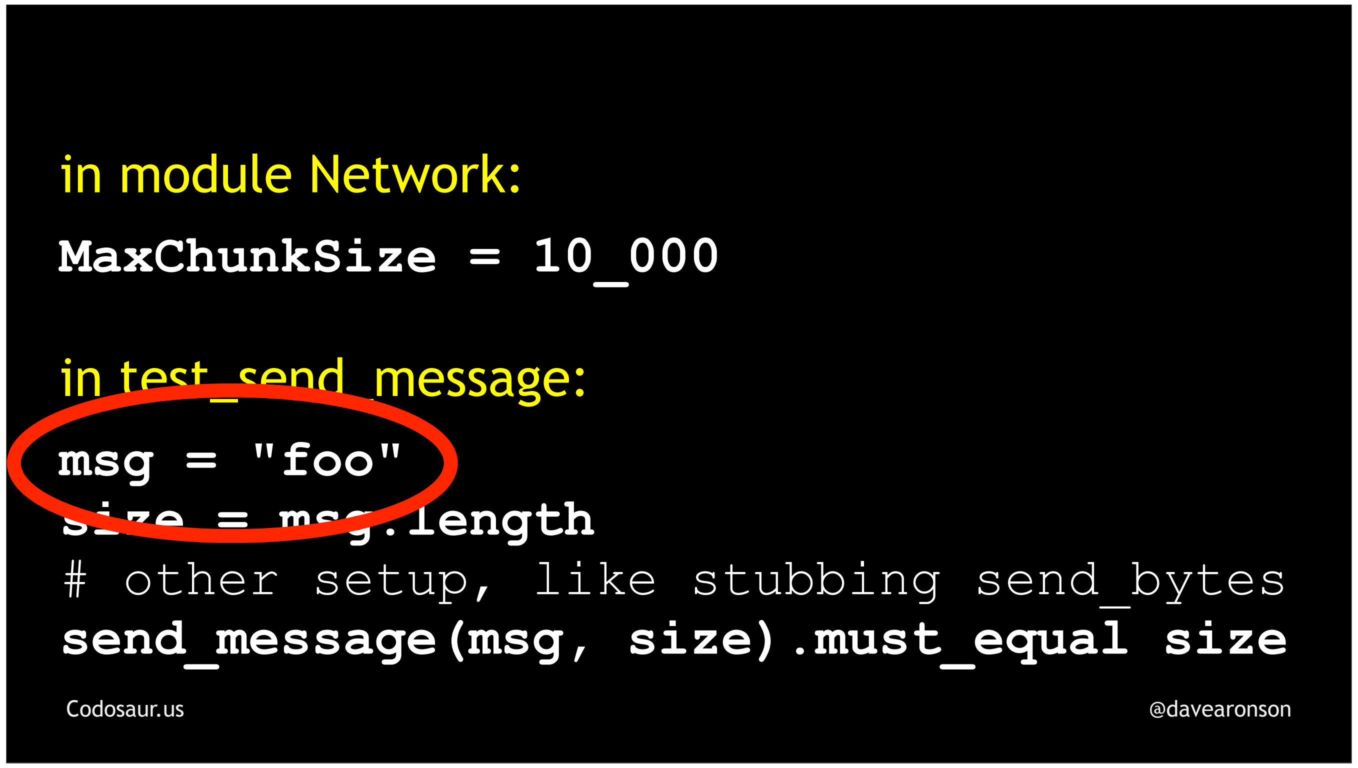
in test_send_message:

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

Codosaur.us

@davearonson

. . . suppose our maximum chunk size, what send_*bytes* can handle in one chunk, is 10,000 bytes. However, for whatever reason, . . .



in module Network:

```
MaxChunkSize = 10_000
```

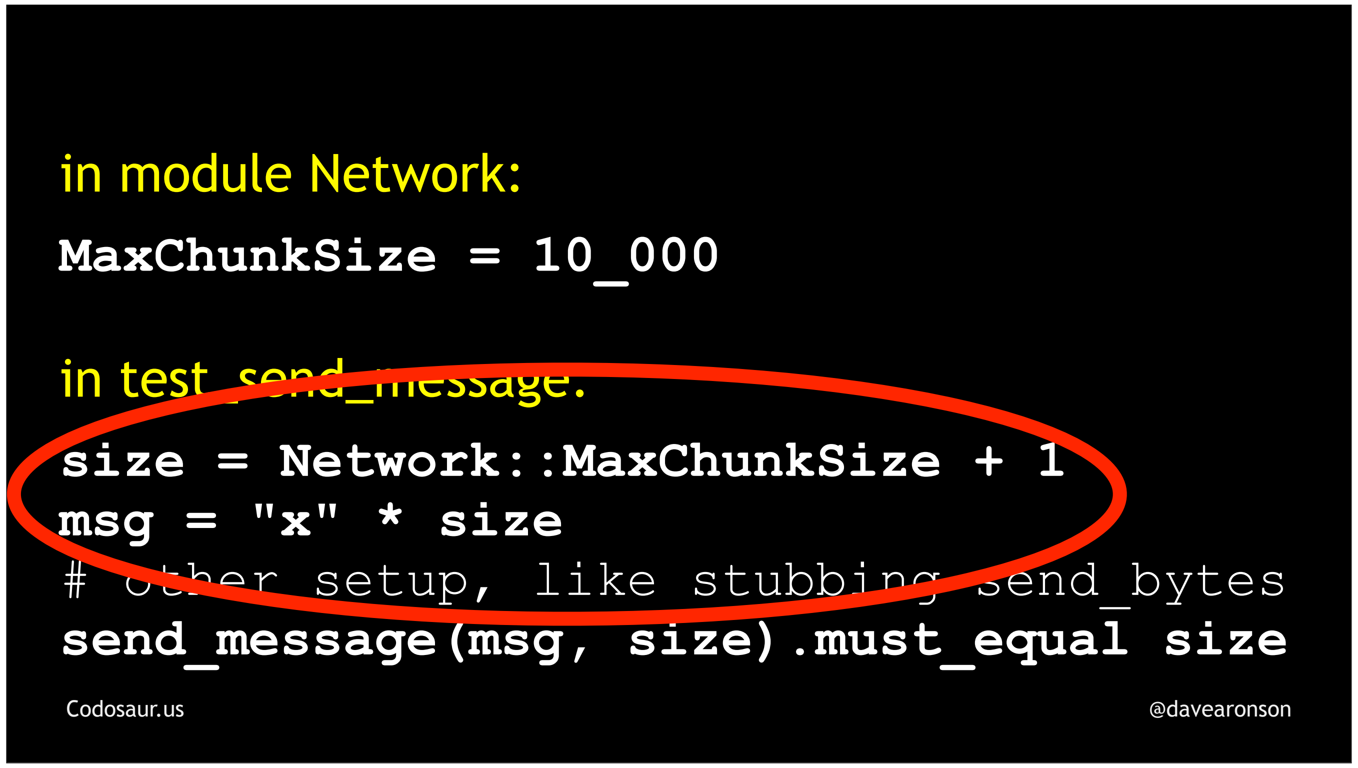
in test_send_message:

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

Codosaur.us @davearonson

. . . we're only testing with a tiny little *three* byte message. (Or maybe four if we include a null terminator. Whatever.) (PAUSE!)

The obvious fix is to use a message larger than our maximum chunk size. We can easily construct one, as shown . . .


in module Network:

```
MaxChunkSize = 10_000
```

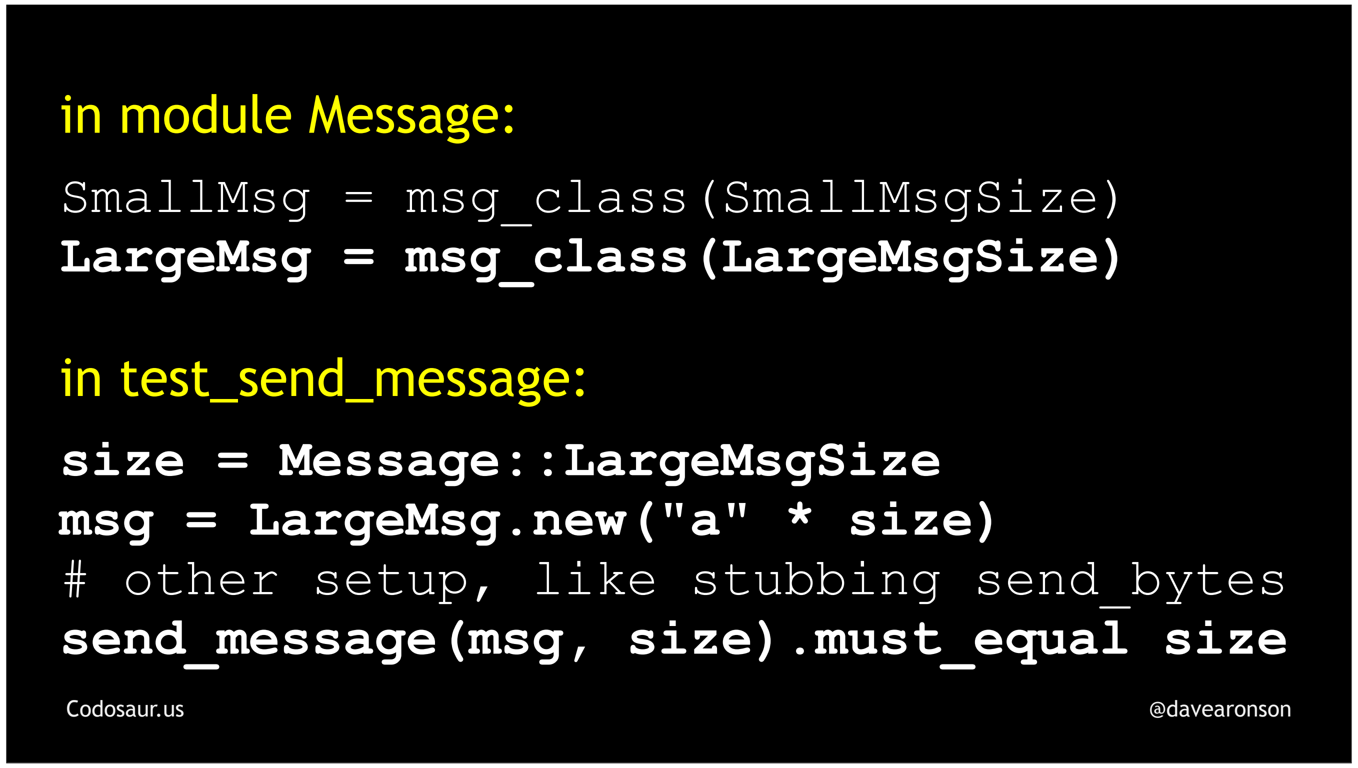
in test_send_message:

```
size = Network::MaxChunkSize + 1
msg = "x" * size
# other setup, like stubbing send_bytes
send_message(msg, size).must_equal size
```

Codosaur.us @davearonson

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

But perhaps, to paraphrase Shakespeare, the fault, dear $CITY, is not in our tests, but in our code, that these mutants are survivors. Perhaps 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:

```
SmallMsg = msg_class(SmallMsgSize)
LargeMsg = msg_class(LargeMsgSize)
```

in test_send_message:

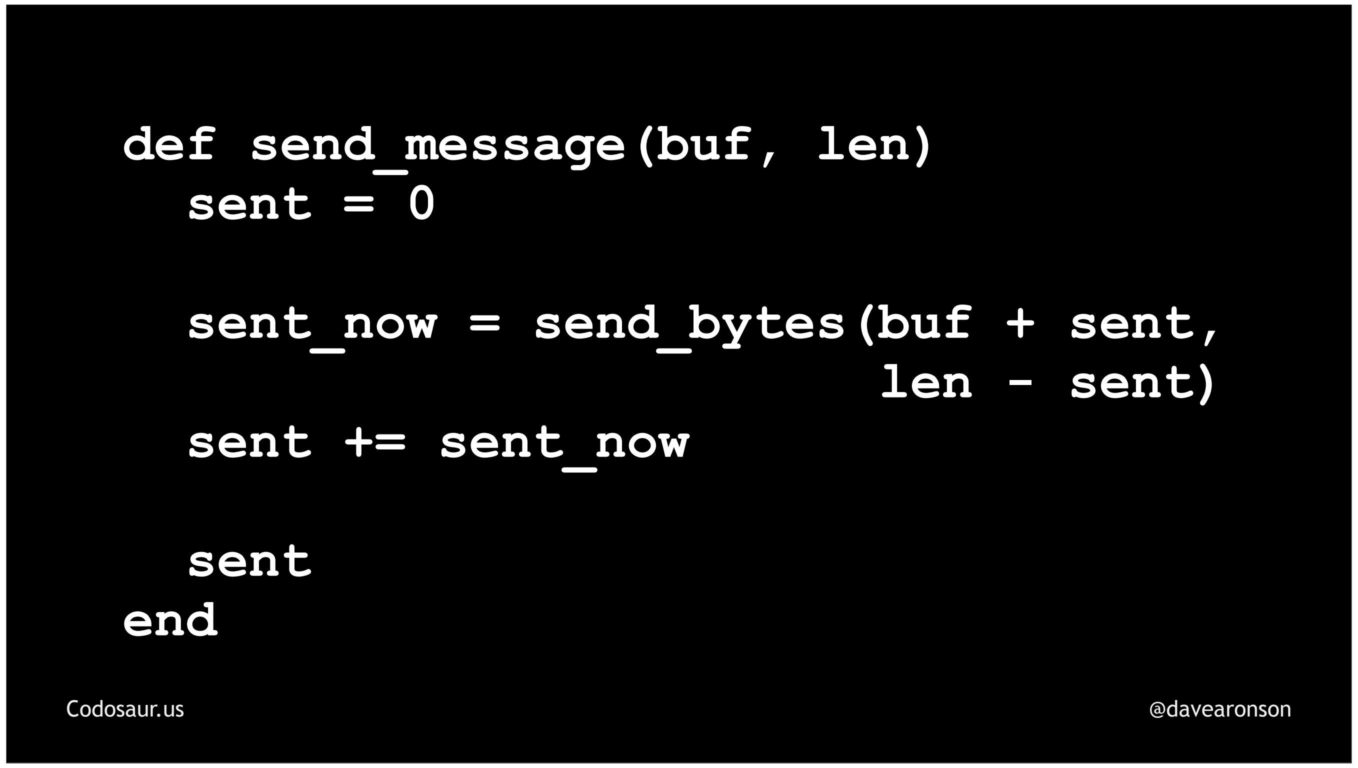
```
size = Message::LargeMsgSize
msg = LargeMsg.new("a" * size)
# other setup, like stubbing send_bytes
send_message(msg, size).must_equal size
```

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? What is this mutant trying to tell us now? (PAUSE!)

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 loop control, we wind up with . . .

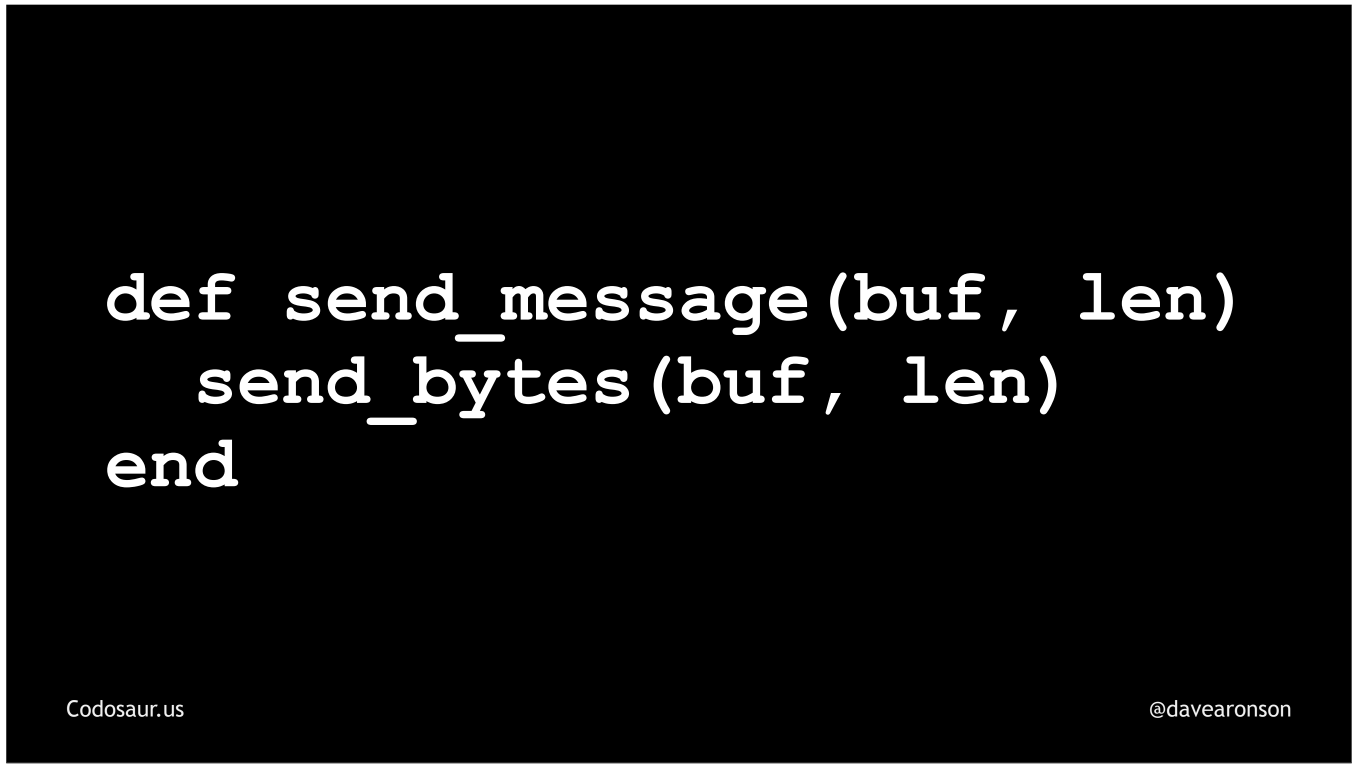


```
def send_message(buf, len)
  sent = 0

  sent_now = send_bytes(buf + sent,
                        len - sent)
  sent += sent_now

  sent
end
```

Codosaur.us

@davearonson

. . . this code I showed you earlier. If we rerun our mutation testing tool, it will show a lot of other stuff as now being redundant, because we only needed it to support the looping. If we *also* remove all of that, and lather rinse repeat, then it boils down to . . .

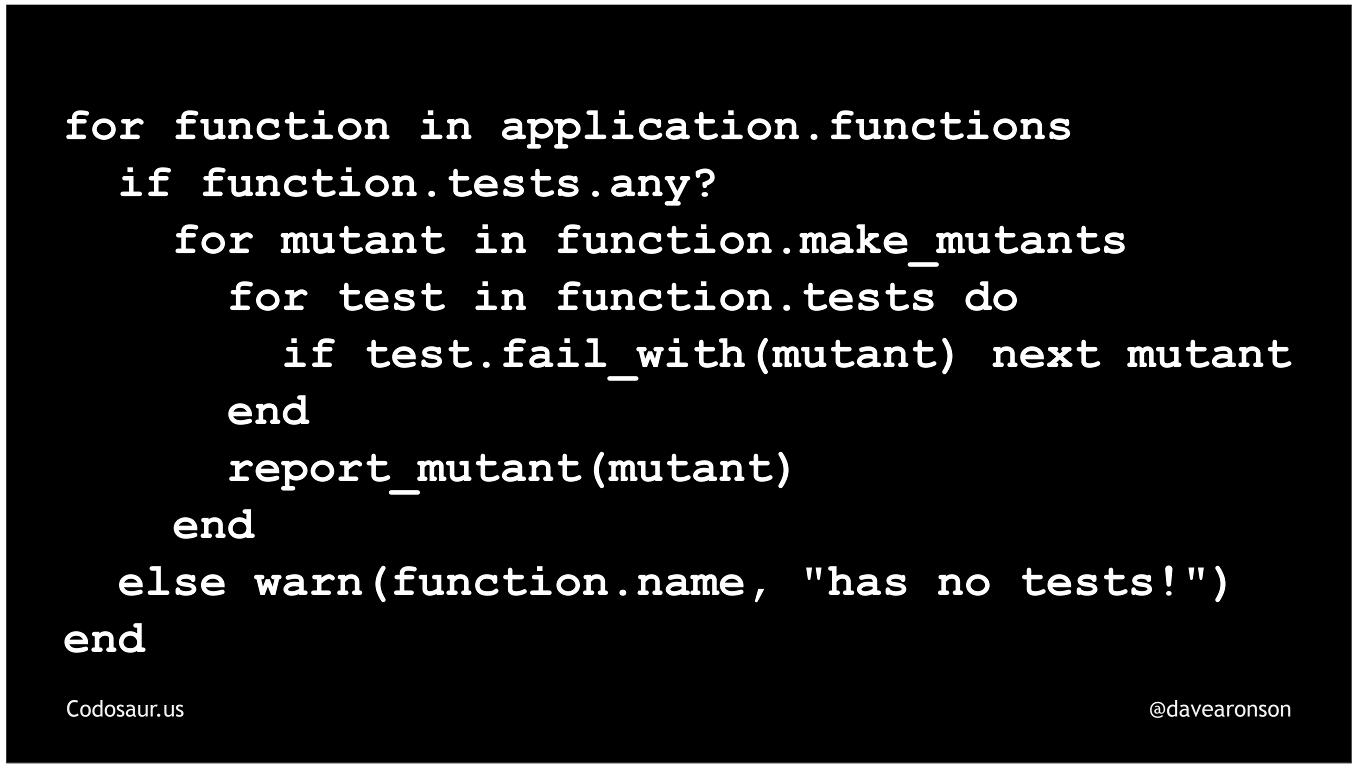


```
def send_message(buf, len)
  send_bytes(buf, len)
end
```

Codosaur.us @davearonson

. . . this. (PAUSE!) Now it's pretty clear: the *entire* send_message *function* may well be *redundant*, so we can just use send_*bytes directly!* It might not be, though, because, in real-world code, there may be some logging, error handling, and so on, needed in send_message, but at least the looping was redundant. Fortunately, when it's this kind of problem, with unreachable or redundant code, the 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.

Now that we've seen a few different examples, of spotting bad tests and redundant code, some of you might do better staring at code than hearing me talk about it, so . . .



```
for function in application.functions
  if function.tests.any?
    for mutant in function.make_mutants
      for test in function.tests do
        if test.fail_with(mutant) next mutant
      end
      report_mutant(mutant)
    end
  else warn(function.name, "has no tests!")
end
```

Codosaur.us @davearonson

. . . here's some pseudocode, showing how mutation testing works, from a very high level view. I'll pause a moment for you to read it or take pictures.

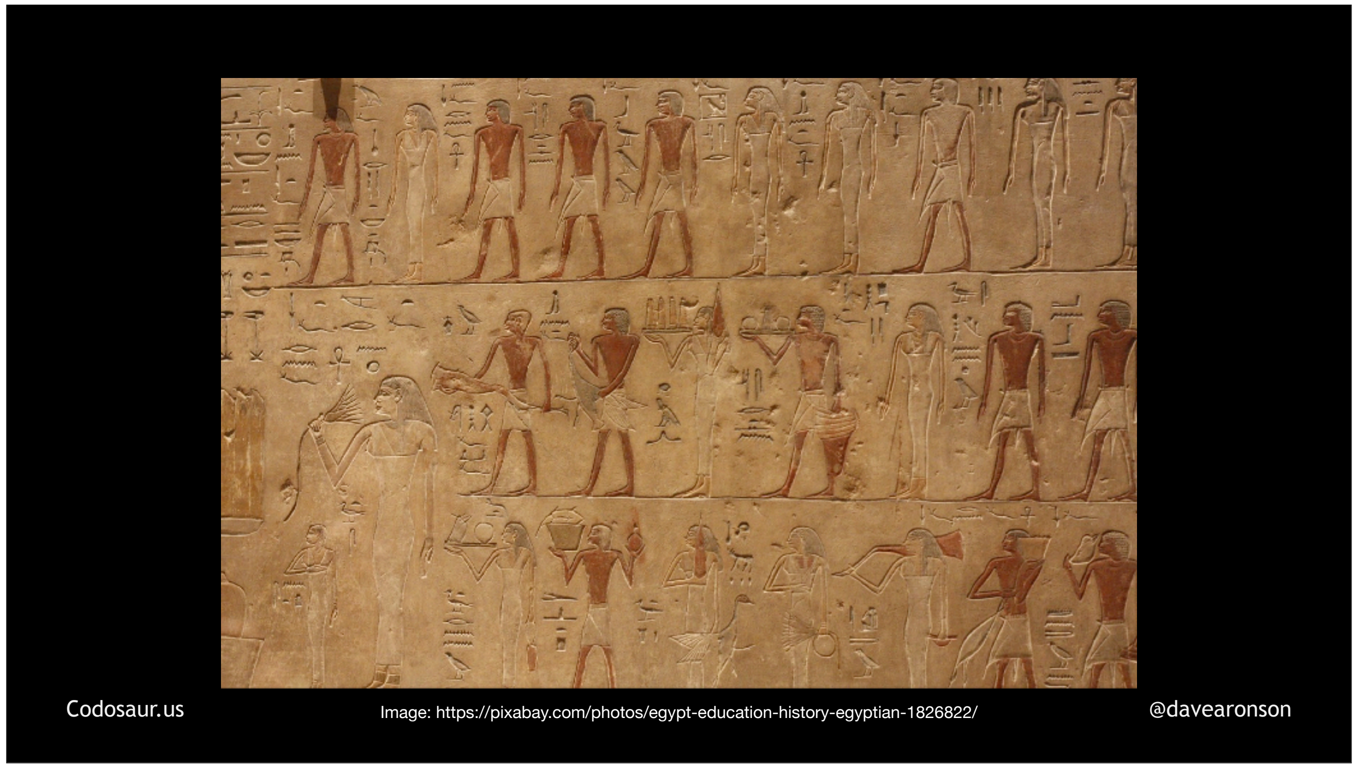
Next up, I'd like to address some . . .

?????

Codosaur.us

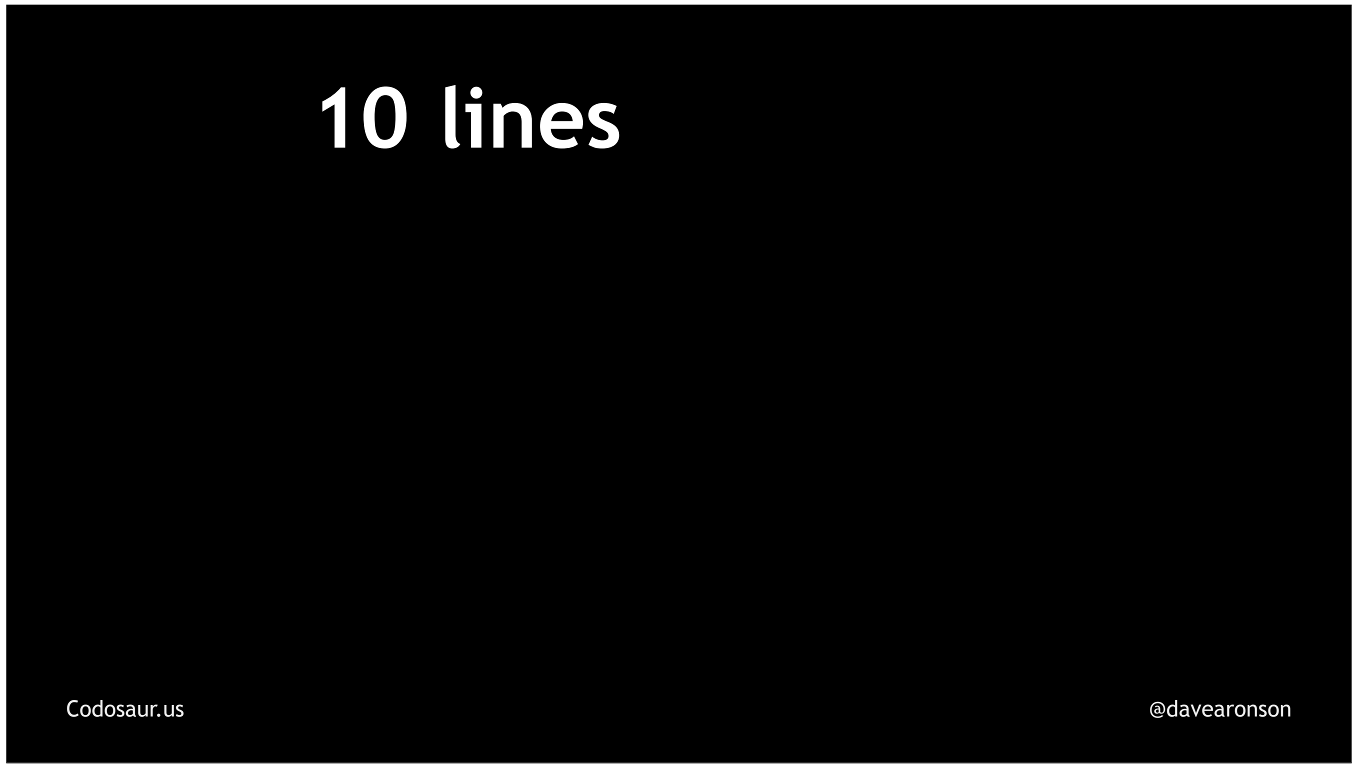
@davearonson

. . . occasionally asked questions. (Mutation testing is still rare enough that I don't think there *are* any *frequently* asked 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 so, it was not *practical* for most people, with consumer-grade computers, until recently, maybe the past couple decades, 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-intensive? To answer that, we need do some math, but don't worry, it's pretty basic. Suppose our functions have, on average, . . .

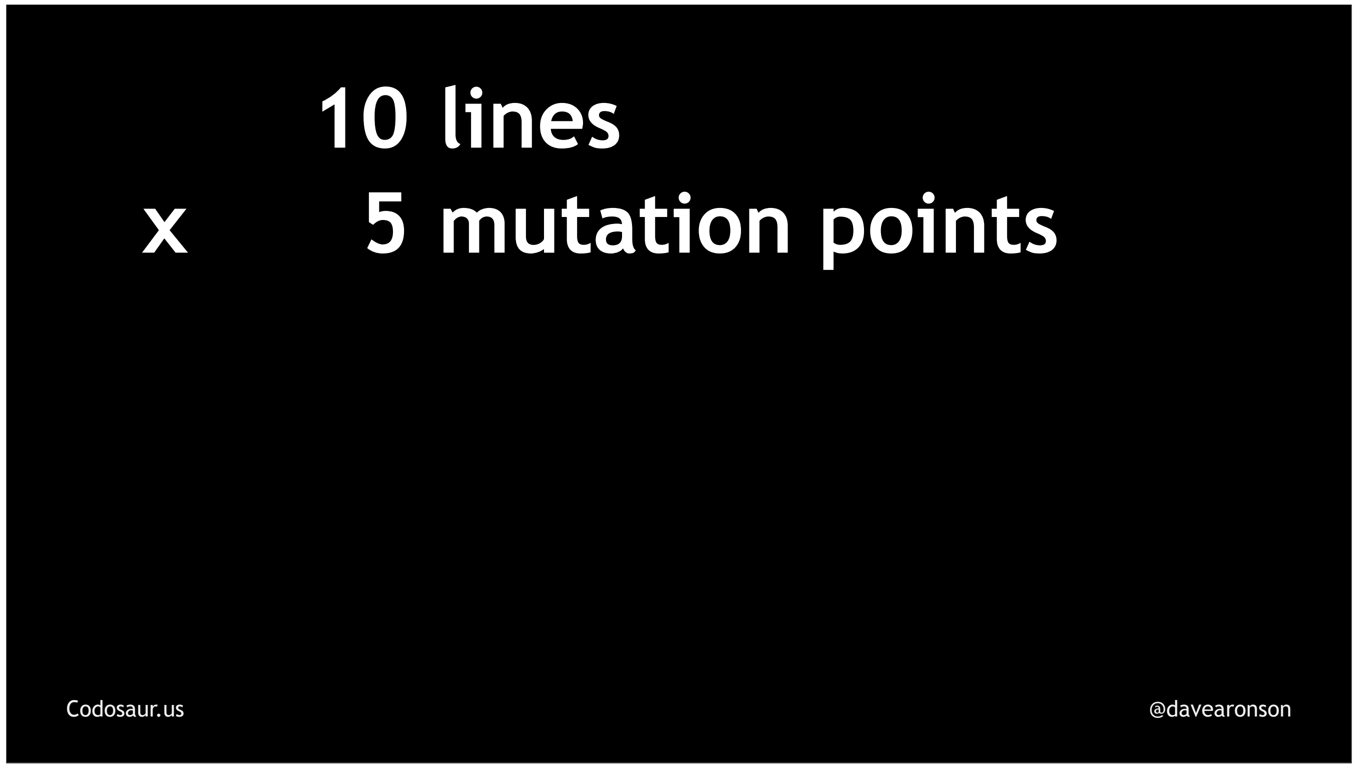


10 lines

Codosaur.us

@davearonson

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

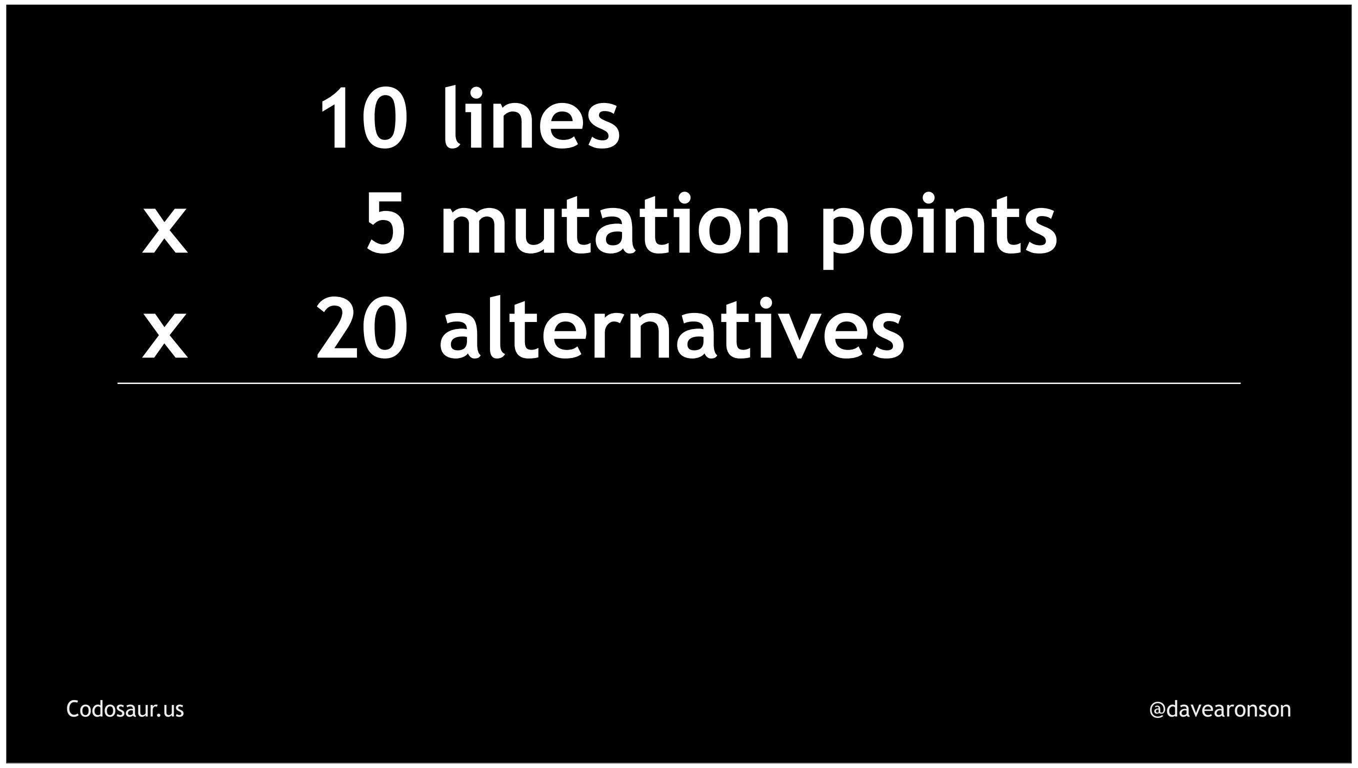


10 lines
x 5 mutation points

Codosaur.us

@davearonson

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

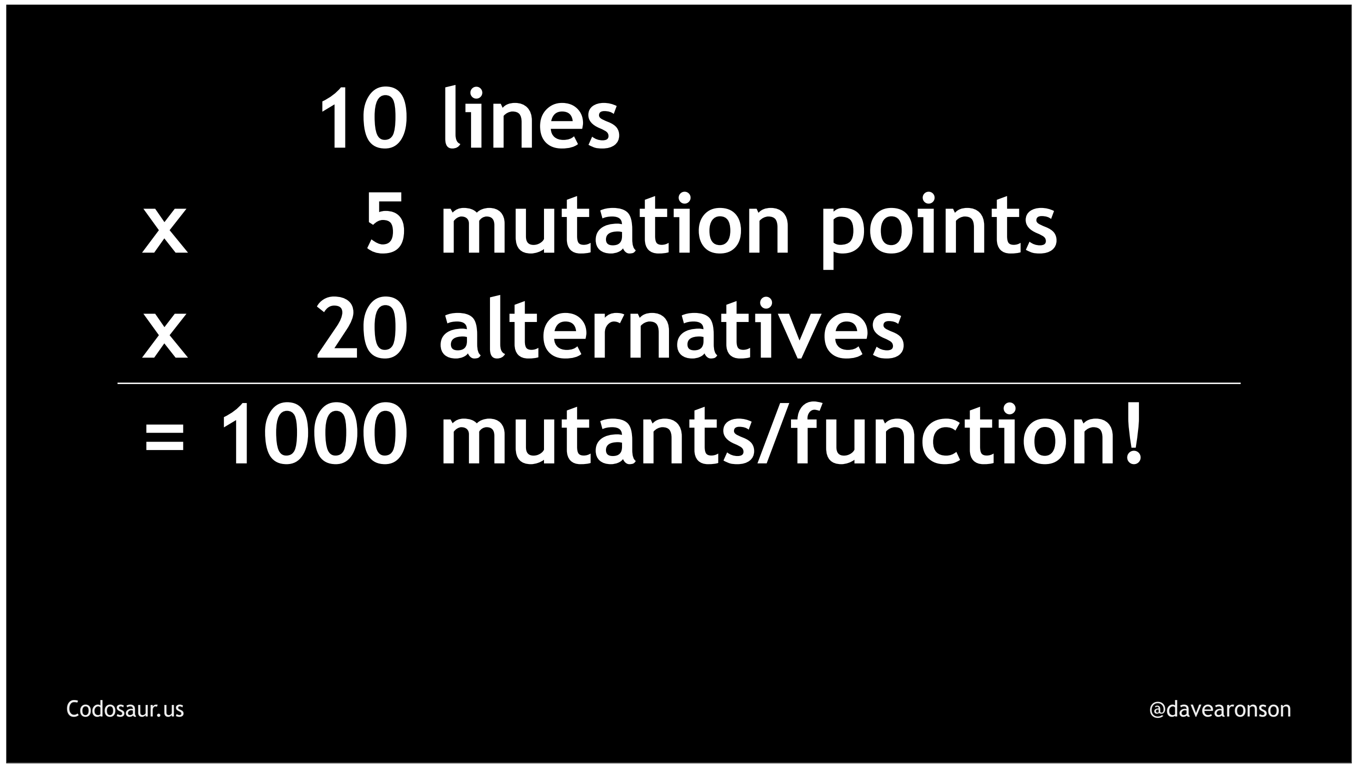


```
     10 lines
x     5 mutation points
x    20 alternatives
```

Codosaur.us

@davearonson

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



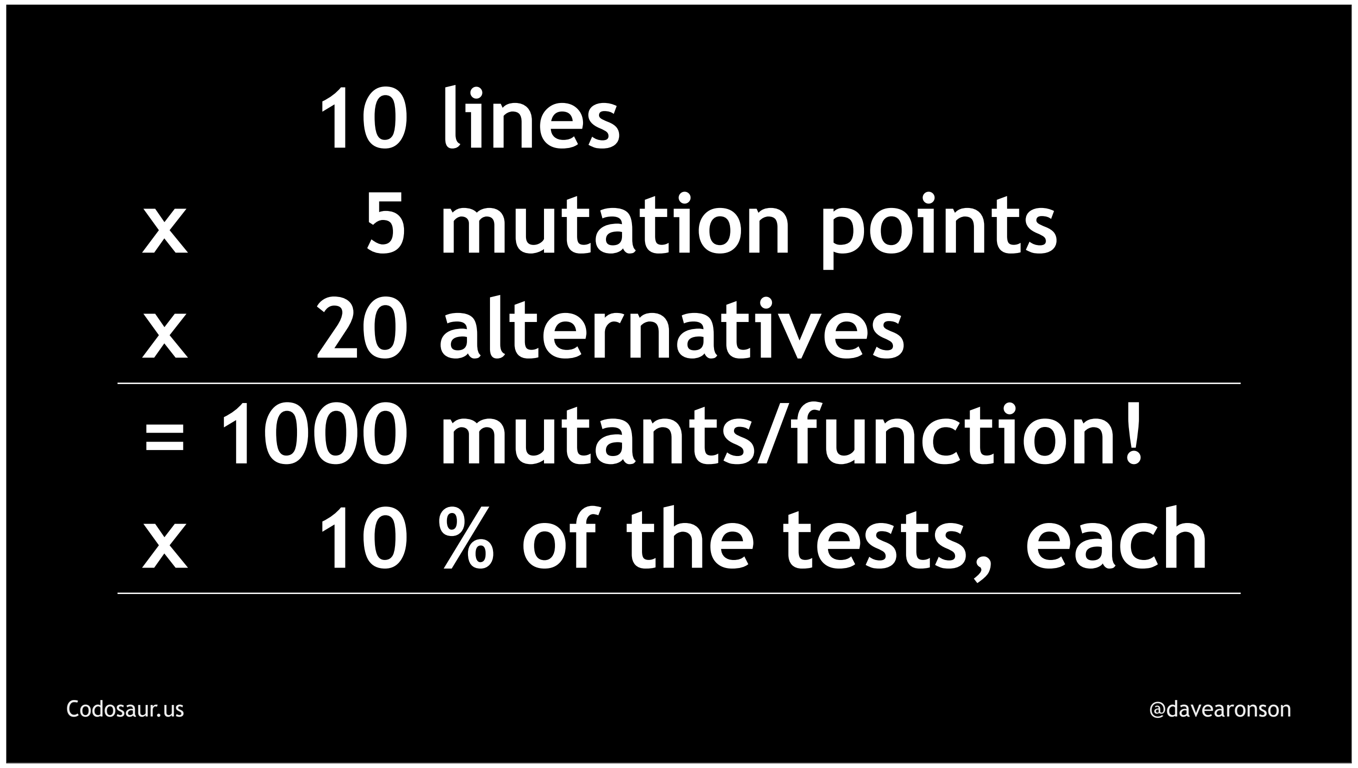
```
       10 lines
x       5 mutation points
x      20 alternatives
-------------------------
= 1000 mutants/function!
```

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 it on the first try, and *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 . . .

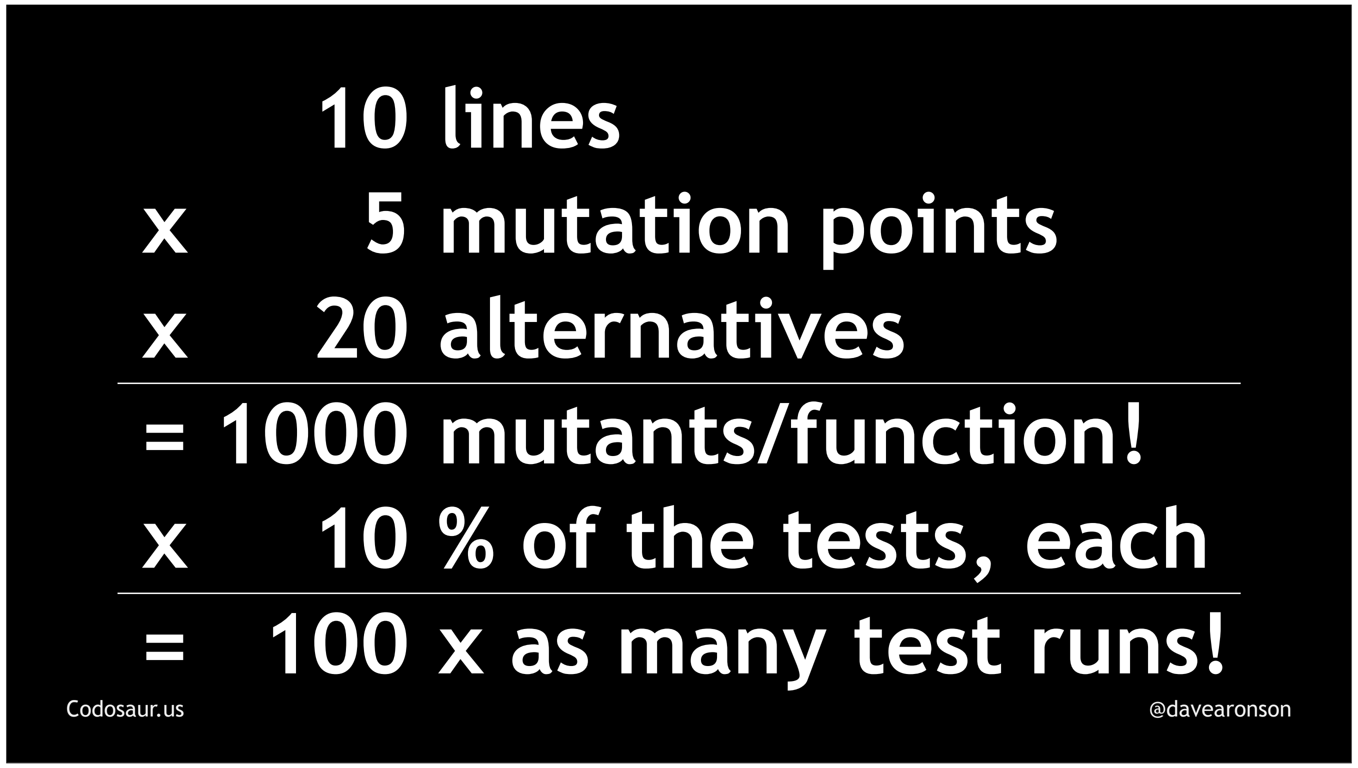


```
     10 lines
x     5 mutation points
x    20 alternatives
= 1000 mutants/function!
x    10 % of the tests, each
```

Codosaur.us

@davearonson

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



```
       10 lines
x       5 mutation points
x      20 alternatives
--------------------------------
=    1000 mutants/function!
x      10 % of the tests, each
--------------------------------
=     100 x as many test runs!
```

Codosaur.us

@davearonson

. . . a *hundred times* the test runs for that function. If our test suite normally takes a zippy ten seconds, mutation testing will take about a *thousand* seconds. That might not sound like much, because I'm saying "seconds", but do the math and it's *almost 17 minutes!*

But there is some . . .

Codosaur.us

Image: https://www.flickr.com/photos/akuchling/50310316

@davearonson

. . . 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 error condition. Such things have reduced the mutant horde by up to about two thirds! But even with that rare level of success, it's still . . .

Codosaur.us

Image: https://www.flickr.com/photos/cavenagh/15815117738/

@davearonson

. . . no silver bullet, as this takes lots of CPU time itself -- and the remaining mutants are still quite a lot.

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

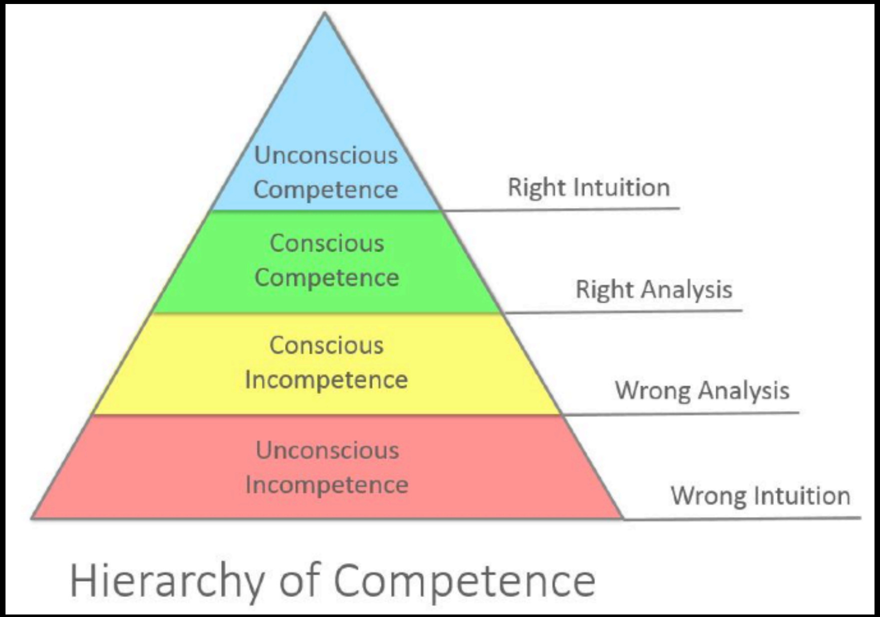
Codosaur.us

Image: https://pixabay.com/vectors/arrow-one-way-right-sign-road-759223/

@davearonson

. . . one way? (NOTE TO SELF: THIS IS ONGOING DEBATE, LOOK INTO HOW THEY MAKE IT OK!)

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



Codosaur.us Image: https://commons.wikimedia.org/wiki/File:Competence_Hierarchy_adapted_from_Noel_Burch_by_Igor_Kokcharov.jpg @davearonson

. . . 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, enough to confirm the hypothesis. The rest of you, you probably really are competent, so you might want to read up on Impostor Syndrome.

But anyway, what is the Competent Programmer Hypothesis? Long story short, it's 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 a typo, or adding when we should subtract, or saying "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 more or less exactly the kind of substitutions that a mutation testing tool makes. You can think of mutation testing as sort of a "did you mean" function, like how Google suggests something else if your search didn't have many hits.

Another reason is that it helps us poor humans . . .

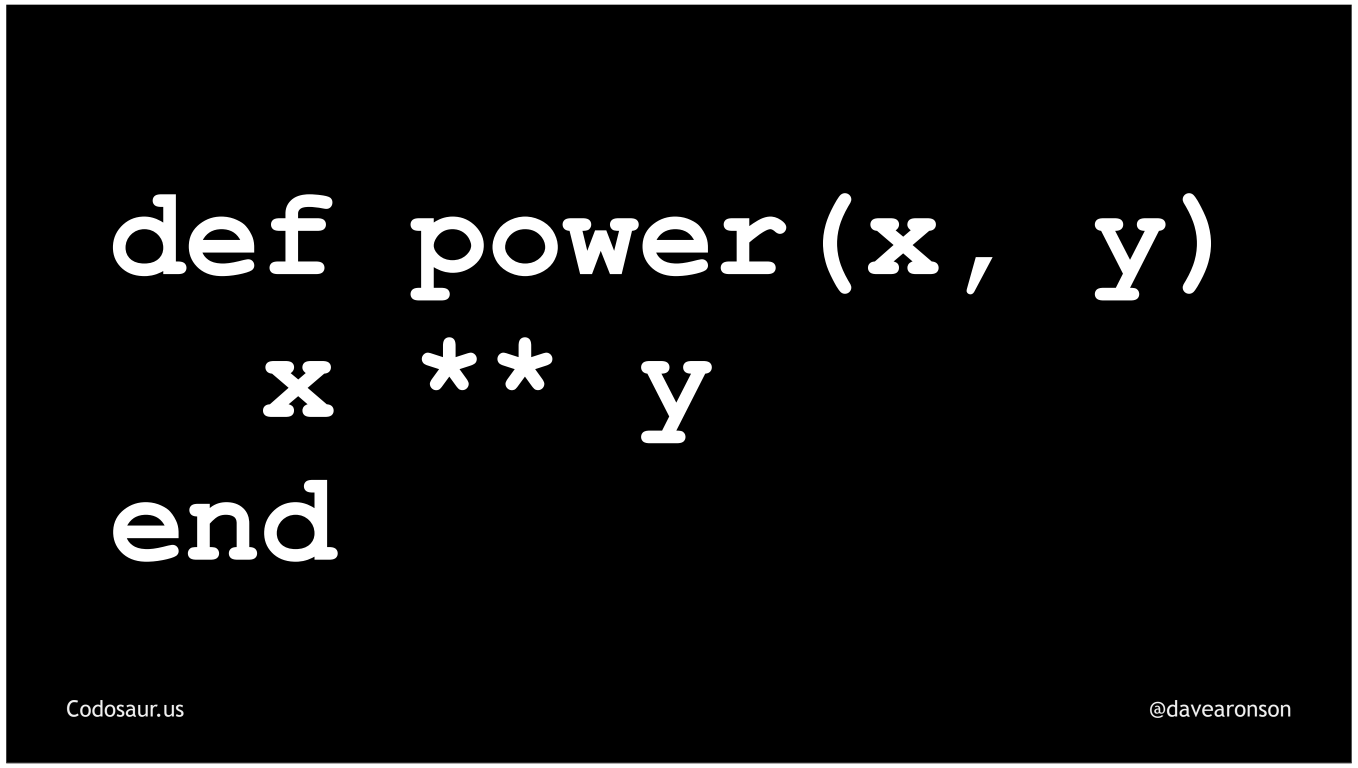
Codosaur.us
Image: https://pixabay.com/vectors/arrow-one-way-right-sign-road-759223/
@davearonson

. . . 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. You can think of it like using the Single Responsibility Principle.

Another reason is that multiple changes may . . .

Codosaur.us
Image: https://www.needpix.com/photo/download/600681/balance-brass-court-justice-law-lawyer-measure-scales-weight
@davearonson

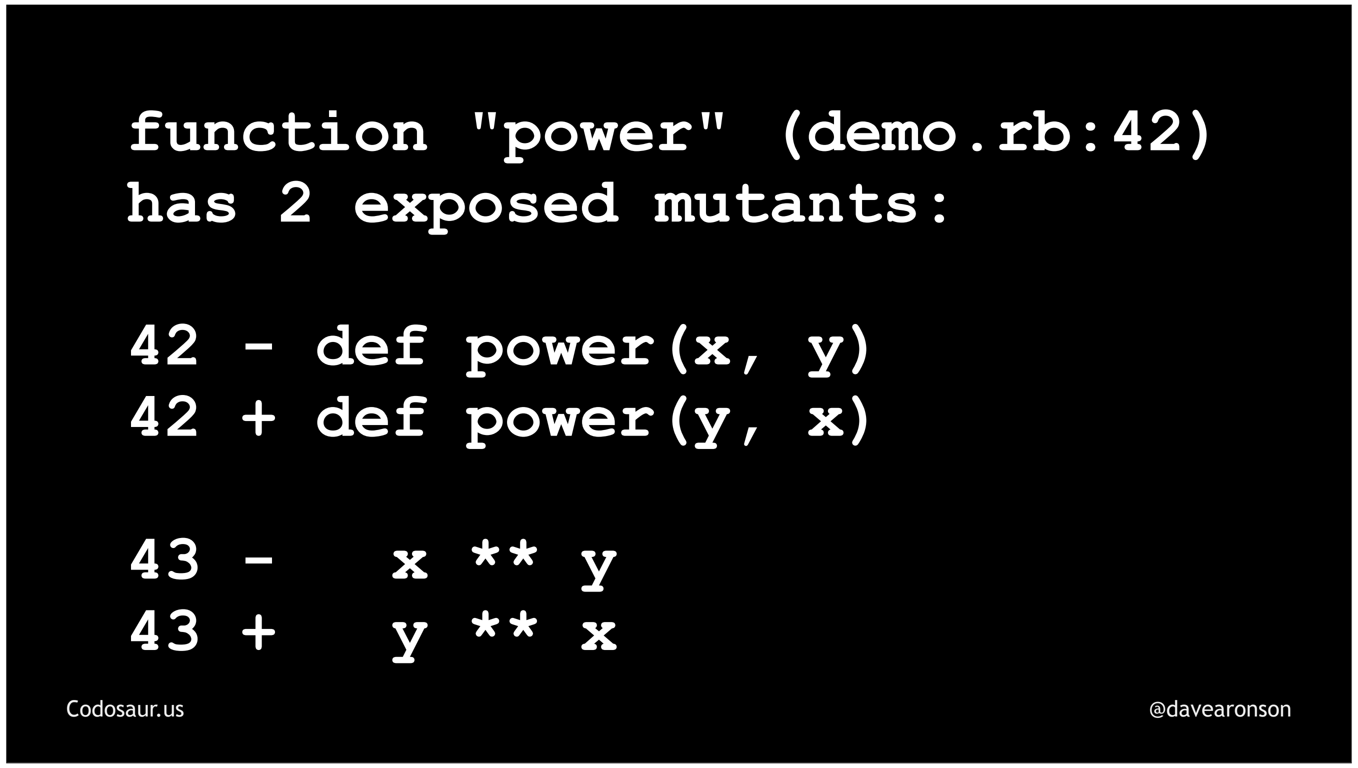
. . . balance each other out, leading to more false alarms.   For instance, remember that . . .



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

Codosaur.us @davearonson

. . . first simple example, and . . .

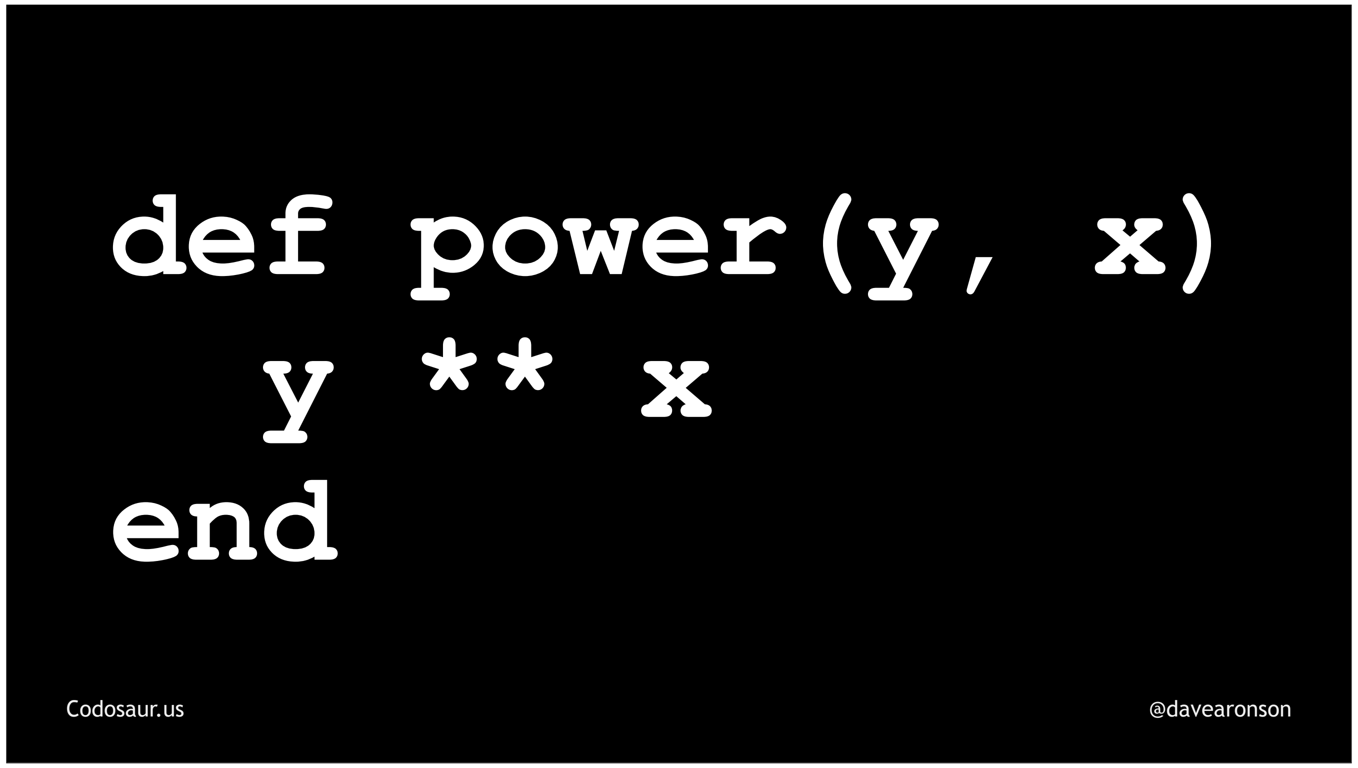


```
function "power" (demo.rb:42)
has 2 exposed mutants:

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

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

Codosaur.us

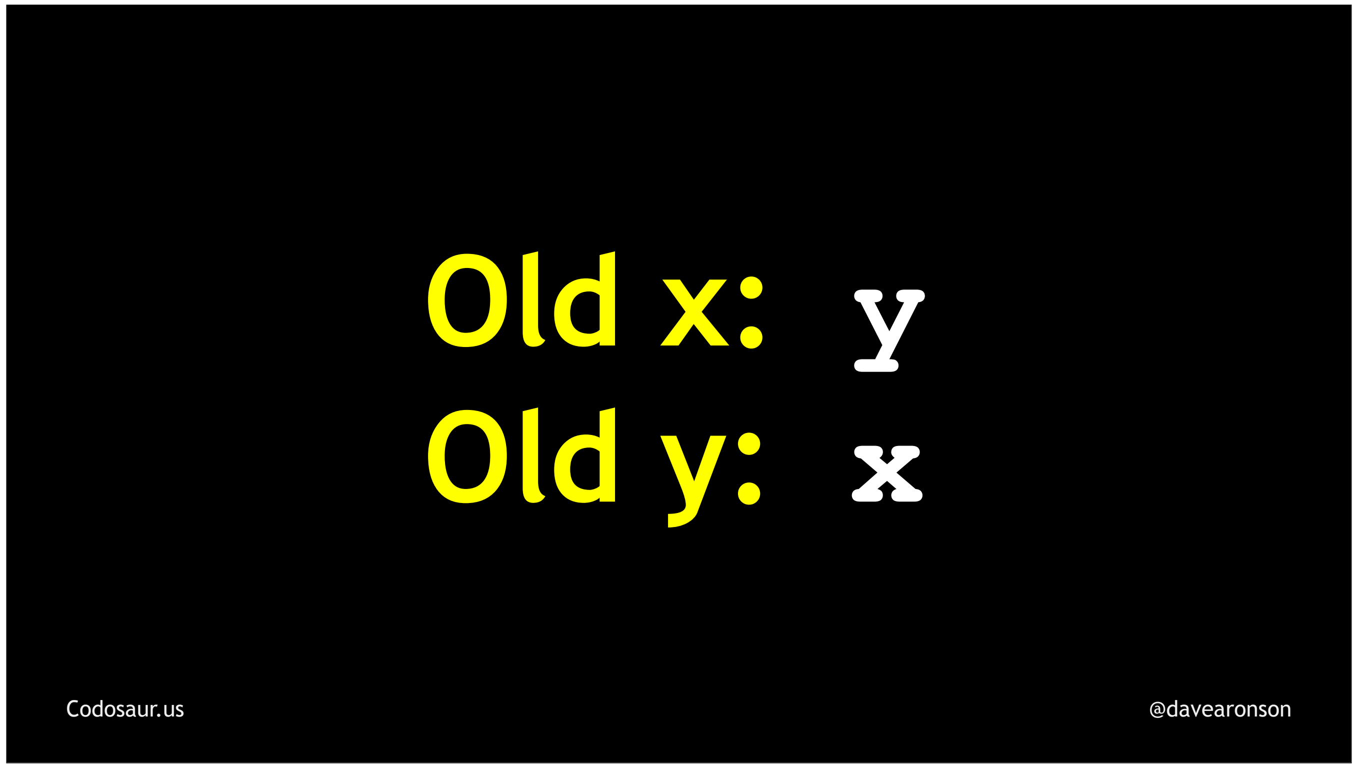
@davearonson

. . . its argument-swapping mutants? If one mutant . . .



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

. . . had *both* of these mutations, . . .



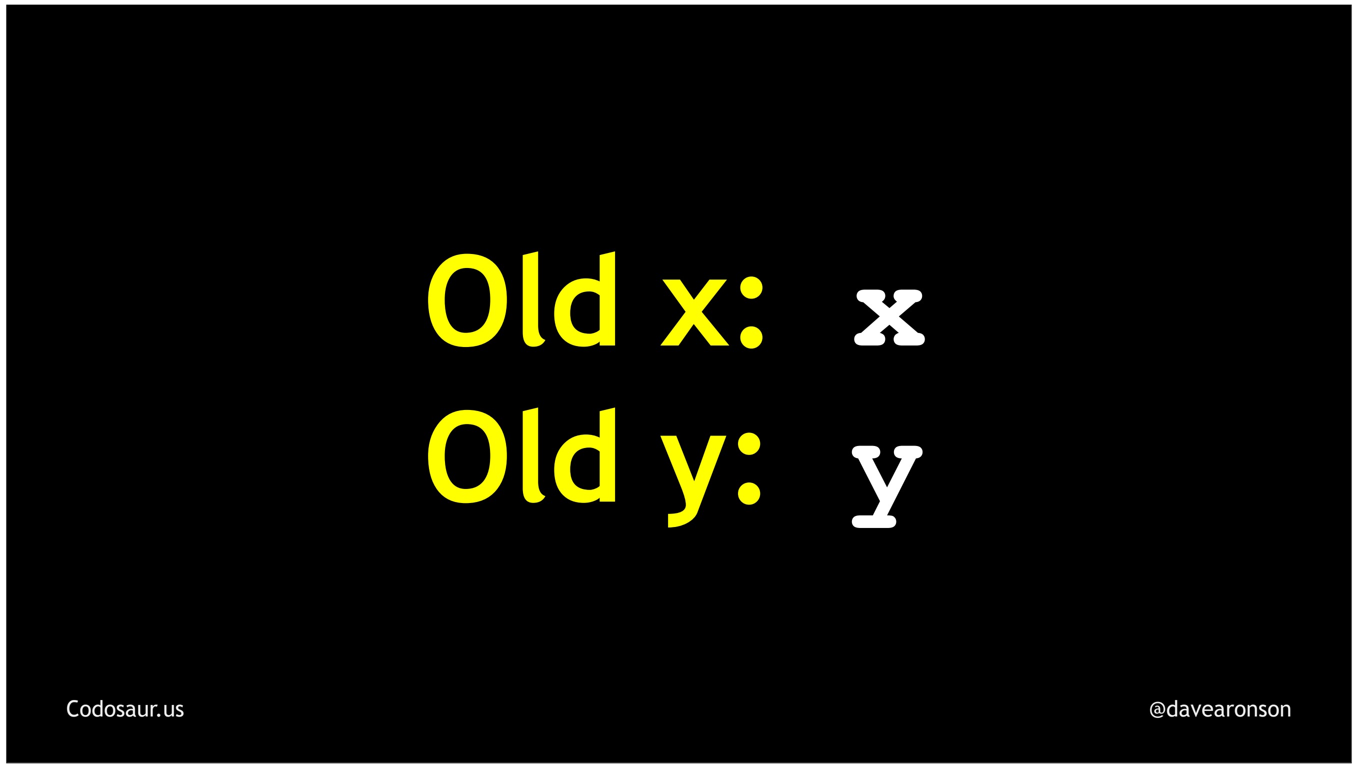
Old x: `y`

Old y: `x`

Codosaur.us

@davearonson

. . . the first would swap the arguments, and the other would . . .



Old x: `x`

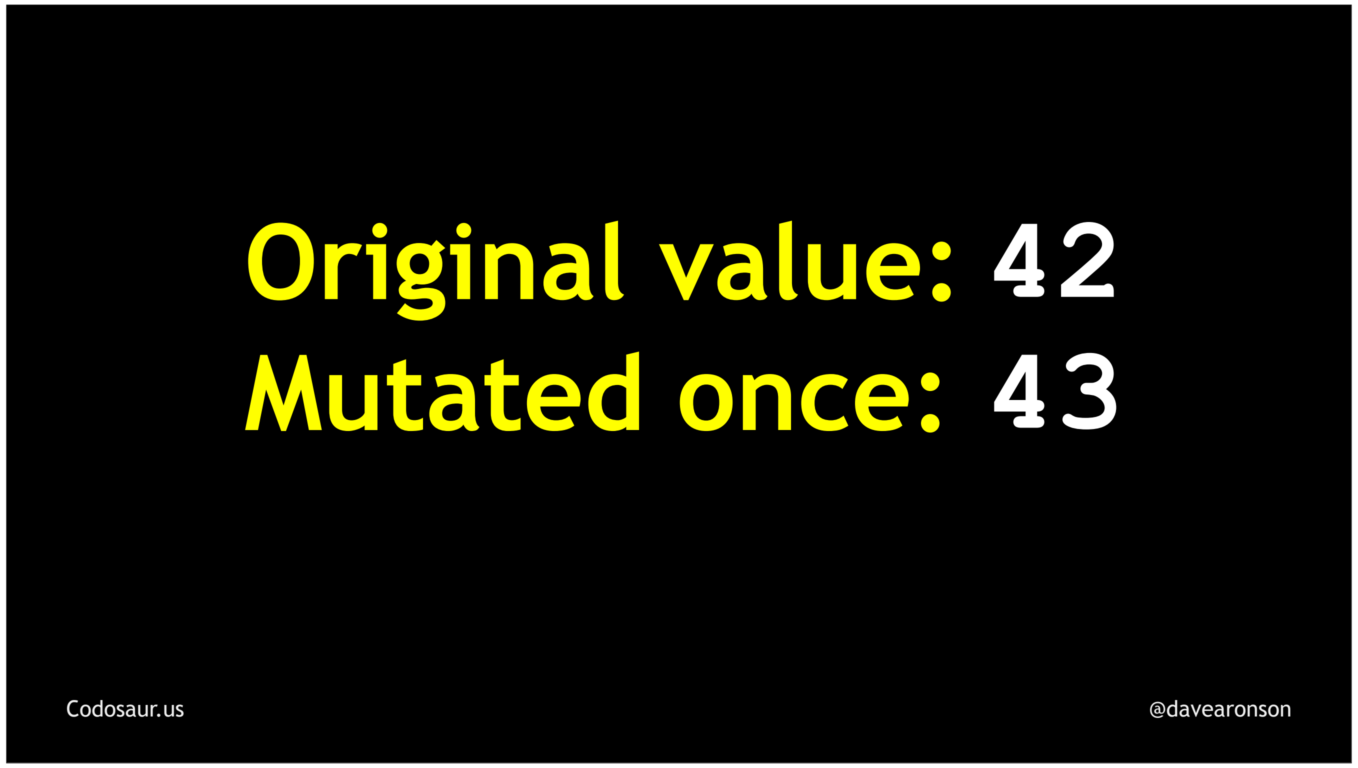
Old y: `y`

Codosaur.us

@davearonson

. . . swap them right back, for no net effect. (SKIP: There has actually been *some* research into weeding out redundant multi-change mutants, but the whole idea of multi-change mutants is not widely accepted.)

Another trivial example would be if . . .

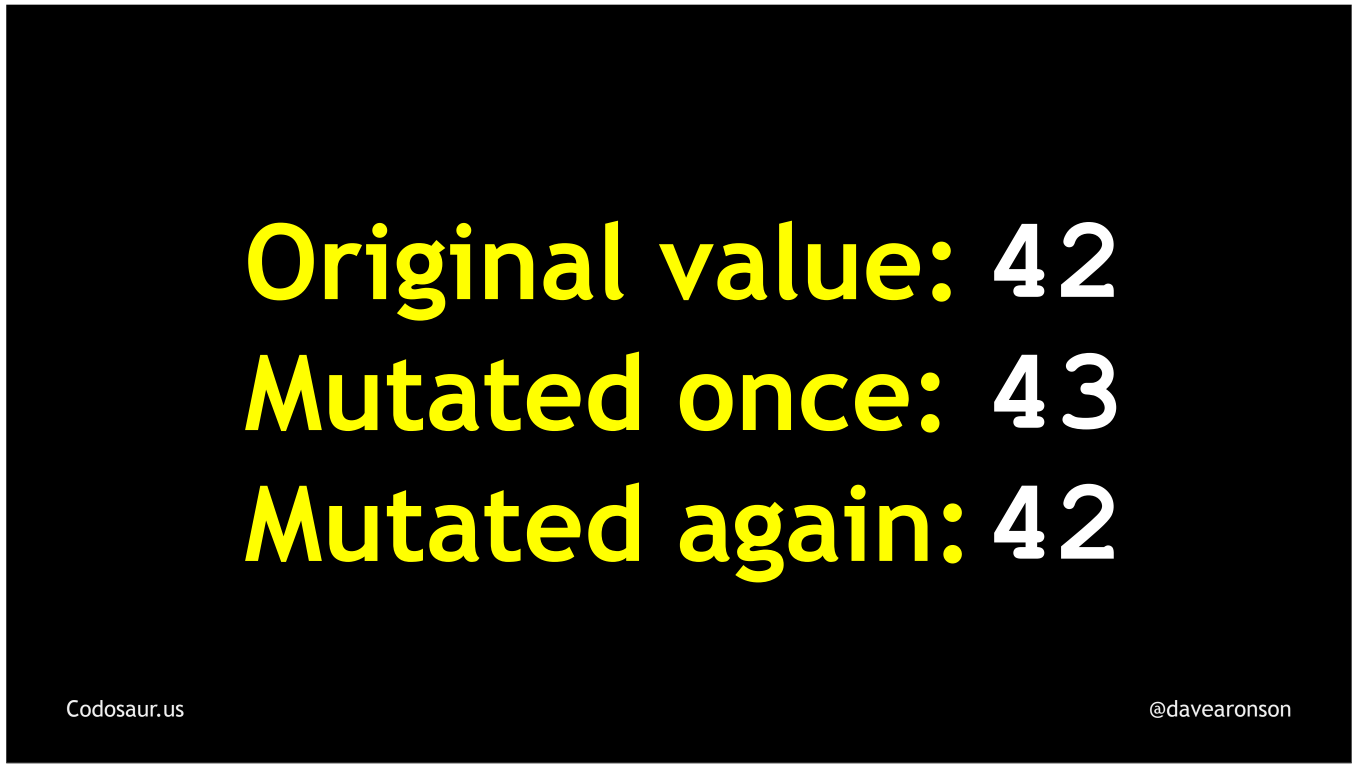


Original value: 42
Mutated once: 43

Codosaur.us

@davearonson

. . . one mutation incremented something, . . .



Original value: 42
Mutated once: 43
Mutated again: 42

Codosaur.us

@davearonson

. . . and another decremented it, right back to its original value.

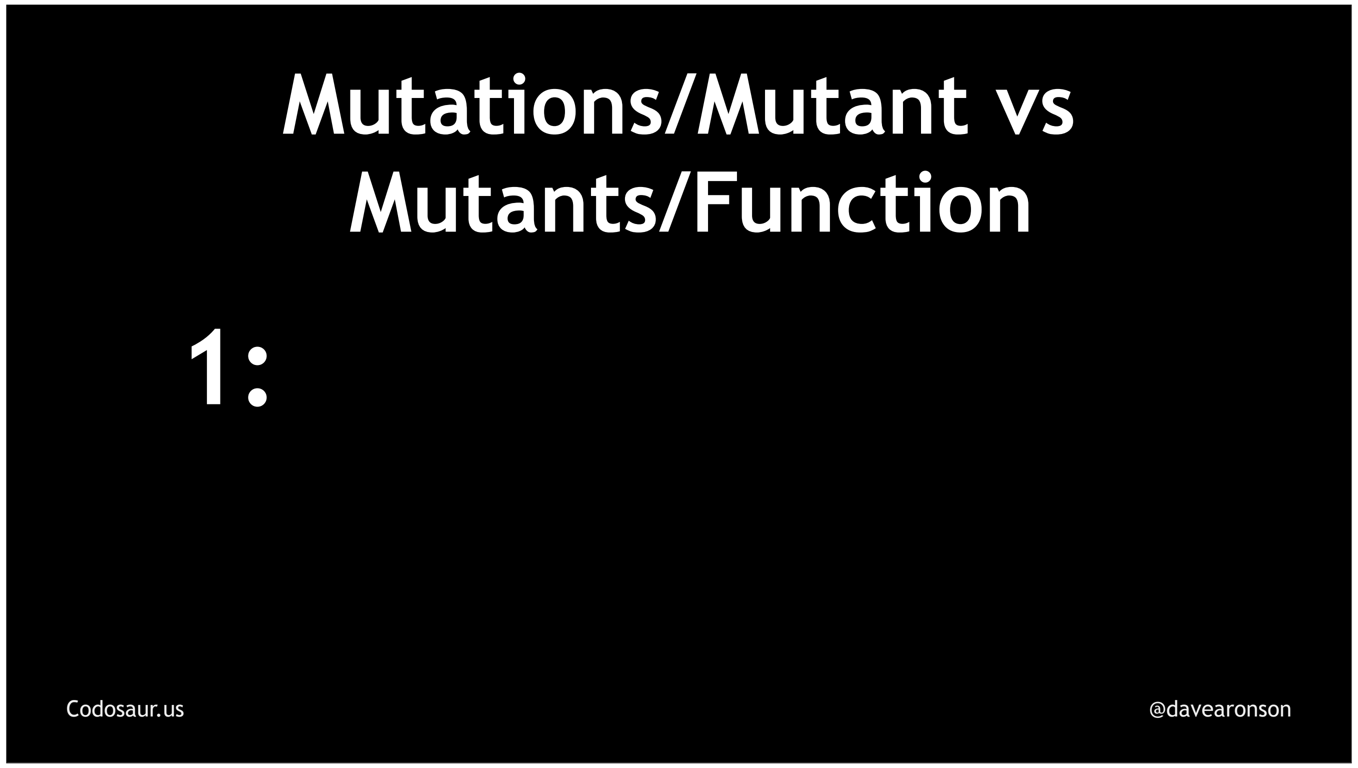
Lastly, allowing multiple mutations would create a combinatorial . . .

Codosaur.u

Image: https://pixnio.com/miscellaneous/fireworks/explosion-party-firework-festival

avearonson

. . . explosion of mutants, with the tool making many *orders of magnitude* more mutants per function, which would make it even *more* CPU-intensive. I'll spare you the math, but with our earlier code size assumptions, even if we manage to weed the mutants down by 2/3 at each step, with . . .



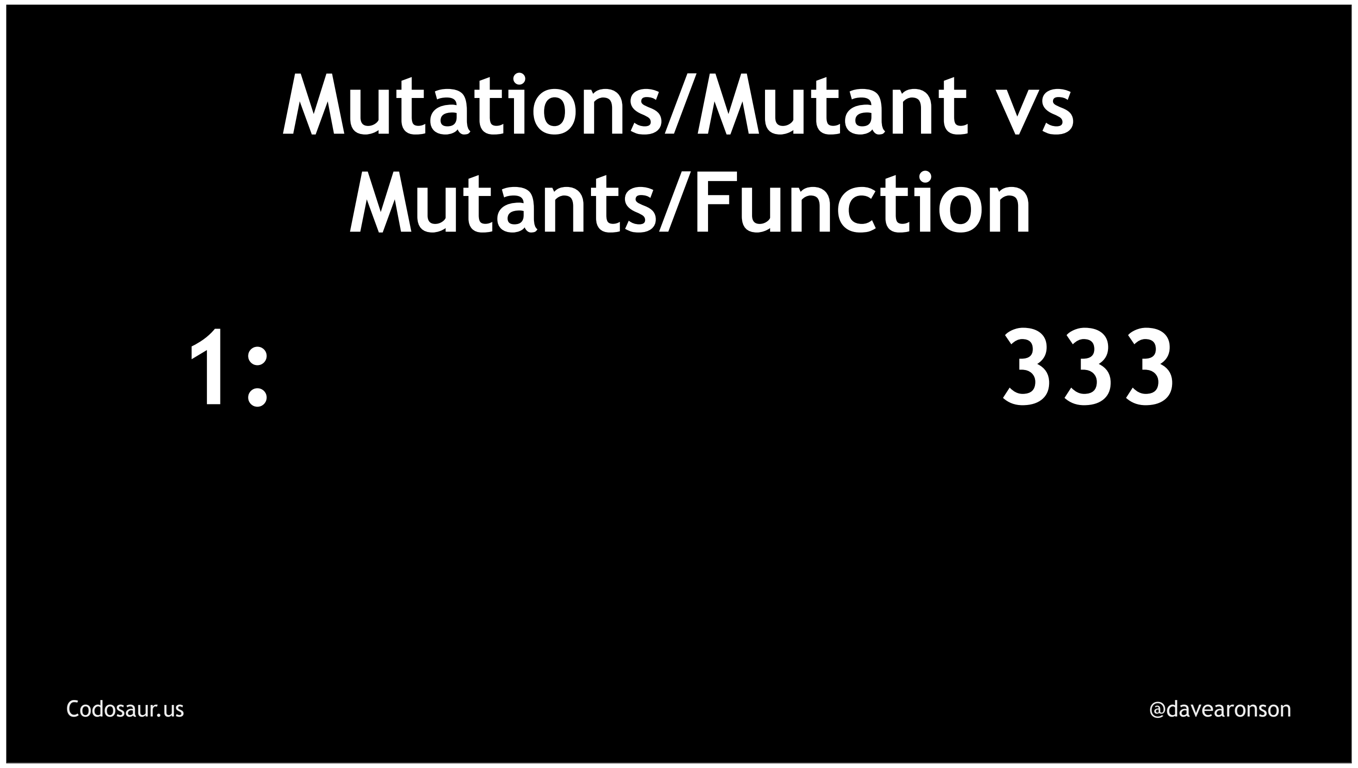
Mutations/Mutant vs Mutants/Function

1:

Codosaur.us

@davearonson

. . . one mutation per mutant, we'd have . . .

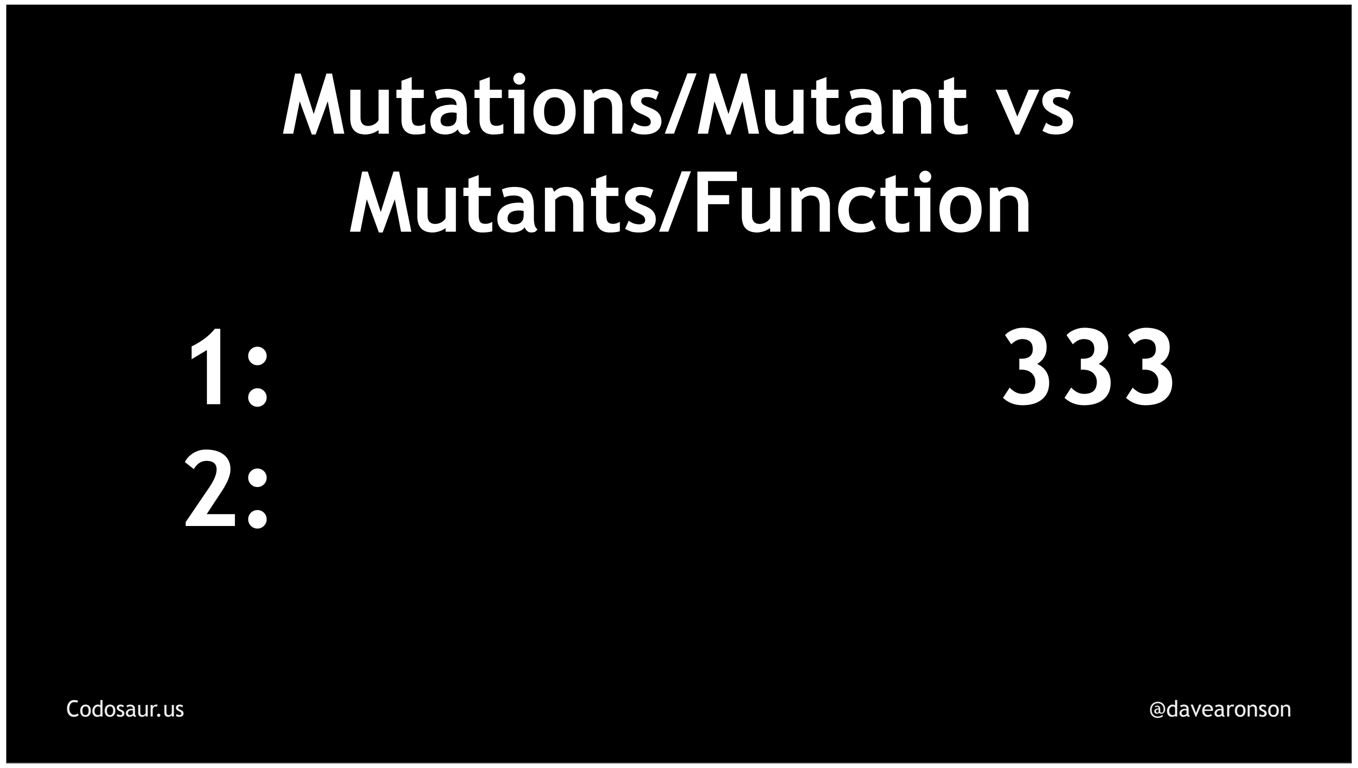


Mutations/Mutant vs Mutants/Function

1: 333

Codosaur.us @davearonson

. . . 333 mutants per function (and a third, but I'm rounding). With . . .

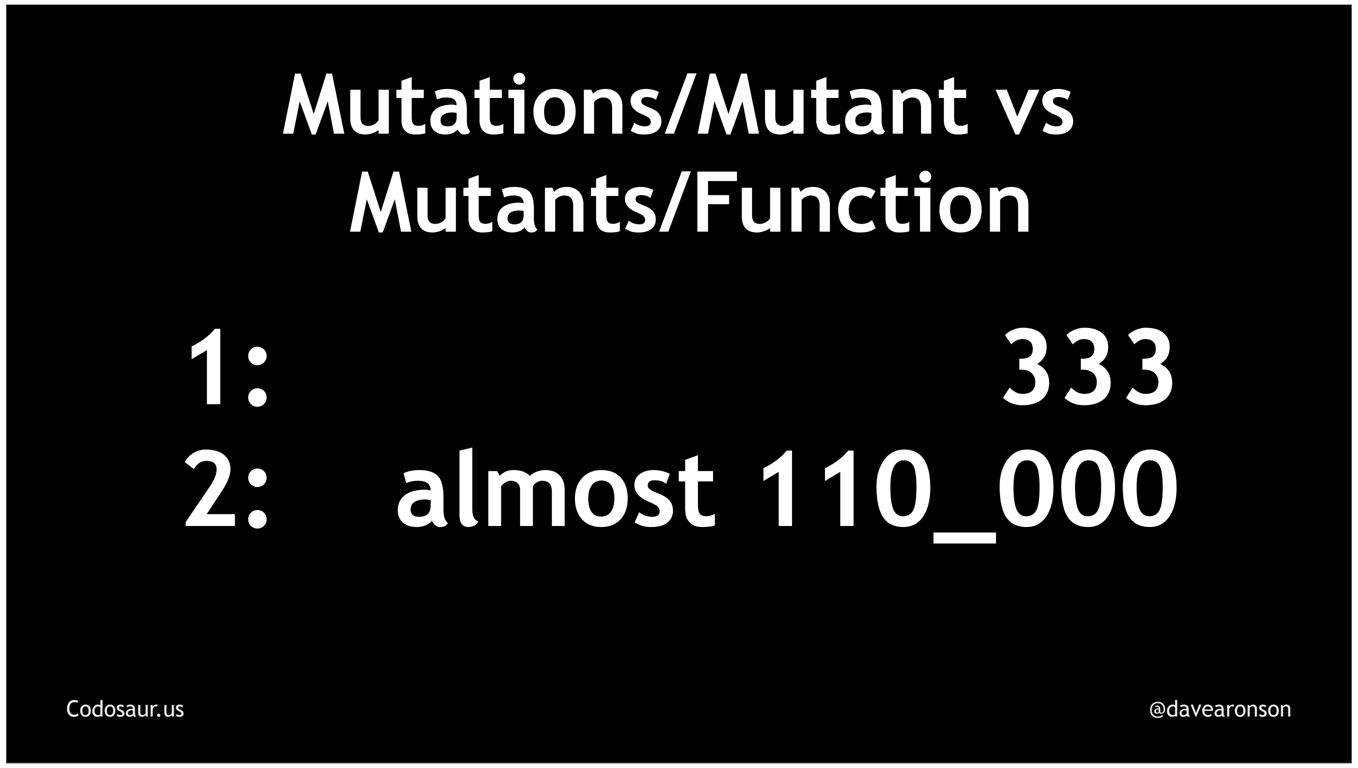


Mutations/Mutant vs Mutants/Function

```
1:                333
2:
```

Codosaur.us @davearonson

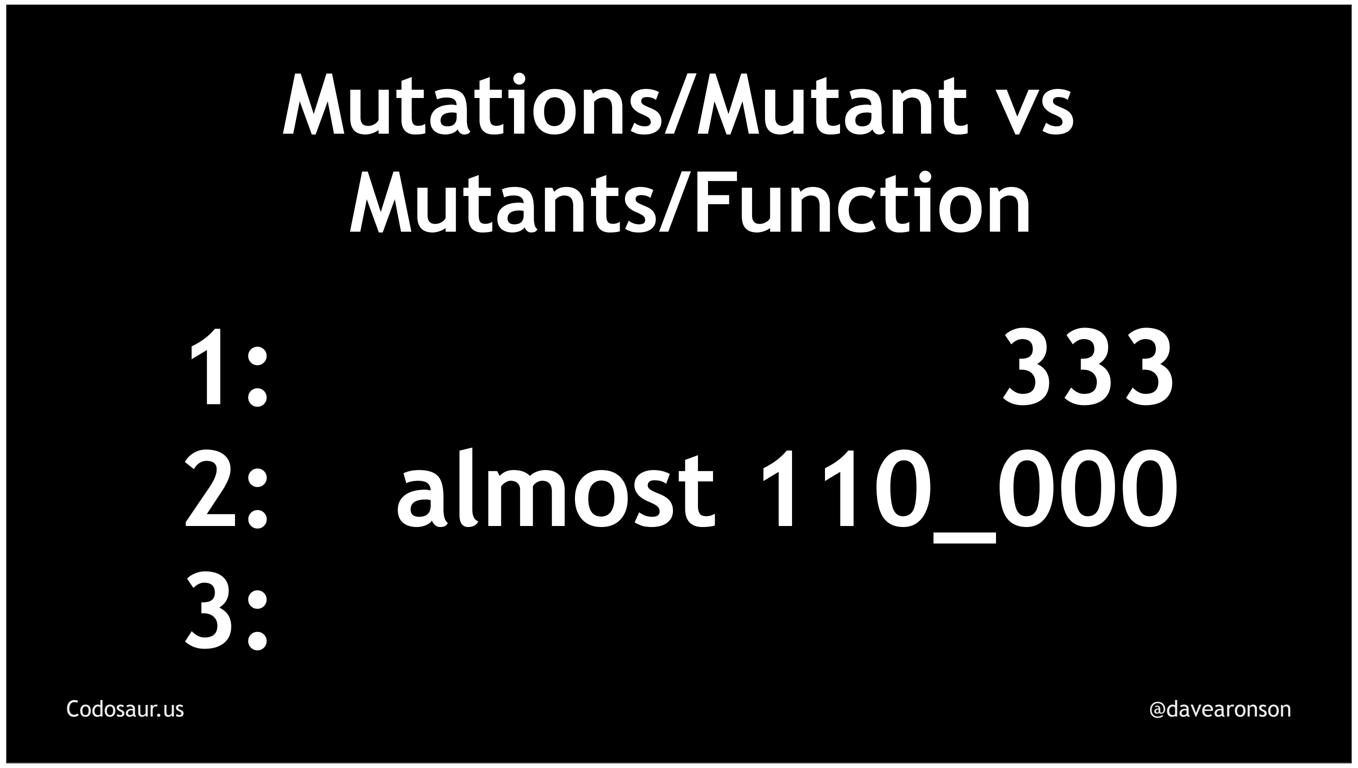
. . . two, we'd already have . . .



Mutations/Mutant vs Mutants/Function

1: 333
2: almost 110_000

. . . almost 110,000, and with . . .



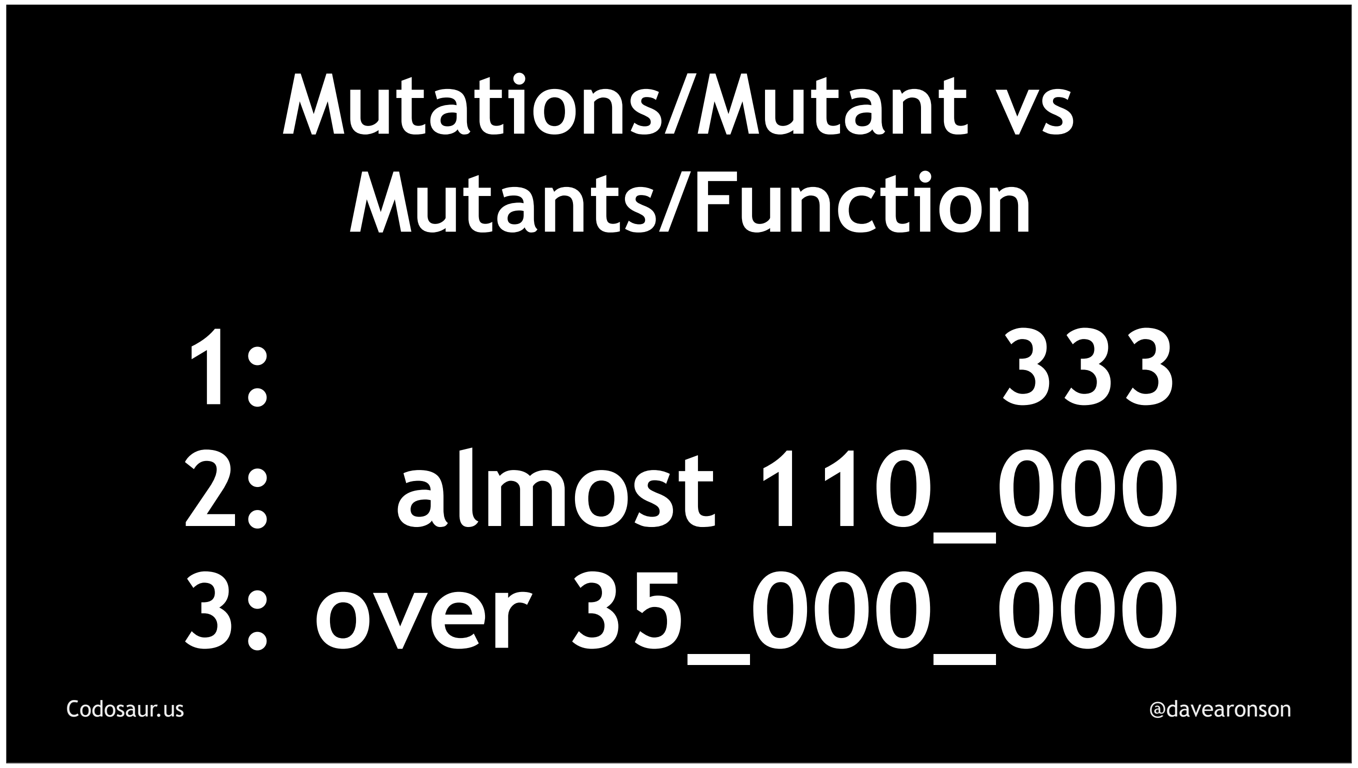
Mutations/Mutant vs Mutants/Function

```
1:                   333
2:   almost 110_000
3:
```

Codosaur.us

@davearonson

. . . three we'd still have . . .



Mutations/Mutant vs Mutants/Function

```
1:              333
2:   almost 110_000
3: over 35_000_000
```

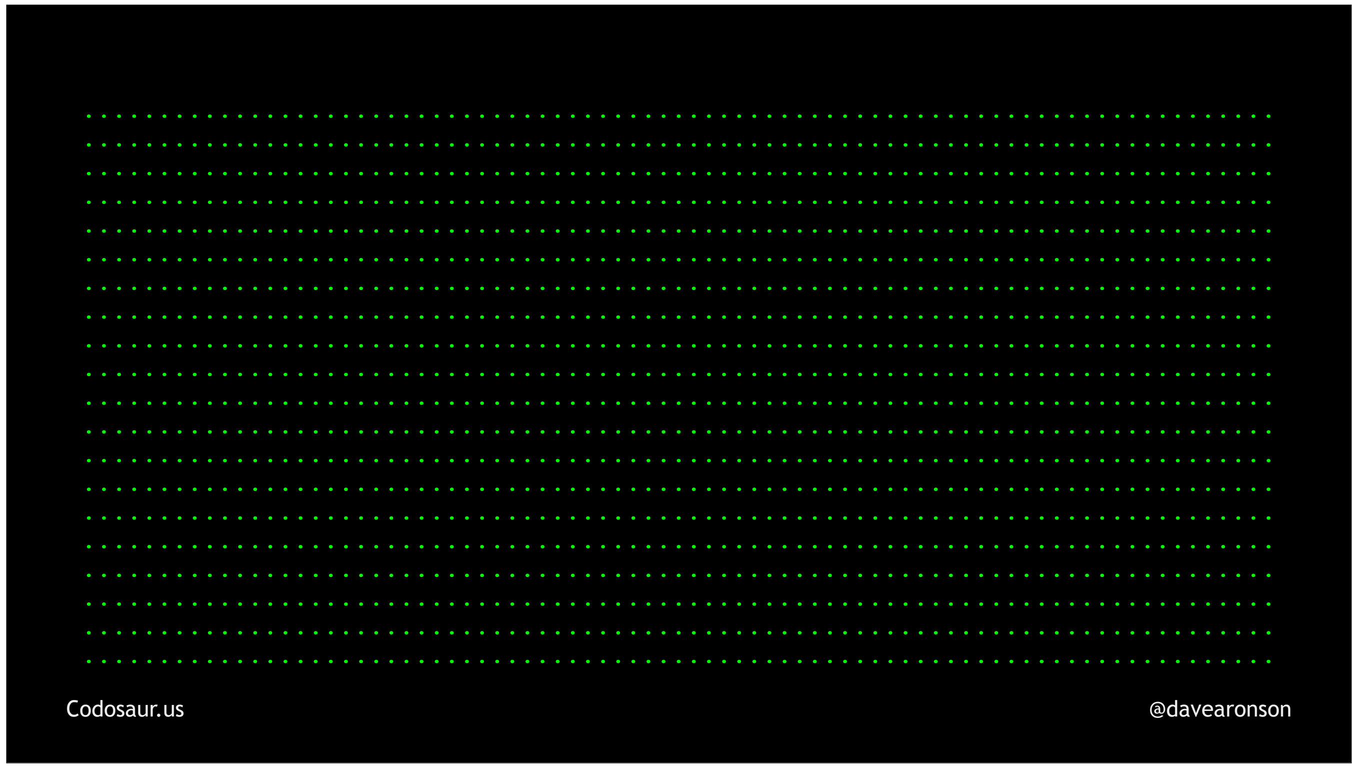
Codosaur.us

@davearonson

. . . over 35 million! Never mind actually *running* the *tests*, just *creating* the *mutants* would get to be quite a heavy workload! But we can avoid this huge workload, *and* the increased false alarms, *and* the lack of focus, if we just . . .

. . . limit it to one mutation per mutant.

The next question is: this sounds like mutation testing only makes sure that our . . .



```
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
```

Codosaur.us @davearonson

. . . test *suite* as a *whole* is strict. Is there any way it can help us assess the quality of . . .

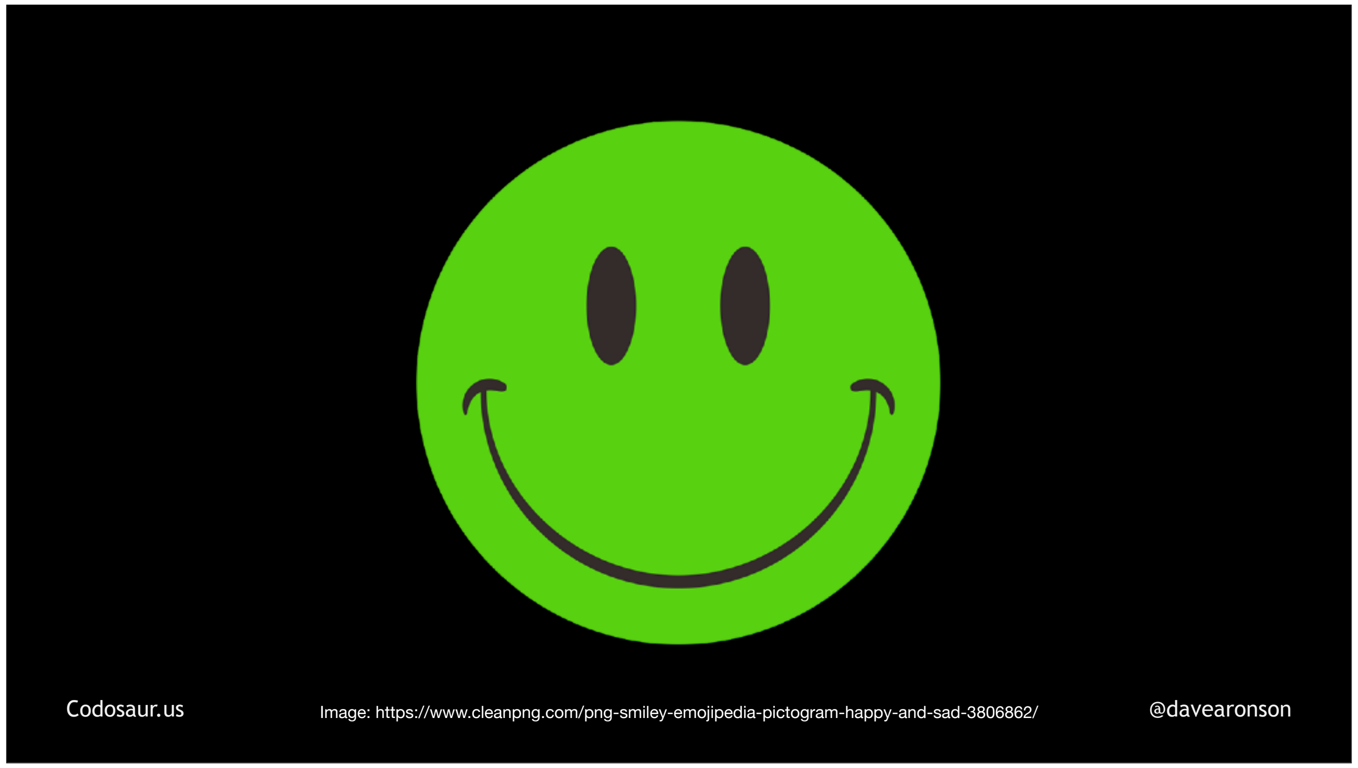
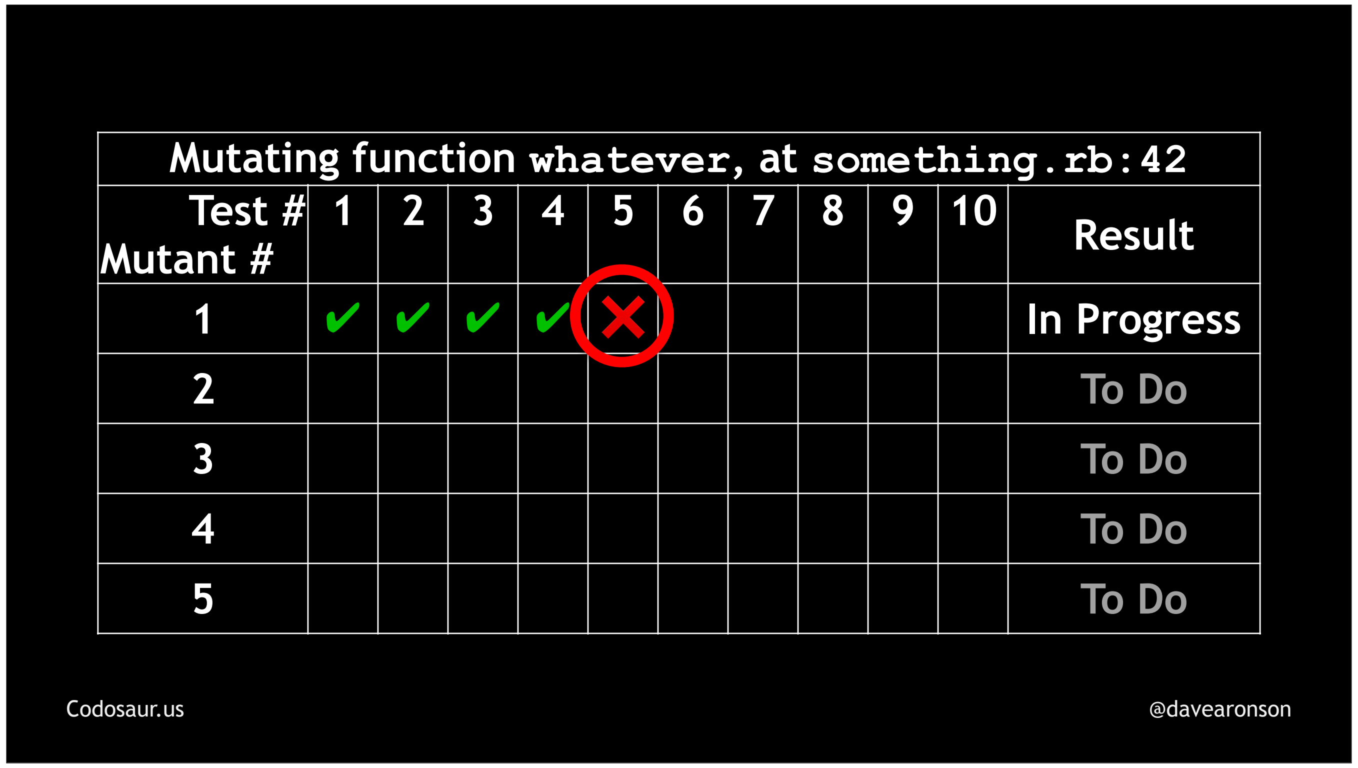

Codosaur.us

Image: https://www.cleanpng.com/png-smiley-emojipedia-pictogram-happy-and-sad-3806862/

@davearonson

. . . *individual* tests?

Yes there is, but it would take a lot longer. You may remember how I said early on, that when . . .

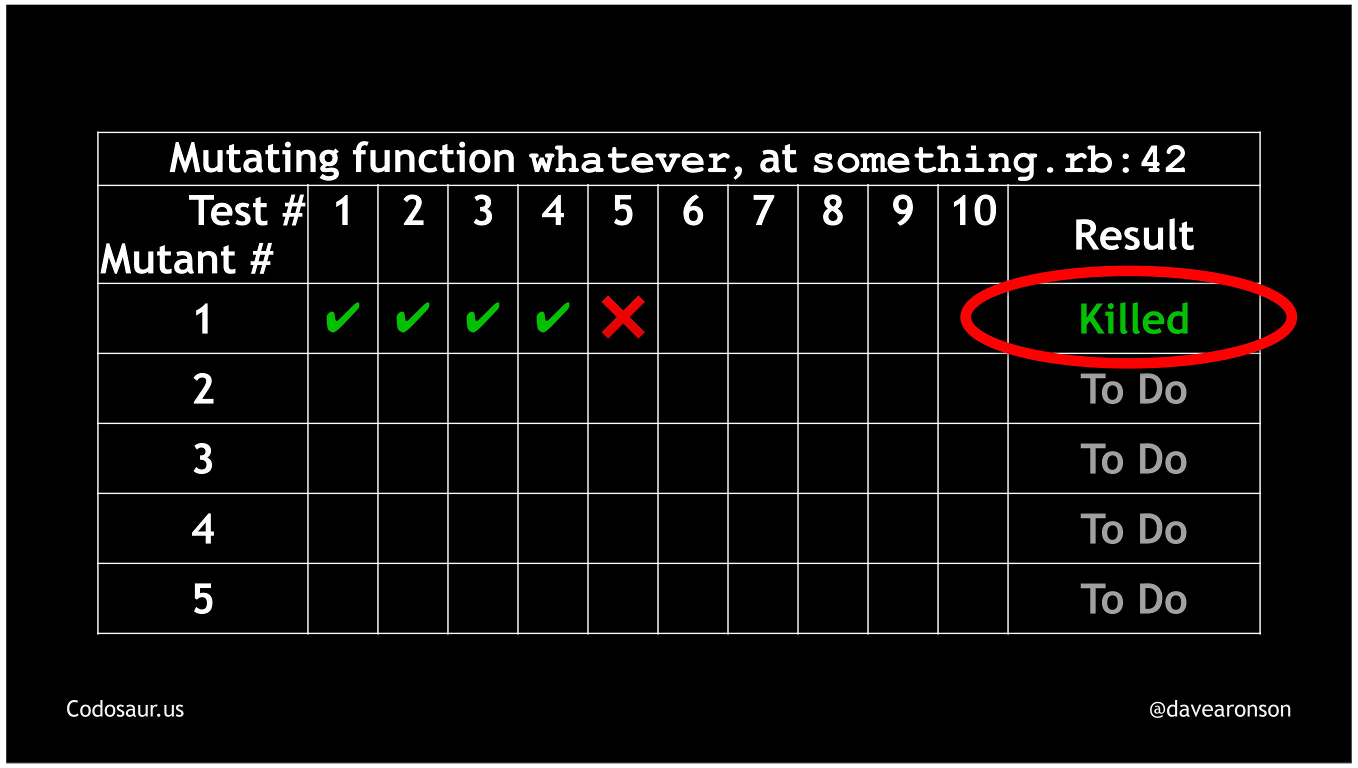


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

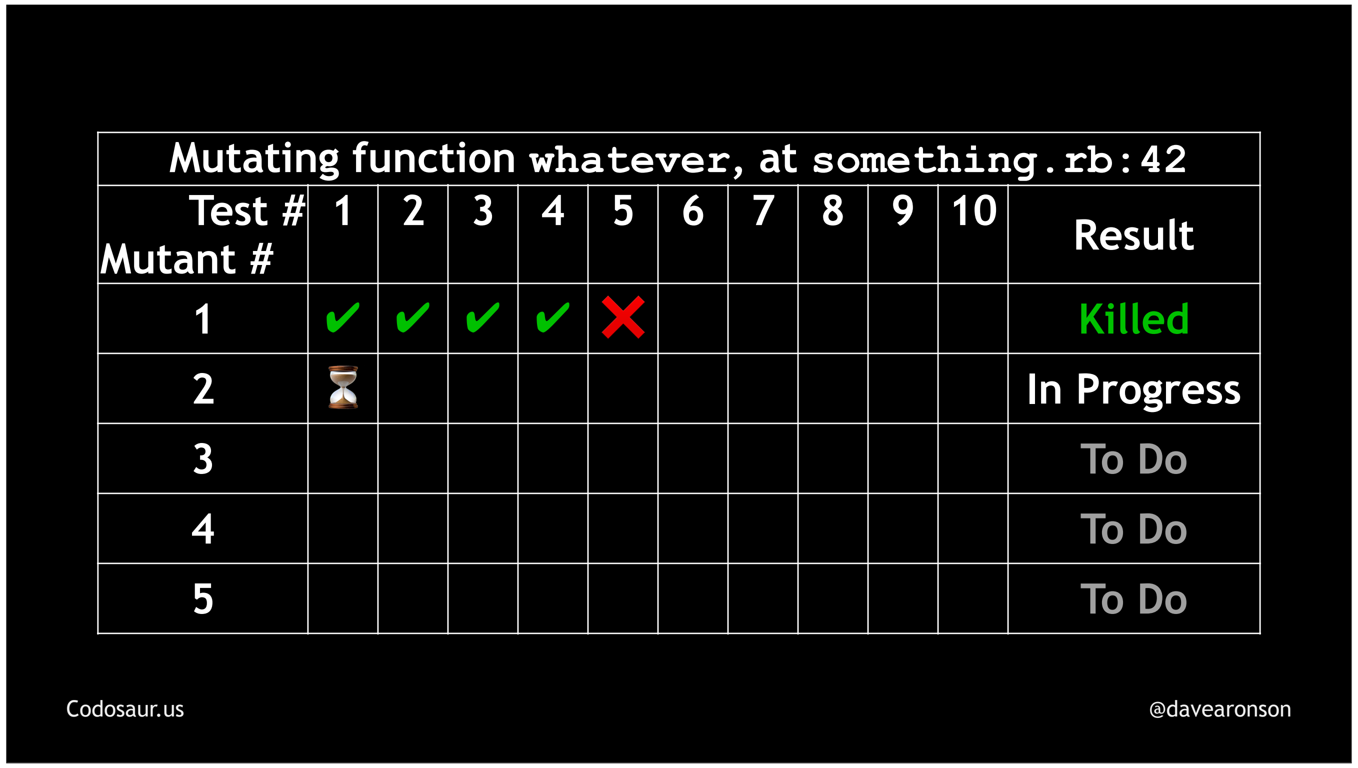
@davearonson

. . . a mutant makes a test fail, the tool will . . .



Mutating function `whatever`, at `something.rb: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, stop running any more tests against it, and . . .

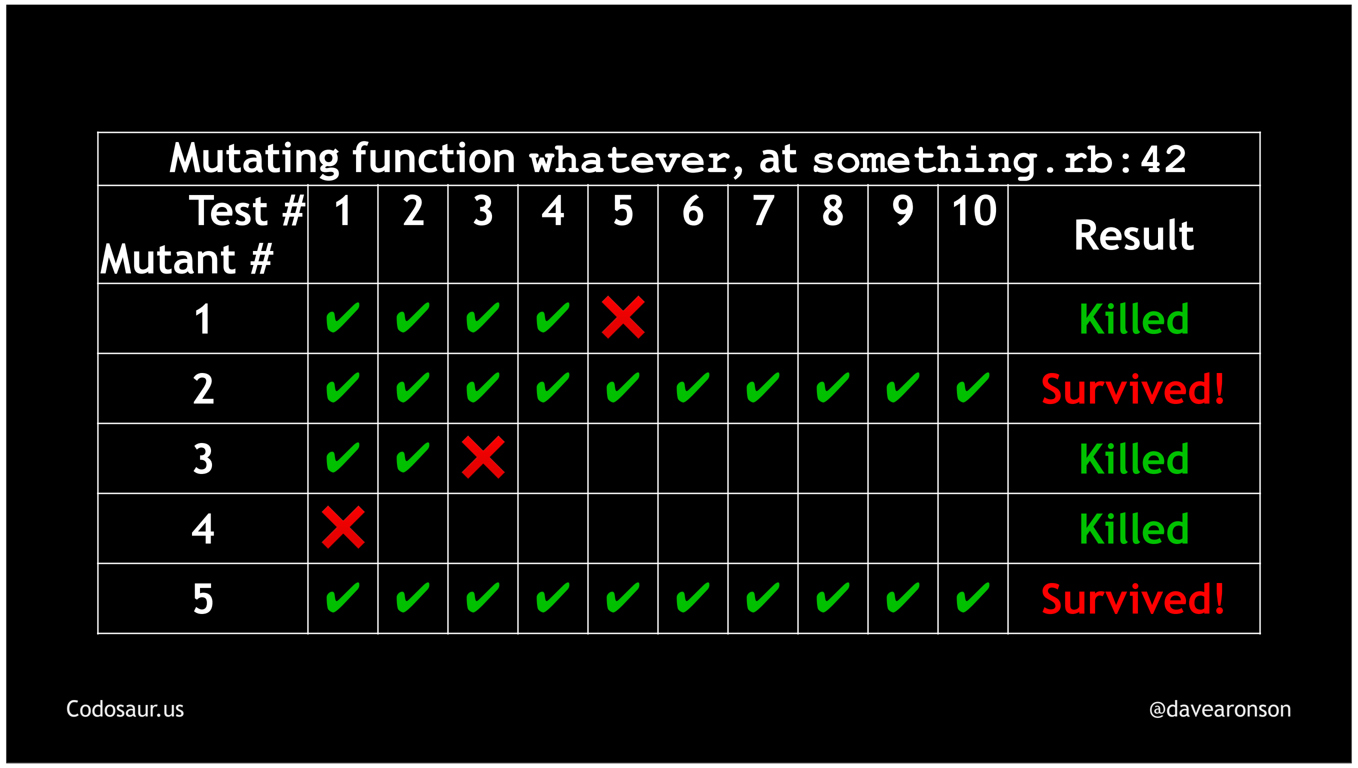


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗						Killed
2	⌛										In Progress
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

. . . move on to the next one. So when we're done with a given function, we wind up with a chart like . . .

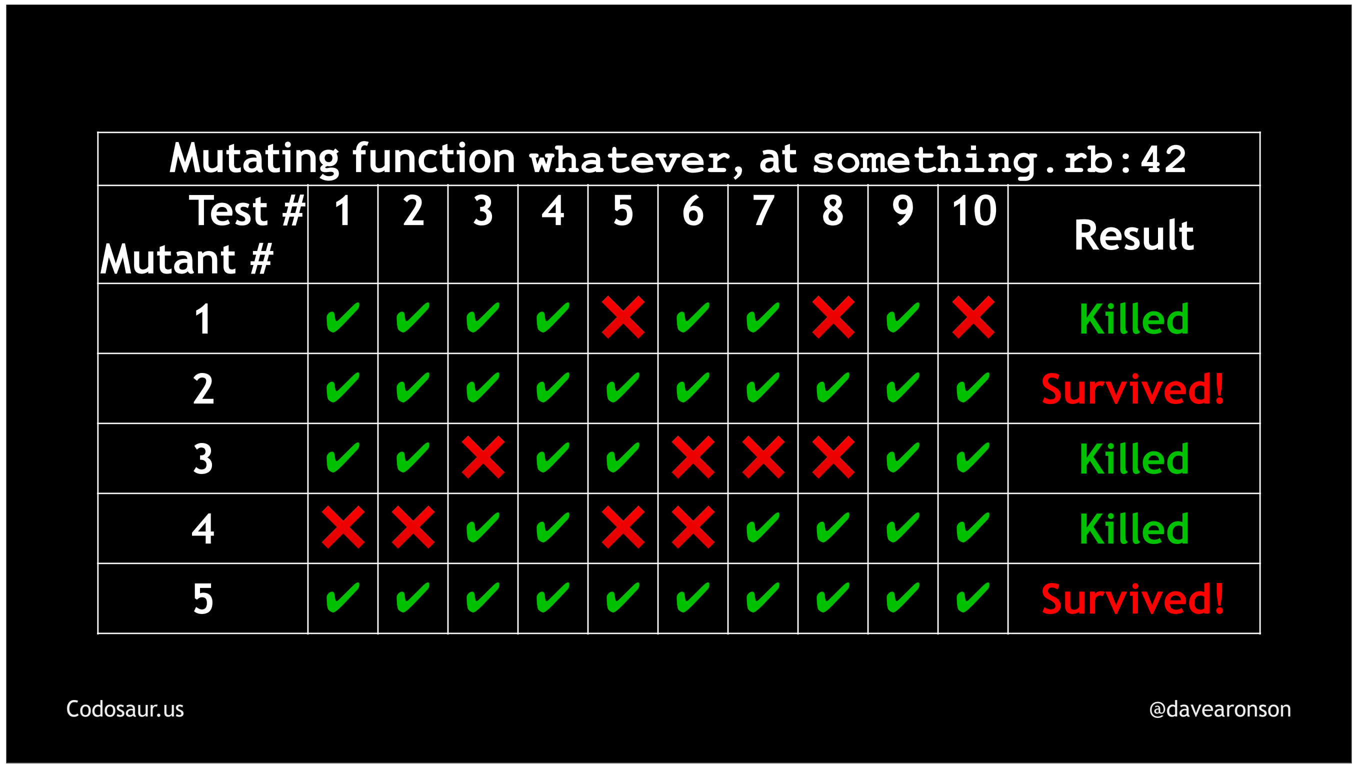


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗						Killed
2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Survived!
3	✓	✓	✗								Killed
4	✗										Killed
5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Survived!

Codosaur.us

@davearonson

. . . this. If we were to run the rest of the tests, that would take a lot longer, but it would give us . . .

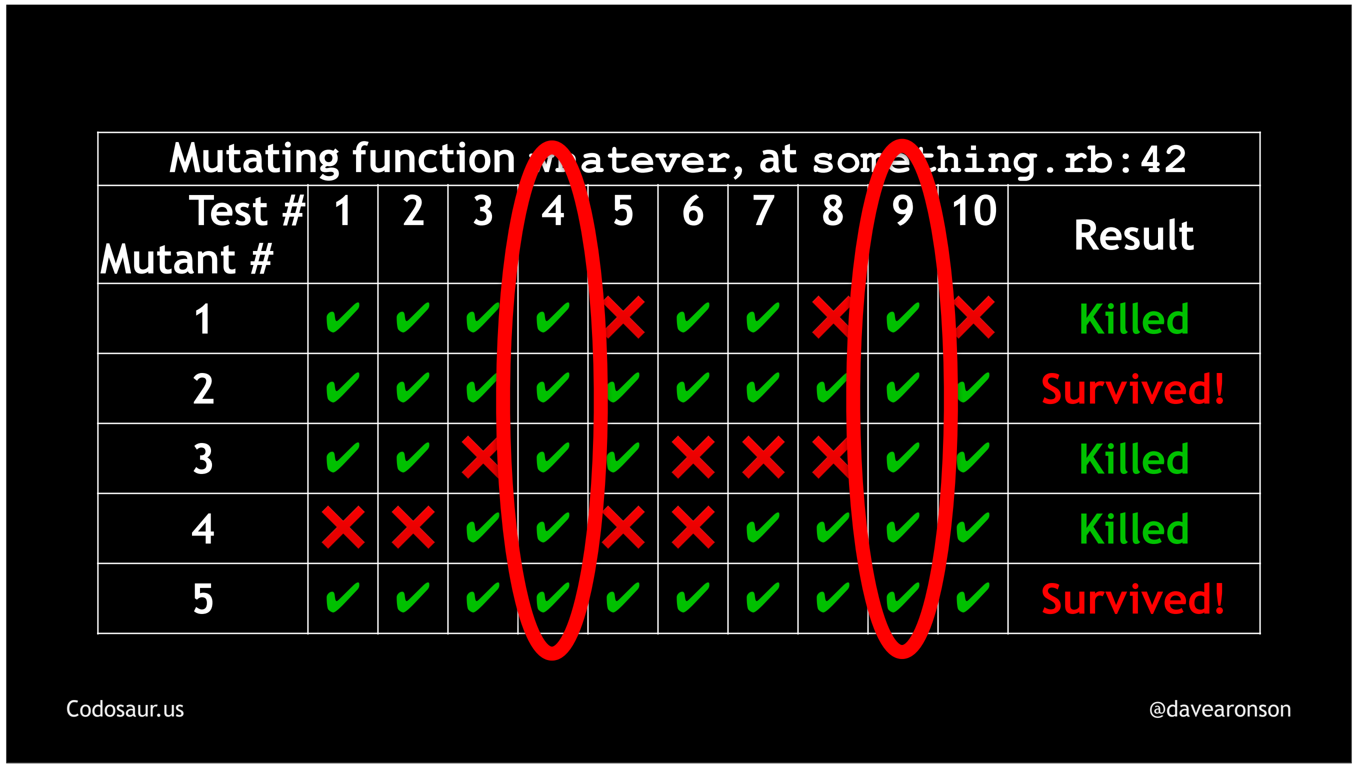


Mutating function `whatever`, at `something.rb:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘	✔	✔	✘	✔	✘	Killed
2	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	Survived!
3	✔	✔	✘	✔	✔	✘	✘	✘	✔	✔	Killed
4	✘	✘	✔	✔	✘	✘	✔	✔	✔	✔	Killed
5	✔	✔	✔	✔	✔	✔	✔	✔	✔	✔	Survived!

Codosaur.us

@davearonson

. . . some useful information, that we can use to assess the quality of *some* individual tests. I don't know for sure of any tools that will do this, but I think Muzak Pro, for Elixir will, and I was recently told that Stryker for JavaScript, not for either of the other languages it does, will too. Look at . . .

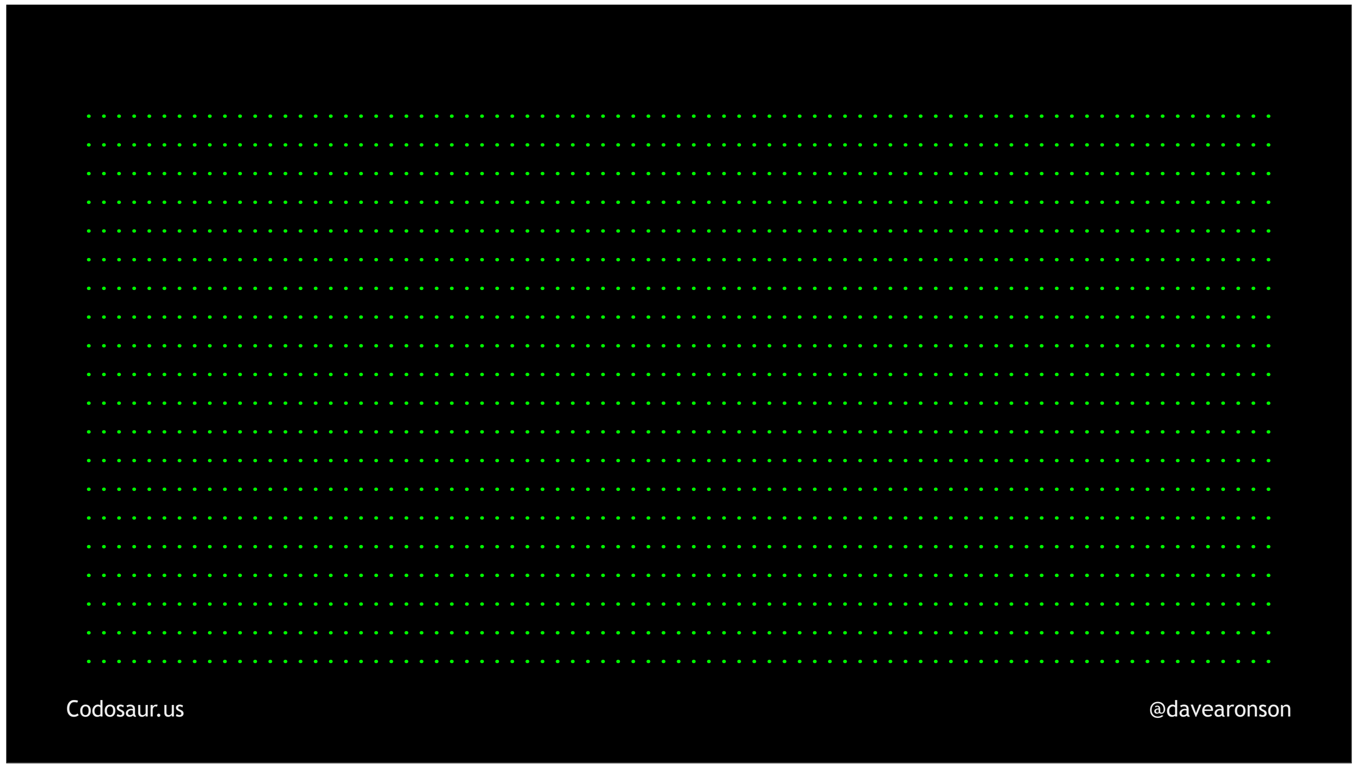


Mutating function whatever, at something.rb:42											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✓	✓	✓	✓	✗	✓	✓	✗	✓	✗	Killed
2	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Survived!
3	✓	✓	✗	✓	✓	✗	✗	✗	✓	✓	Killed
4	✗	✗	✓	✓	✗	✗	✓	✓	✓	✓	Killed
5	✓	✓	✓	✓	✓	✓	✓	✓	✓	✓	Survived!

Codosaur.us @davearonson

. . . tests four and nine. *None* of the mutants make *either* of *those* tests fail! This isn't an absolute indication that they're no good, but it does mean that they may merit a closer look, somewhat like a code smell. However, remember, the tools don't actually give us a chart, it's just a conceptual model I'm using to illustrate the situation. If you find one that gives you this full of a report, *and* lets you run all applicable tests against all mutants, you could do this. In fact, you could even take this concept a step further and look *next* at those that only stop *one* mutant, then *two*, and so on, but I think it would rapidly reach a point of diminishing returns, probably at one.

The last question is: as mentioned earlier, mutation testing *assumes* that we have . . .

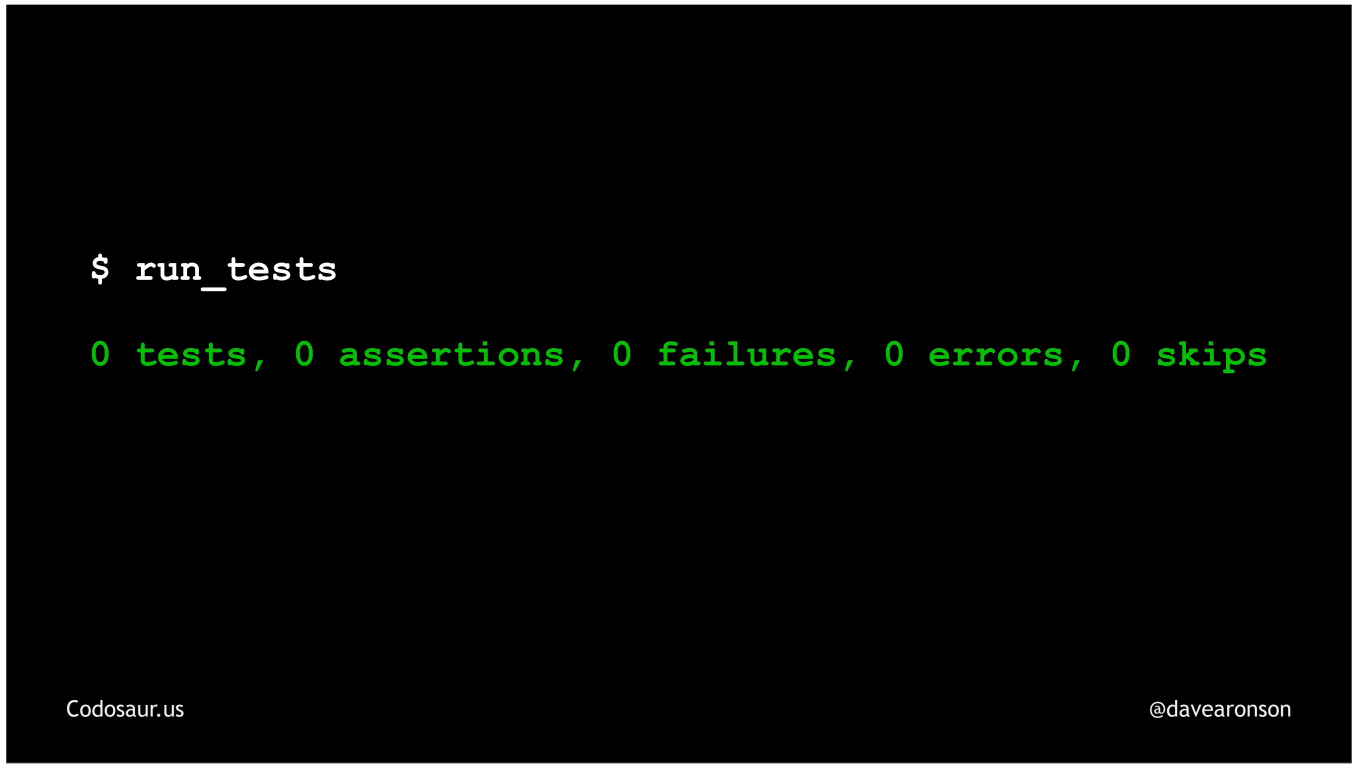


```
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
...............................................................................
```

Codosaur.us

@davearonson

. . . tests already. What if . . .



```
$ run_tests

0 tests, 0 assertions, 0 failures, 0 errors, 0 skips
```

Codosaur.us

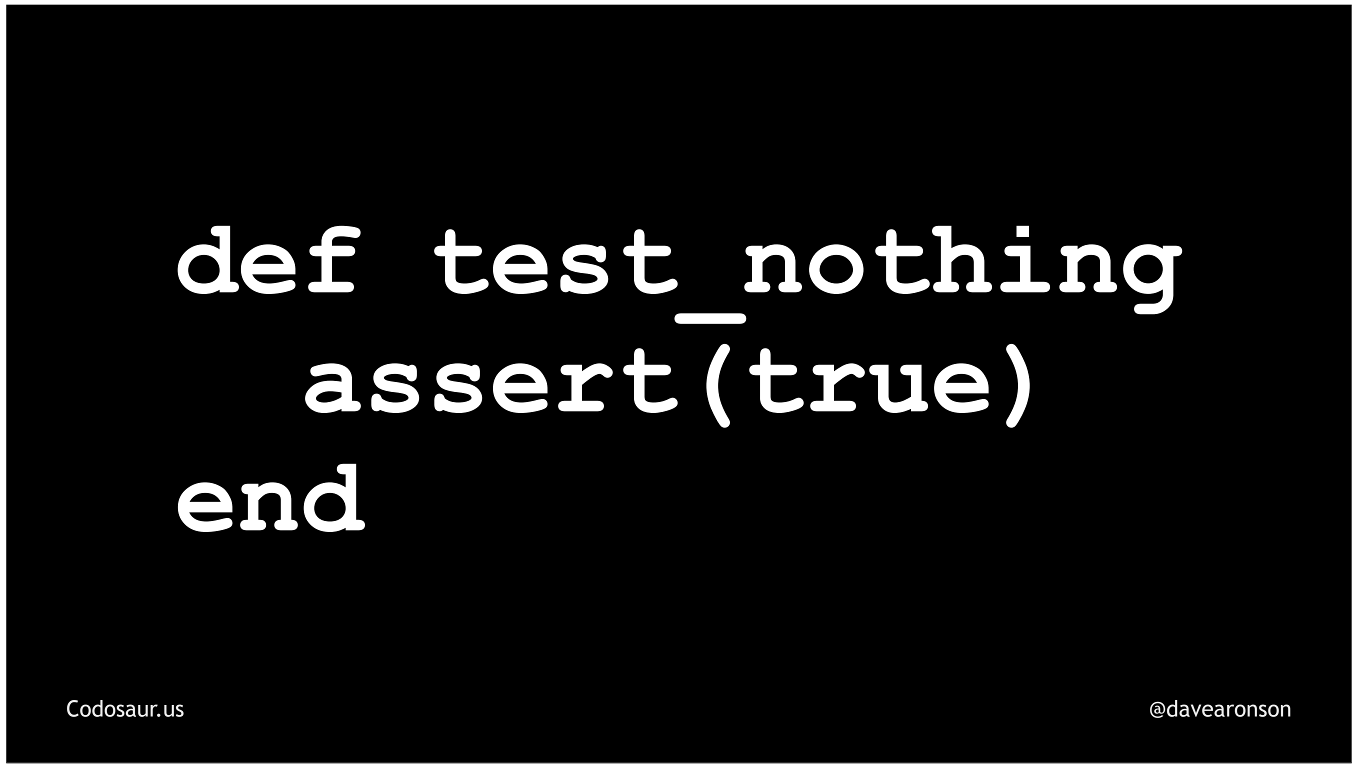
@davearonson

. . . we don't? Can mutation testing be of any help in *that* case?

Well, first of all, whoever wrote a substantial production codebase with no tests needs some educating about the value of tests. But yes, mutation testing can help you . . .

Image: https://www.pxfuel.com/en/free-photo-qzzxl

. . . *build* your test suite in the first place! You can start with a . . .



```
def test_nothing
  assert(true)
end
```

Codosaur.us @davearonson

. . . meaningless test, and run your mutation testing tool. You'll probably get a . . .

Image: https://commons.wikimedia.org/wiki/File:The_mutants!_(9363719694).jpg

. . . lot of mutants, including many that are essentially duplicates, telling about the same problem. Out of the mutants for each function, . . .

Codosaur.us
Image: https://pixabay.com/photos/mutant-daisy-flower-bloom-macro-593712/
@davearonson

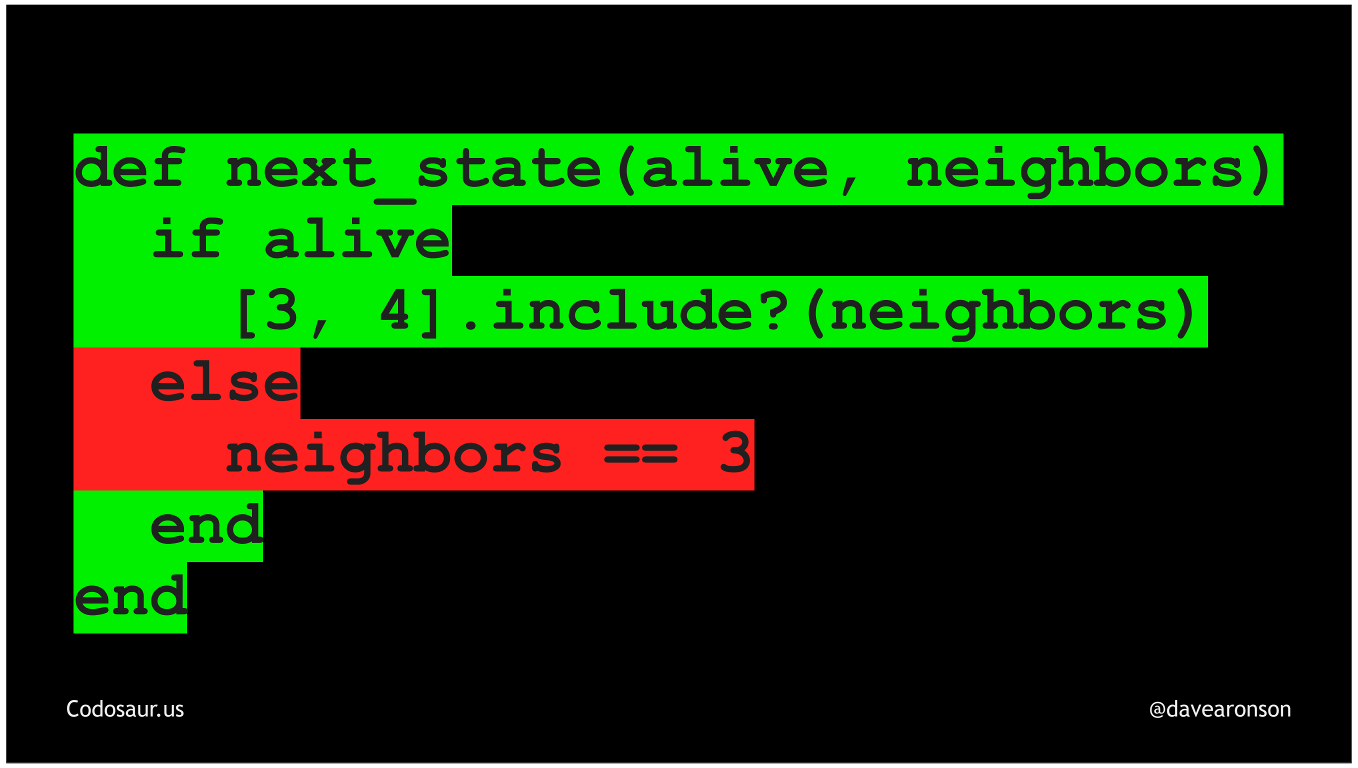
. . . pick one. You can just pick it randomly, no need to overthink it. Try to kill the mutant, by adding one test. This will probably kill many other mutants as well. Then lather, rinse, repeat, though on further iterations you might *improve* a test rather than *add* any. Now, this won't . . .

Codosaur.us

Image: https://pixabay.com/illustrations/satisfaction-guarantee-2109235/

@davearonson

. . . *guarantee* that you wind up with a great test suite. Many mutation testing tools don't do anywhere near all the possible mutations I showed you earlier, so a lot of code will probably remain . . .



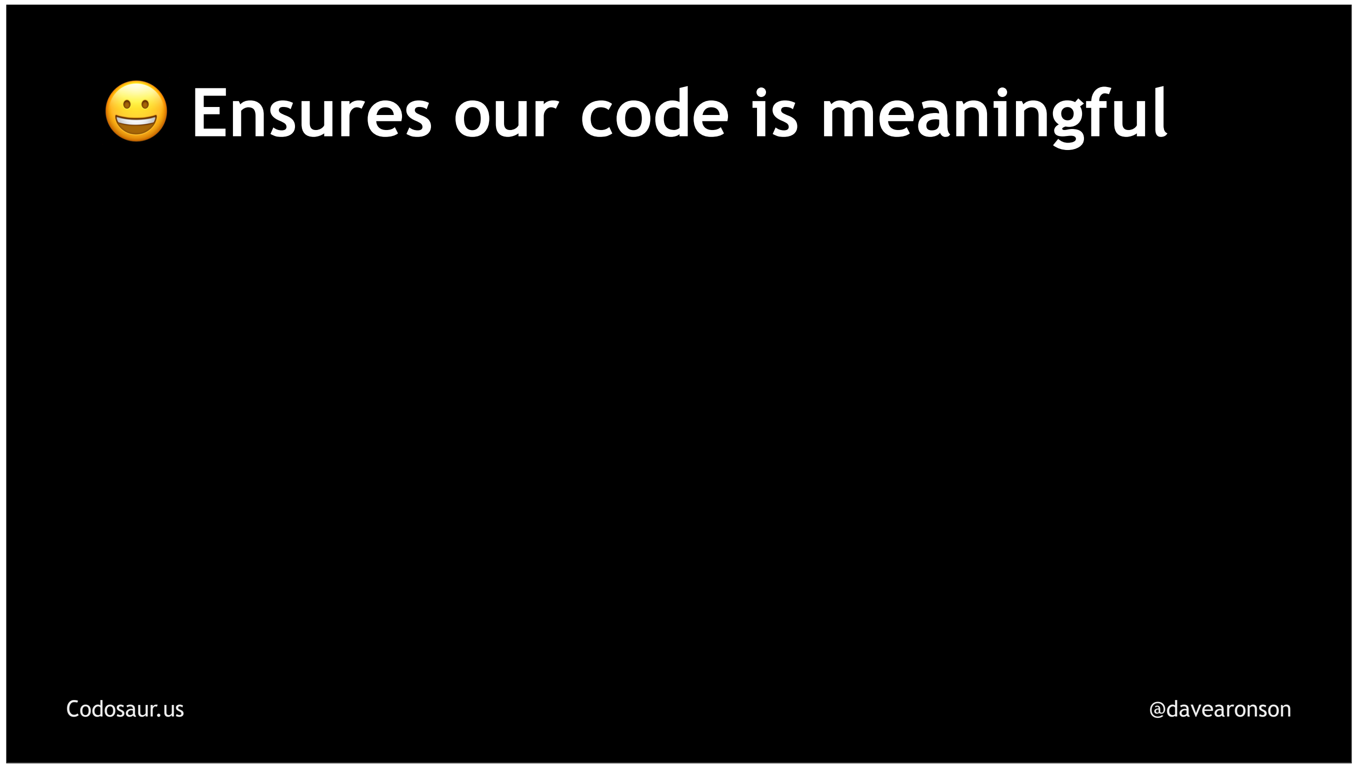
```
def next_state(alive, neighbors)
  if alive
    [3, 4].include?(neighbors)
  else
    neighbors == 3
  end
end
```

Codosaur.us

@davearonson

. . . untested, if not in statements, like this, then at least in semantics. However, this idea will get you off to a decent start. Then you can look at what code is untested, and write more tests to fill in the holes.

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

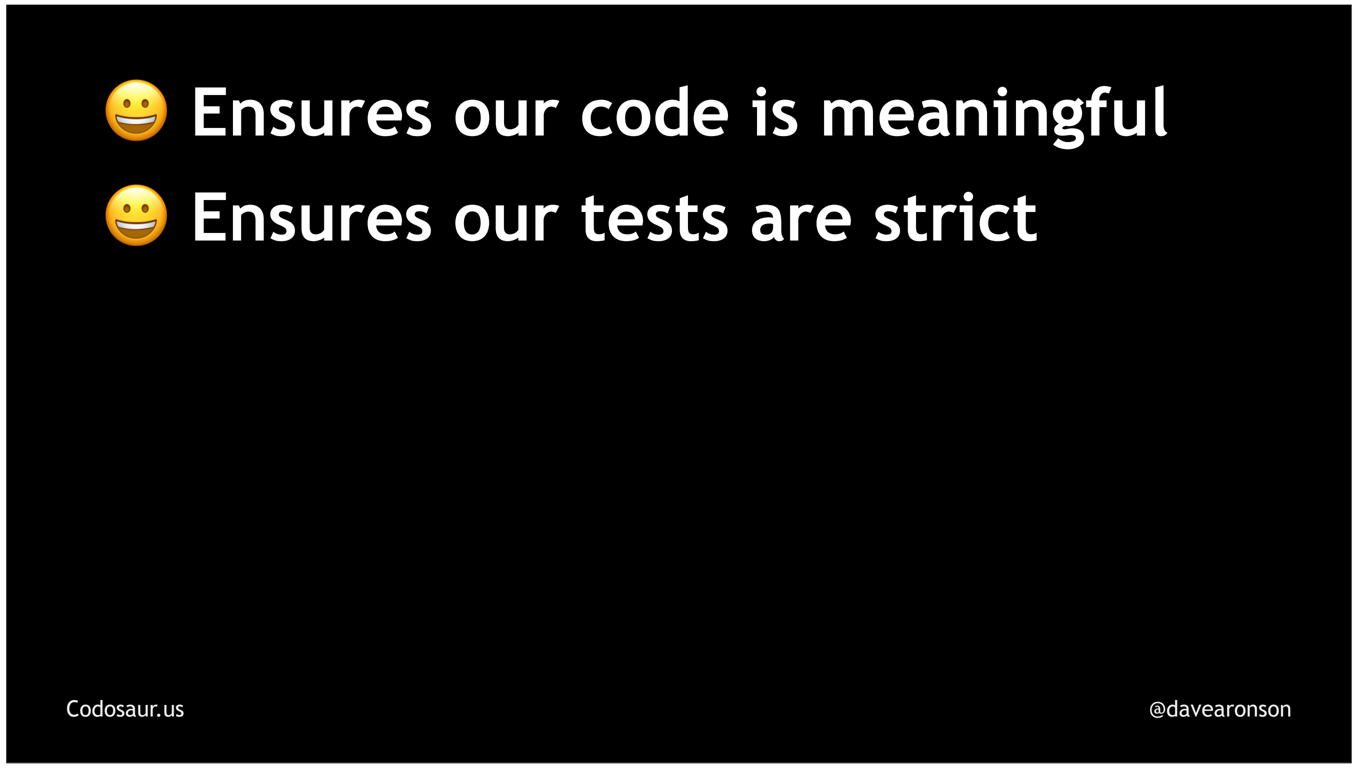


😀 Ensures our code is meaningful

Codosaur.us

@davearonson

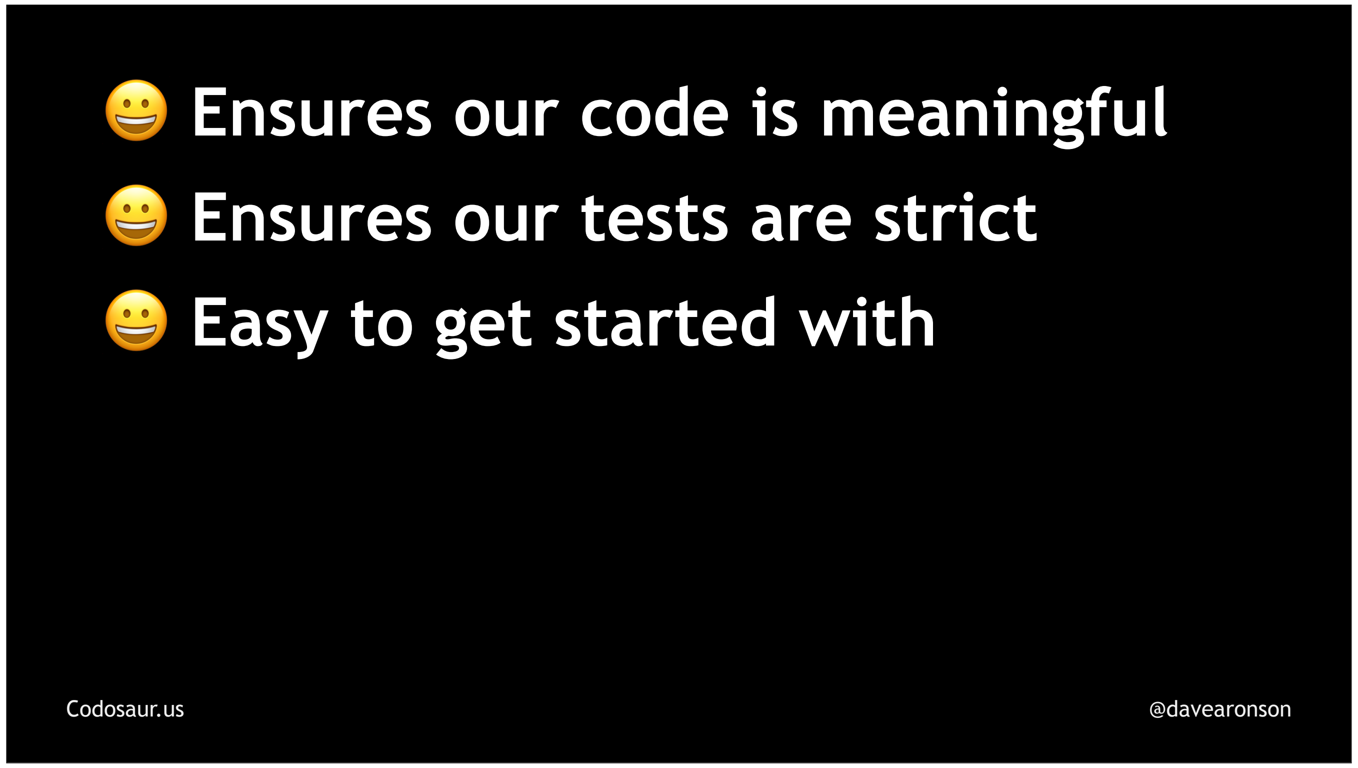
. . . ensure that our code is meaningful and . . .


😀 Ensures our code is meaningful

😀 Ensures our tests are strict

Codosaur.us @davearonson

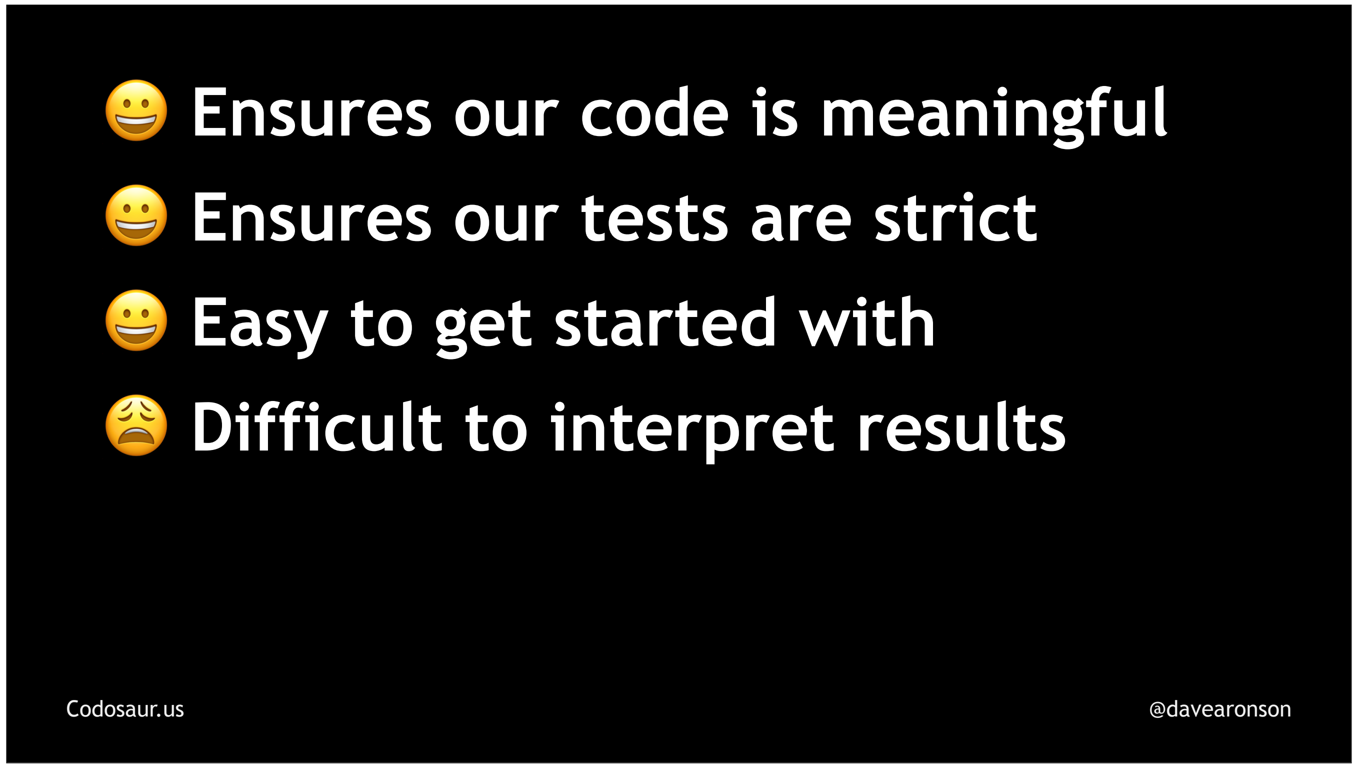
. . . our tests are strict. It's . . .


😀 Ensures our code is meaningful

😀 Ensures our tests are strict

😀 Easy to get started with

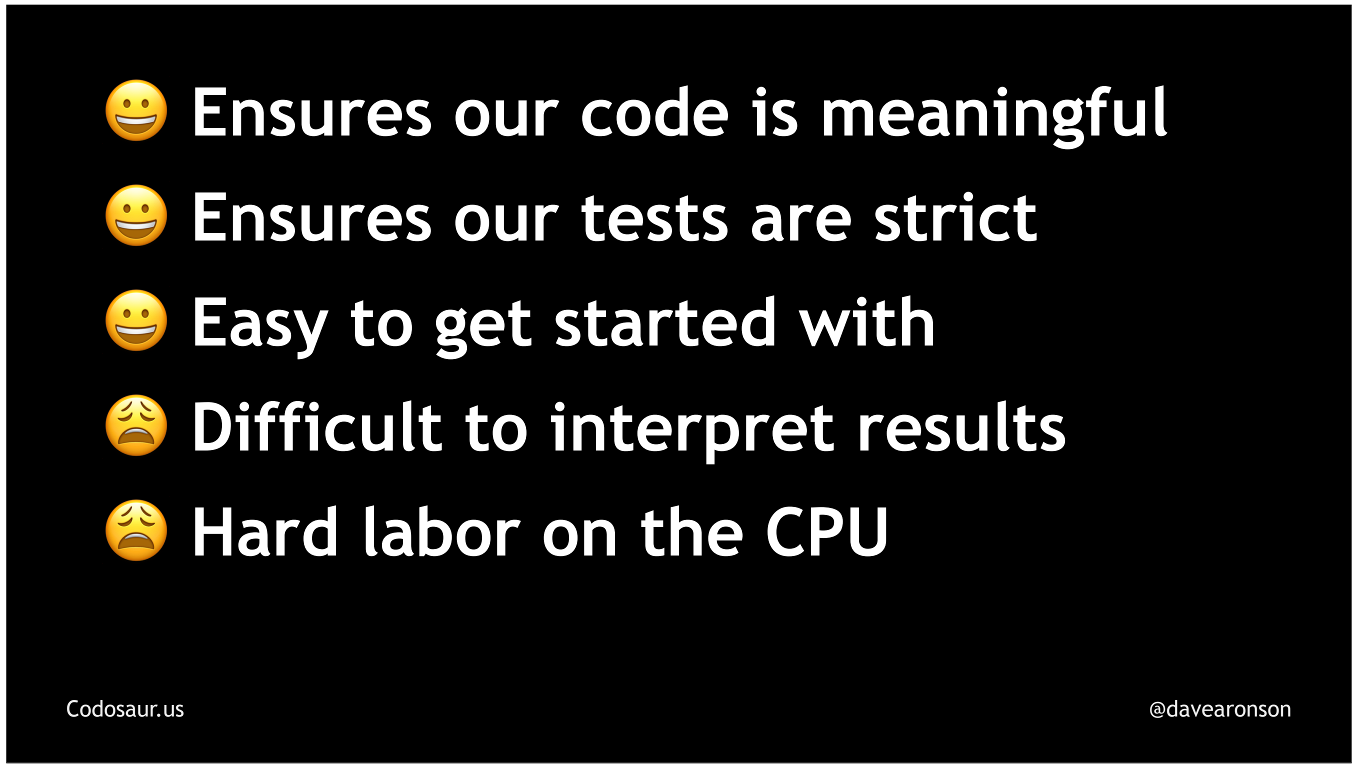
easy to get started with, in terms of setting up most of the tools and annotating our tests if needed
(which may be *tedious* but at least it's *easy*),
but it's . . .


- 😀 Ensures our code is meaningful
- 😀 Ensures our 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 . . .

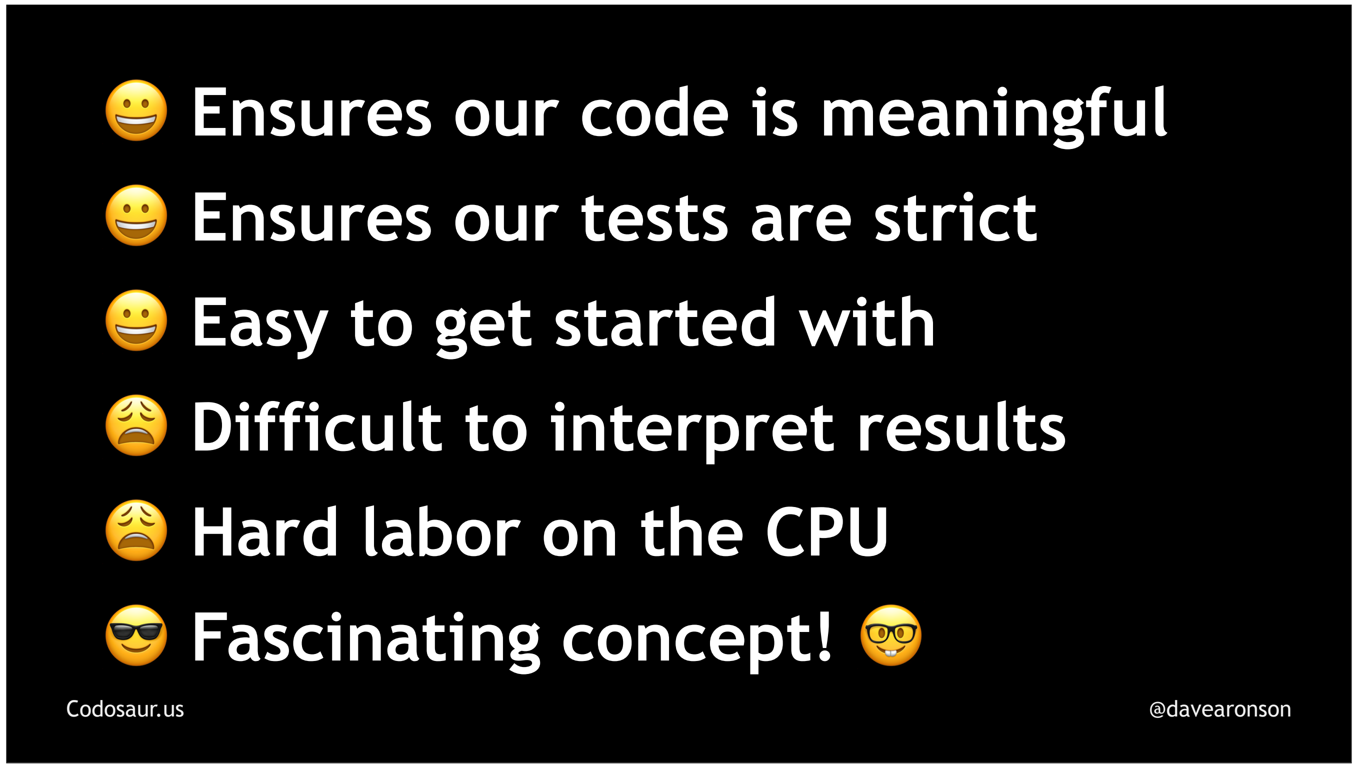


- 😀 Ensures our code is meaningful
- 😀 Ensures our tests are strict
- 😀 Easy to get started with
- 😩 Difficult to interpret results
- 😩 Hard labor on the CPU

Codosaur.us

@davearonson

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

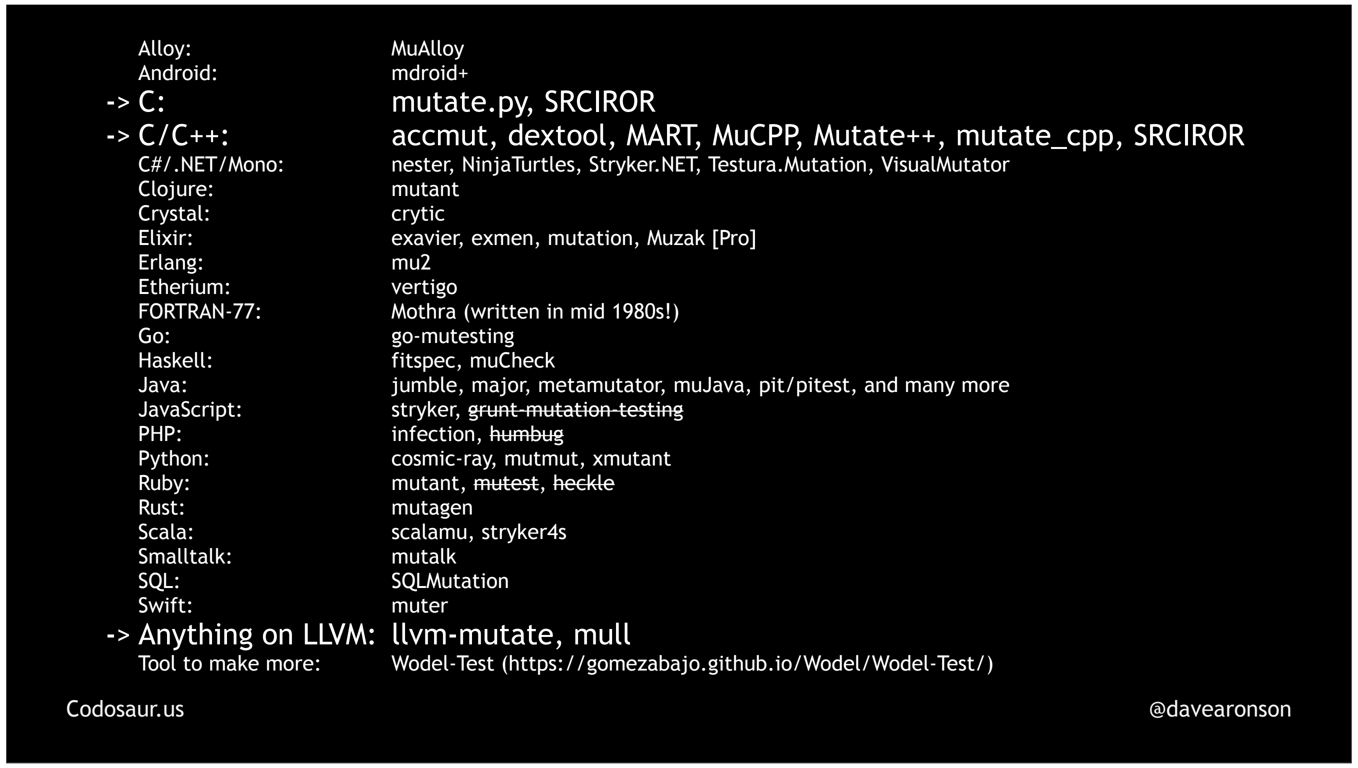


- 😀 Ensures our code is meaningful
- 😀 Ensures our tests are strict
- 😀 Easy to get started with
- 😩 Difficult to interpret results
- 😩 Hard labor on the CPU
- 😎 Fascinating concept! 🤓

Codosaur.us @davearonson

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

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



Alloy:	MuAlloy
Android:	mdroid+
-> C:	mutate.py, SRCIROR
-> C/C++:	accmut, dextool, MART, MuCPP, Mutate++, mutate_cpp, SRCIROR
C#/.NET/Mono:	nester, NinjaTurtles, Stryker.NET, Testura.Mutation, VisualMutator
Clojure:	mutant
Crystal:	crytic
Elixir:	exavier, exmen, mutation, Muzak [Pro]
Erlang:	mu2
Etherium:	vertigo
FORTRAN-77:	Mothra (written in mid 1980s!)
Go:	go-mutesting
Haskell:	fitspec, muCheck
Java:	jumble, major, metamutator, muJava, pit/pitest, and many more
JavaScript:	stryker, ~~grunt-mutation-testing~~
PHP:	infection, ~~humbug~~
Python:	cosmic-ray, mutmut, xmutant
Ruby:	mutant, ~~mutest~~, ~~heckle~~
Rust:	mutagen
Scala:	scalamu, stryker4s
Smalltalk:	mutalk
SQL:	SQLMutation
Swift:	muter
-> Anything on LLVM:	llvm-mutate, mull
Tool to make more:	Wodel-Test (https://gomezabajo.github.io/Wodel/Wodel-Test/)

Codosaur.us

@davearonson

. . . 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. I'll talk a bit so you can take pictures. Just be aware that many of these are outdated; I don't know or follow quite *all* of these languages and platforms. The ones I *know* are outdated, are crossed out. There's also a promising tool called Wodel-Test, a *language-independent* mutation engine, with which you can make language-specific mutation testing tools. (Sorry, I haven't looked into it much myself.)

FOR C/C++ CONFS: For plain old C, there is mutate.py, yes it's written in Python but it's for mutation-testing C code. For both C *and* C++, there are accmut, dextool, MART, MuCPP, Mutate++, mutate_cpp, and SRCIROR, in just alphabetical order, nothing else implied there. And if you're using the LLVM toolchain, you can also use llvm-mutate, or mull.

Is everybody done taking pictures? Before we get to Q&A, I'd like to give a shoutout to . . .

Thanks to Toptal and their Speakers Network!

https://toptal.com/#accept-only-candid-coders

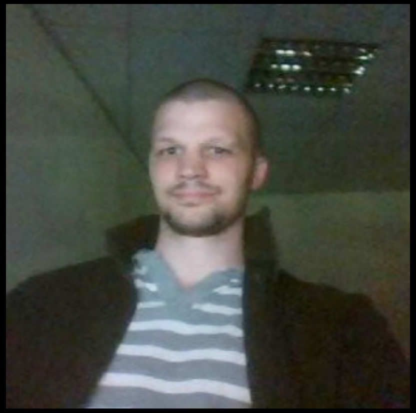
Codosaur.us

Images: Toptal logo, used by permission; QR code for my referral link

@davearonson

. . . Toptal, a consulting network I'm in, whose Speakers Network helped me prepare and practice previous versions of this presentation. (Please use that referral link if you want to hire us or join us, and that's also where that QR code goes.)

Also, many thanks to . . .



Thank you Markus Schirp!

https://github.com/mbj

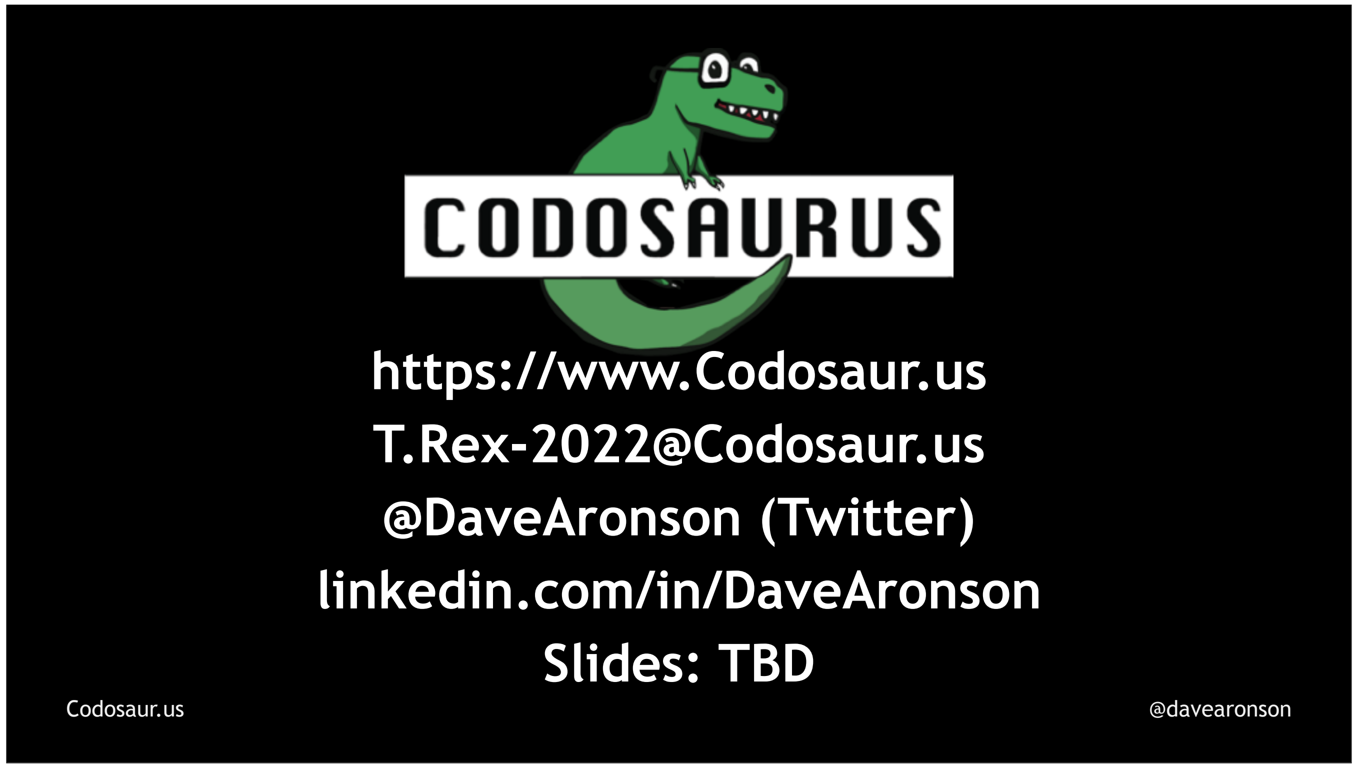
Codosaur.us

Images: Markus, from his Github profile

@davearonson

. . . Markus Schirp, who created mutant, the main mutation testing tool I've actually used, for Ruby. He has also been very willing to answer my ignorant questions and critique the original version of this presentation.

And now, . . .



https://www.Codosaur.us

T.Rex-2022@Codosaur.us

@DaveAronson (Twitter)

linkedin.com/in/DaveAronson

Slides: TBD

Codosaur.us @davearonson

. . . it's almost your turn! If you have any questions, I'll take them in just a moment. If you think of anything later, I'll be around for the rest of the conference. If it's too late by then, there's my contact information up there, plus the URL where you can get the slides, complete with script. And of course I have cards. Any questions?