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

CURRENT TIME: ~23, want ~20-25 since slot is 30 w/ 5-10 Q&A — OOPS THEY MADE IT 35, slow down or add back in material chopped out

FONT NOTE: Intro uses Optima for Serbian since the small-T character showed up on *some* slides as an m, in my usual Trebuchet font!

Kill All Mutants!

(Intro to Mutation Testing)

by Dave Aronson

Codosaur.us

@davearonson

Здраво Београде!

(Hello, Belgrade!)

Codosaur.us

Image: standard emoji

@davearonson

ZDRAvo BEH-o-GRAH-deh!

Ја сам Даве Аронсон,

(I'm Dave Aronson,)

Codosaur.us

Image: me speaking at JSConf Hawai'i 2020

@davearonson

YA sam Dave Aronson,

Тираносаурс Рекс из Кодосауруса,

(the T. Rex of Codosaurus,)

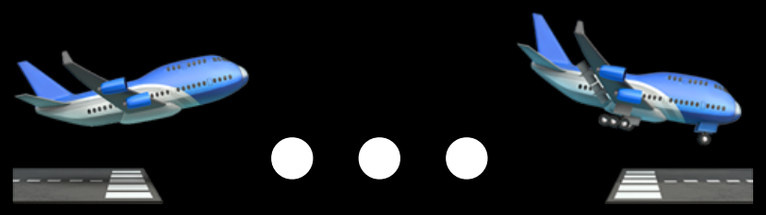
Codosaur.us

Image: my company logo!

@davearonson

Teeranosaurus Rex iz CodozowRUSa,

и долетео сам овде



(and I flew here)

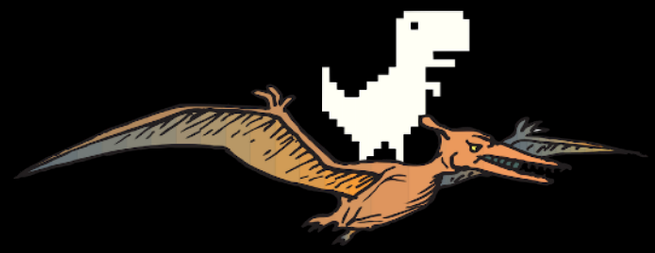
Codosaur.us

Image: standard emoji

@davearonson

ee Doh-LET-eh-o sam OV-deh

на свом птеродактилу



(on my pterodactyl)

Codosaur.us

Images: https://pixabay.com/vectors/dinosaur-tyrannosaurus-t-rex-6273164/
and https://pixabay.com/vectors/bird-flying-wings-dinosaur-ancient-44859/

@davearonson

Na svom PTE-ro-DAK-Tee-lo

да вас научим како

(to teach you)

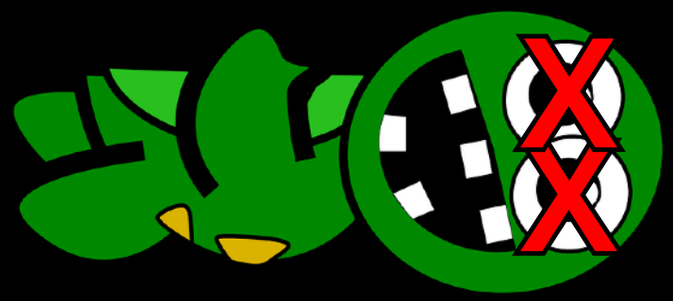
Codosaur.us

Image: standard emoji

@davearonson

Da vas NA-oo-cheem Ka-ko

да *убијате мутанте!*



(to *kill mutants!*)

Codosaur.us

Image: https://pixabay.com/vectors/turtle-tortoise-cartoon-animal-152079/

@davearonson

da *OO-bee-ya-teh MOO-Tan-teh!*

Али . . .

(But . . .)

Codosaur.us

Image: standard emoji

@davearonson

AH-lee . . .

урадићу то на енглеском.

(I will do it in English.)

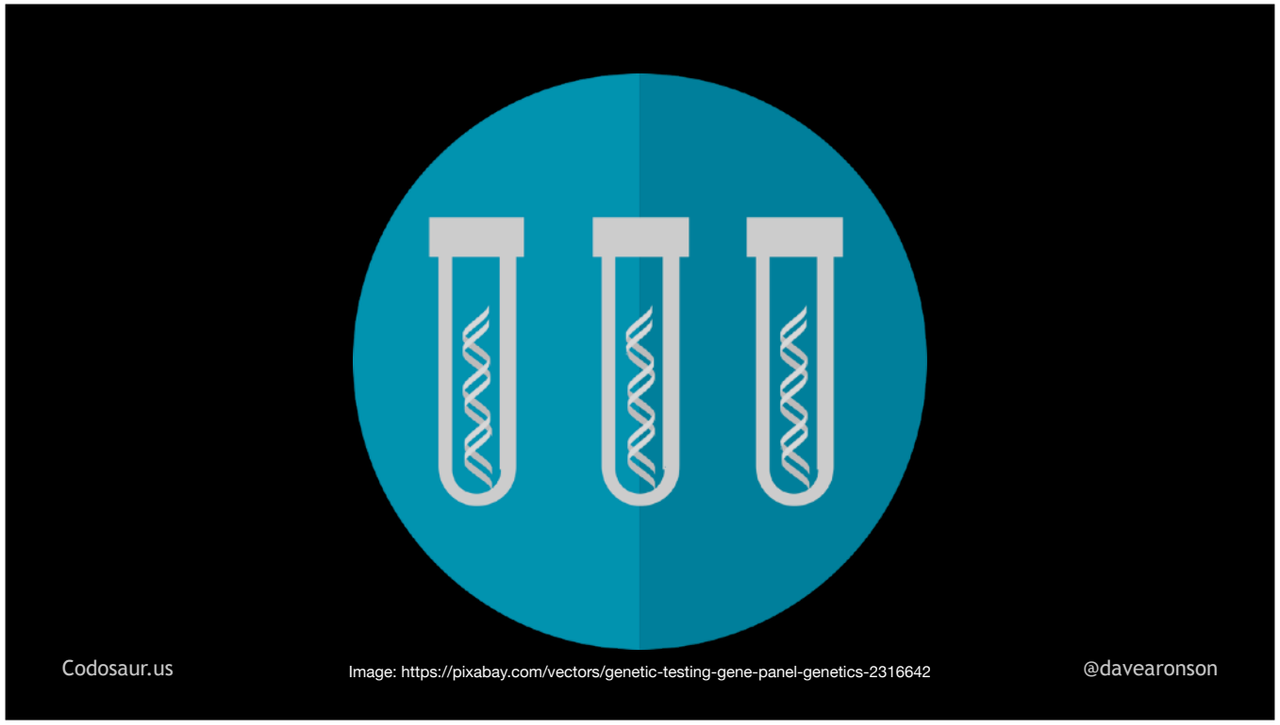
Codosaur.us

Image: standard emoji

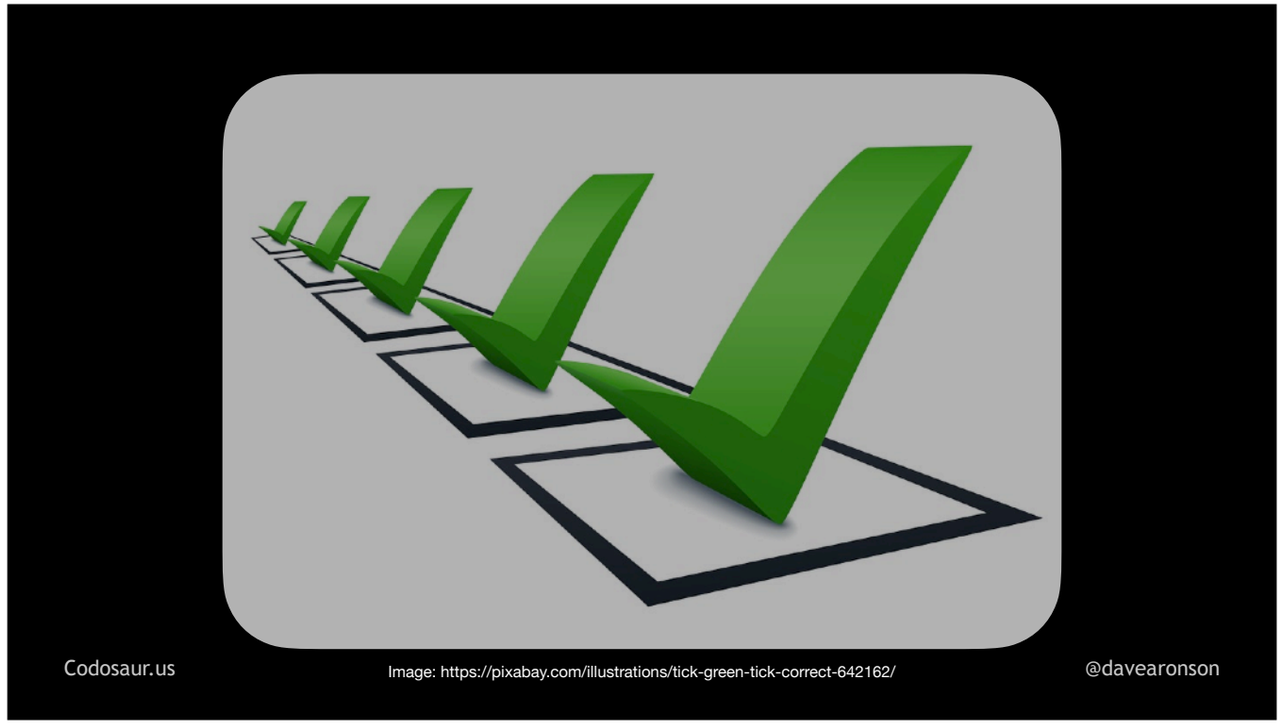
@davearonson

OO-RAH-dee-choo toh na Eng-les-kom. (PAUSE!)

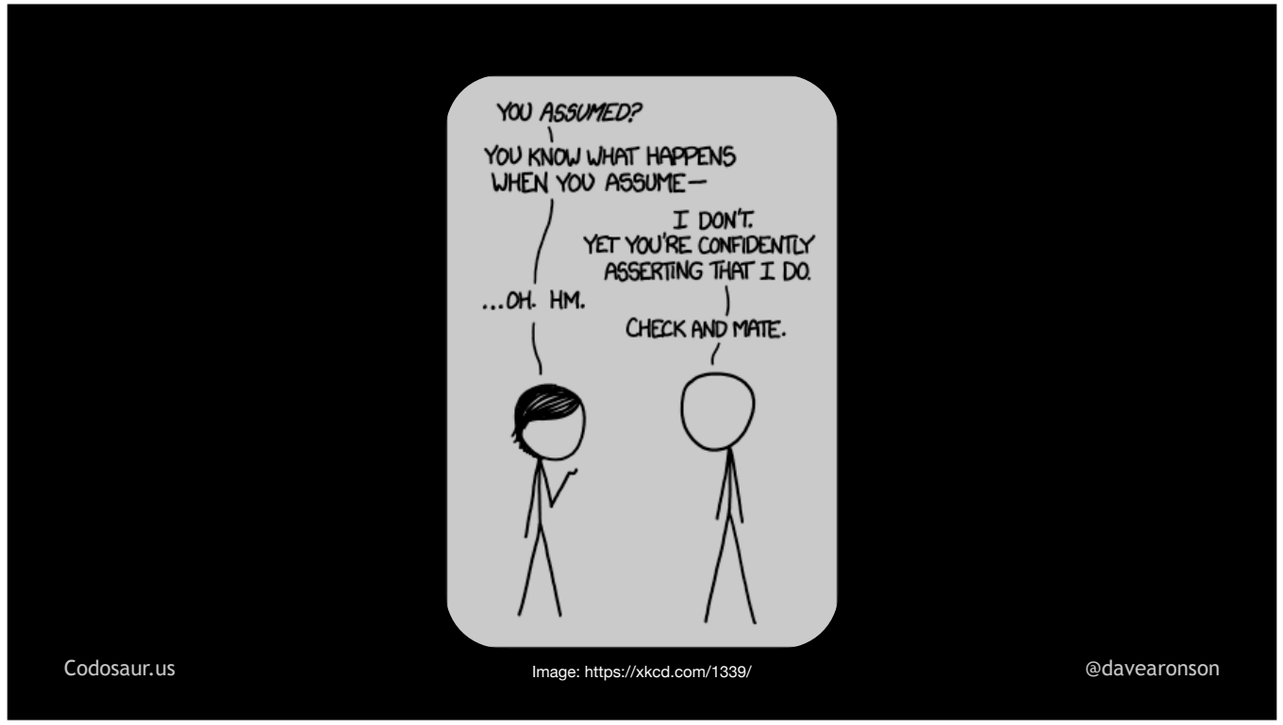
So, let’s start with the very basics. What makes . . .



. . . mutation testing different from all *other* software testing techniques? Mainly, most of the others are about . . .



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

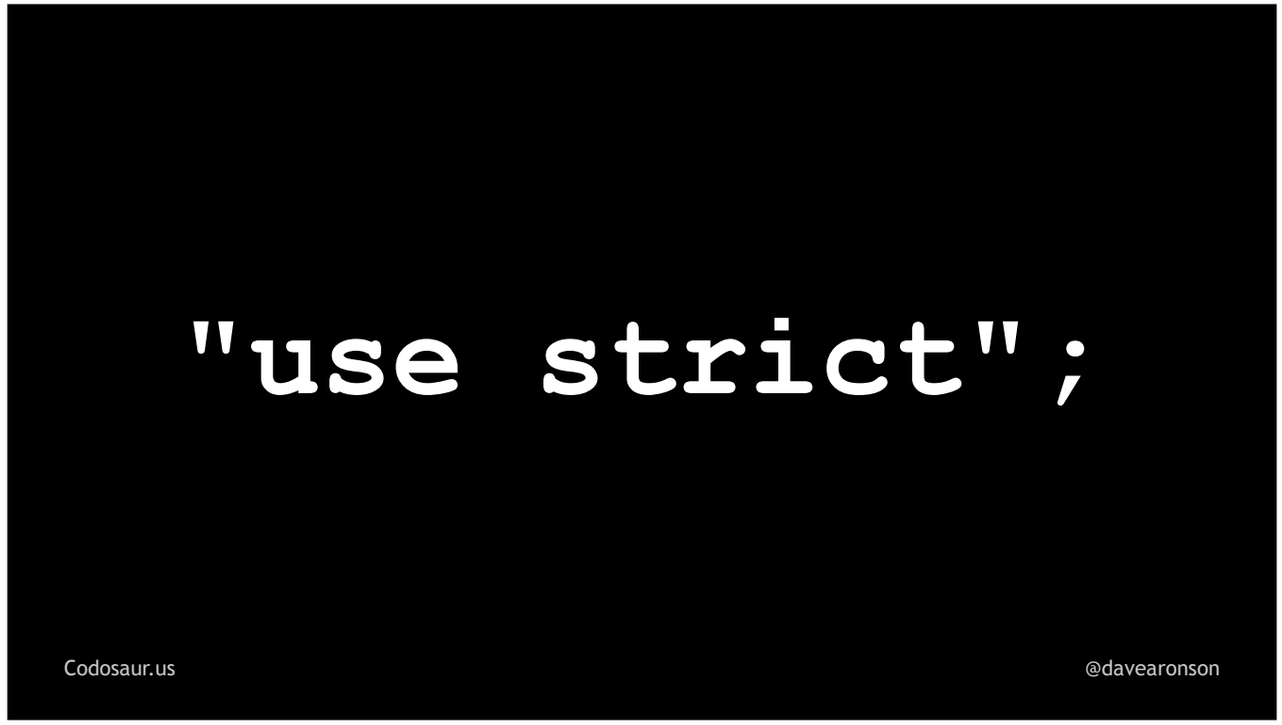


Codosaur.us

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

@davearonson

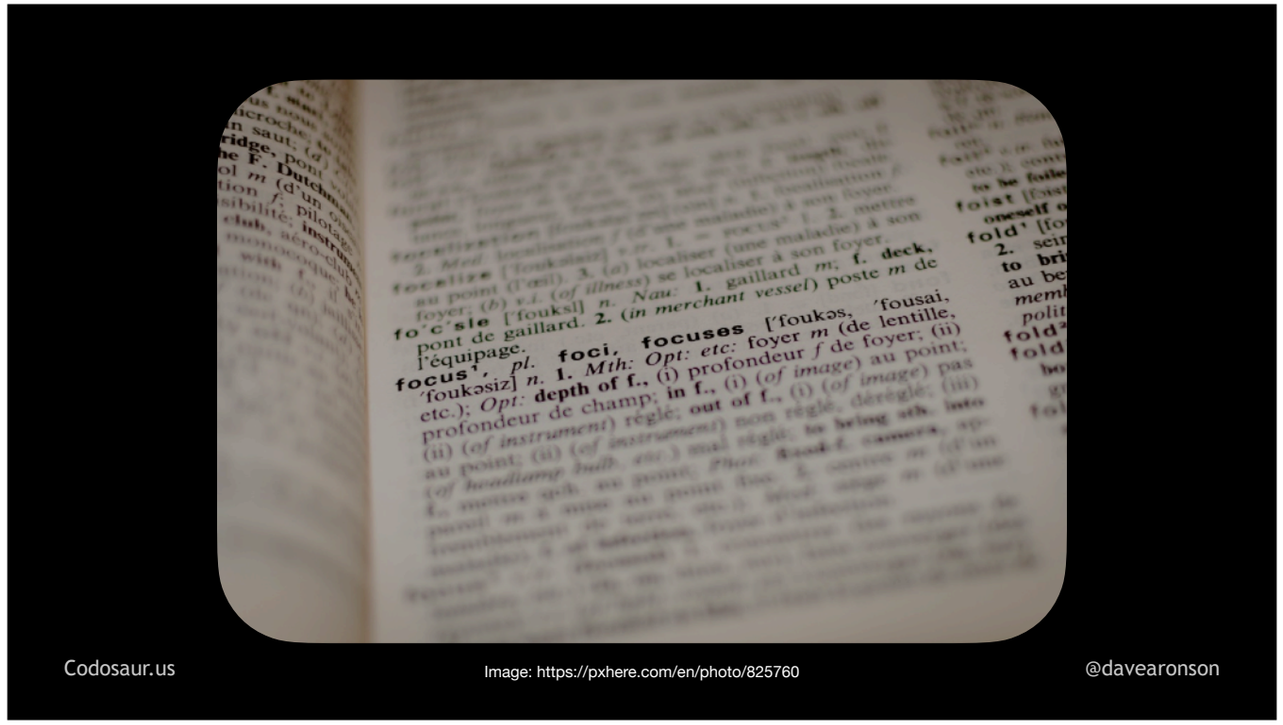
. . . *assumes* that our code is correct, in the sense of passing its tests. Instead, mutation testing checks two *other* qualities, and I think the more *important* one is that our test suite is . . .



. . . *strict*. To check that, a mutation testing tool will try to . . .

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

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

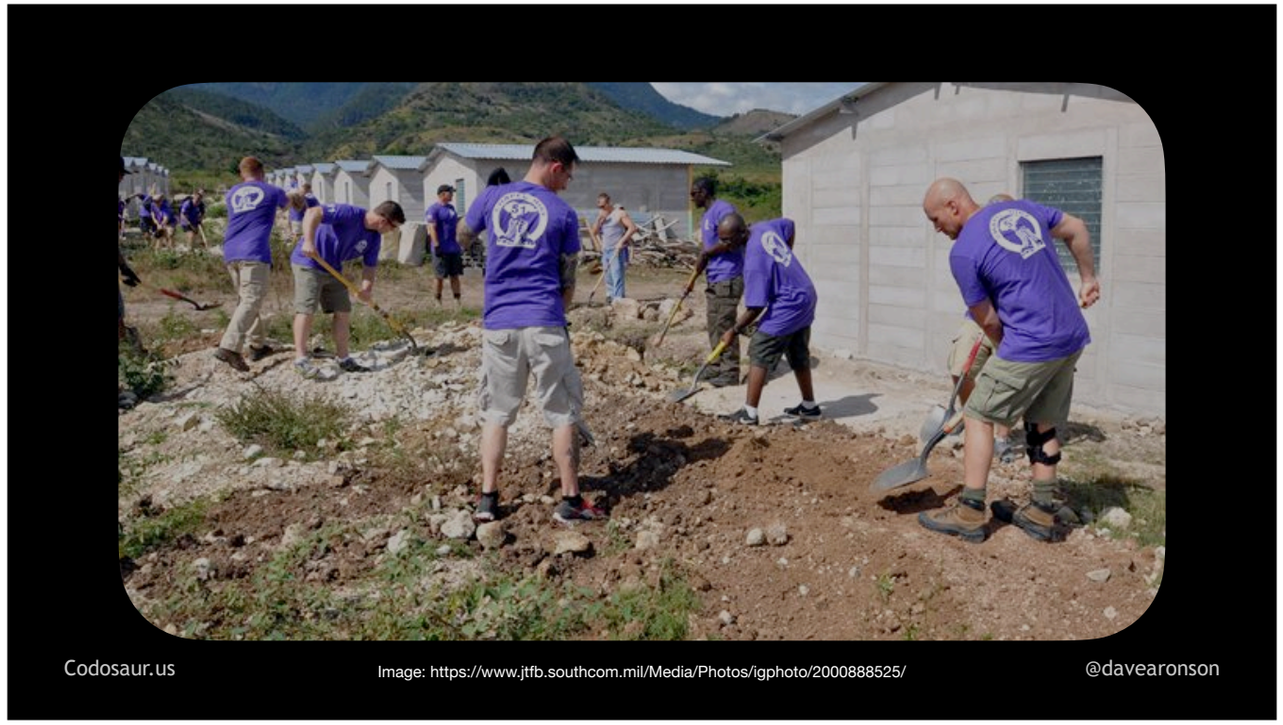


. . . *meaningful*, so that any semantic change to the code will produce a noticeable change in its behavior. Lack of *meaning* comes mainly from code being unreachable, redundant, or lacking without any real effect. When we find such code, the usual fix is just to remove it.

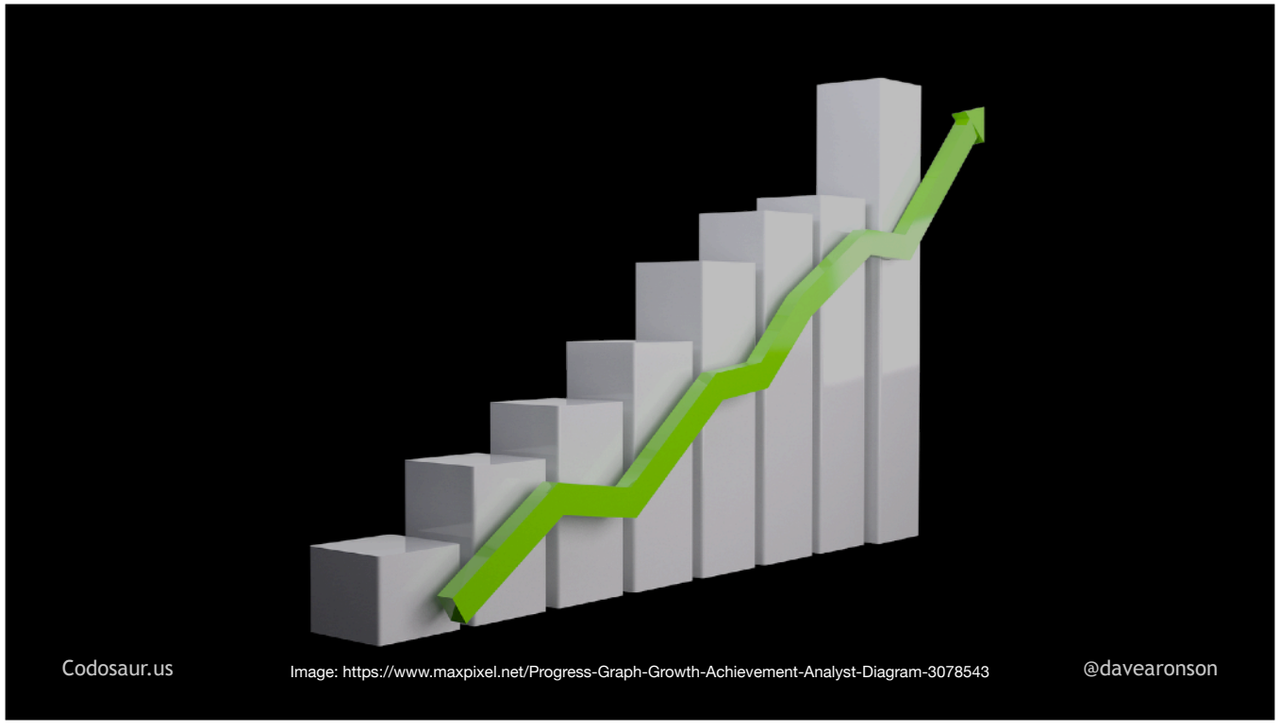
Mutation testing . . .

. . . puts these two together, by checking that any change to the code, that the tool knows how to do, *makes* a noticeable change to its behavior, *and* that at least one test *notices* that change, and fails.

That's the positive side, but there are some drawbacks. The first is that it's rather . . .



. . . hard labor for the CPU, so it’s usually ra-ther slllooooow. We certainly won’t want to mutation-test our entire codebase every time we save a file! Maybe over a lunch break for a smallish system, or a weekend for a large one. Fortunately, most tools let us just check specific functions, classes, files, and so on. Plus, many have an . . .

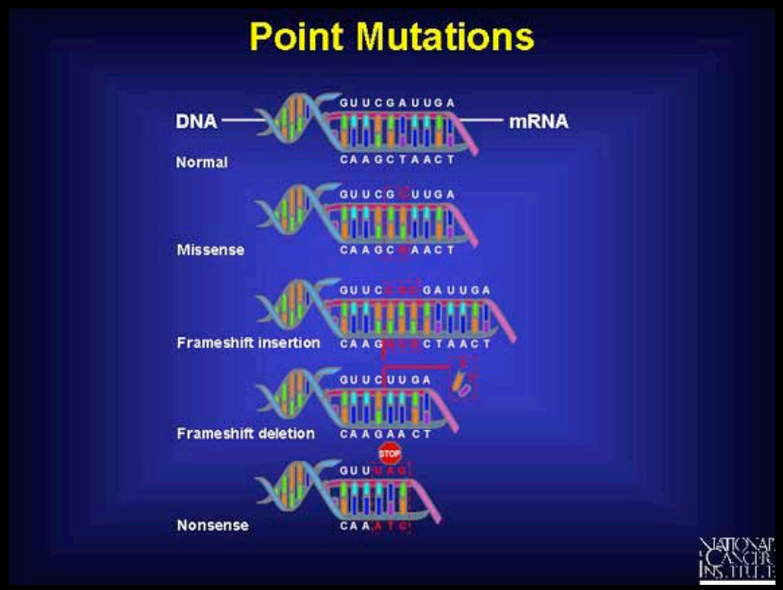


. . . incremental mode, so that we can test only the changes since the last mutation test, or the last git commit, or the main branch, or some such milestone. With such filtering, we can test just some relevant subset of the code, over a *much* shorter break.

Another drawback is that it's often . . .

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

Now that we've seen some of the pros and cons, what does mutation testing *do?* It . . .

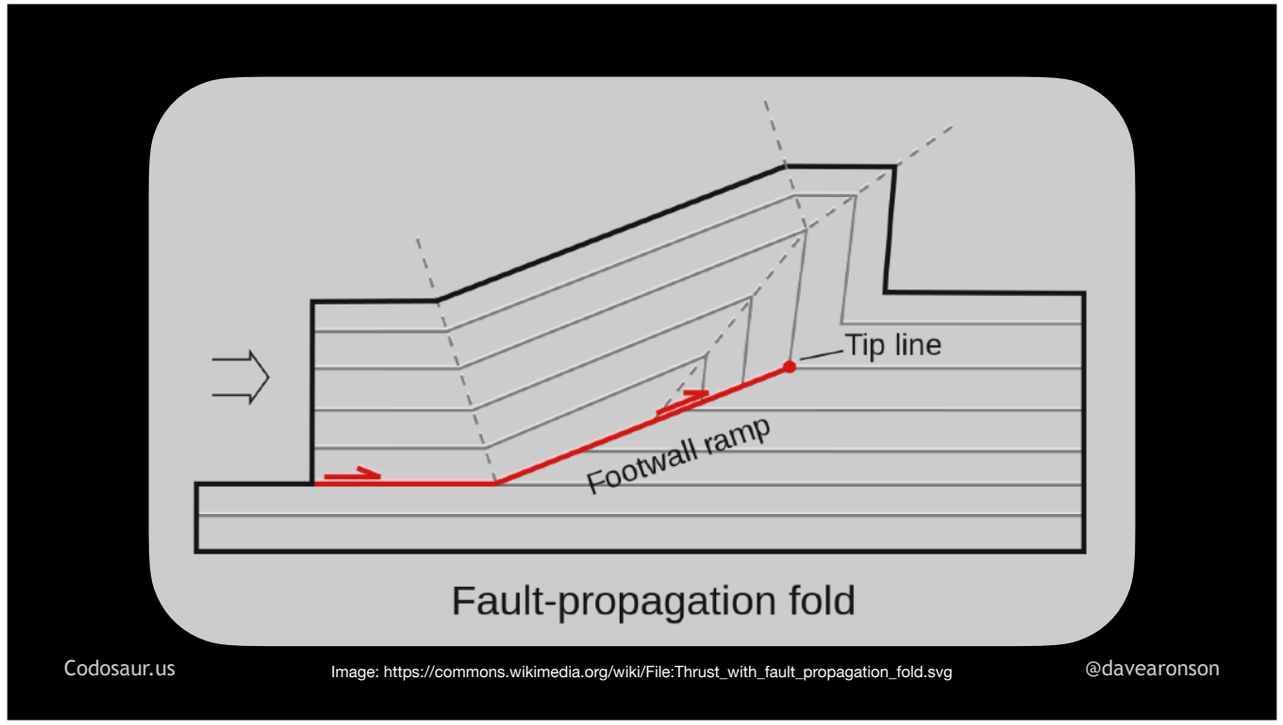


Codosaur.us

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

@davearonson

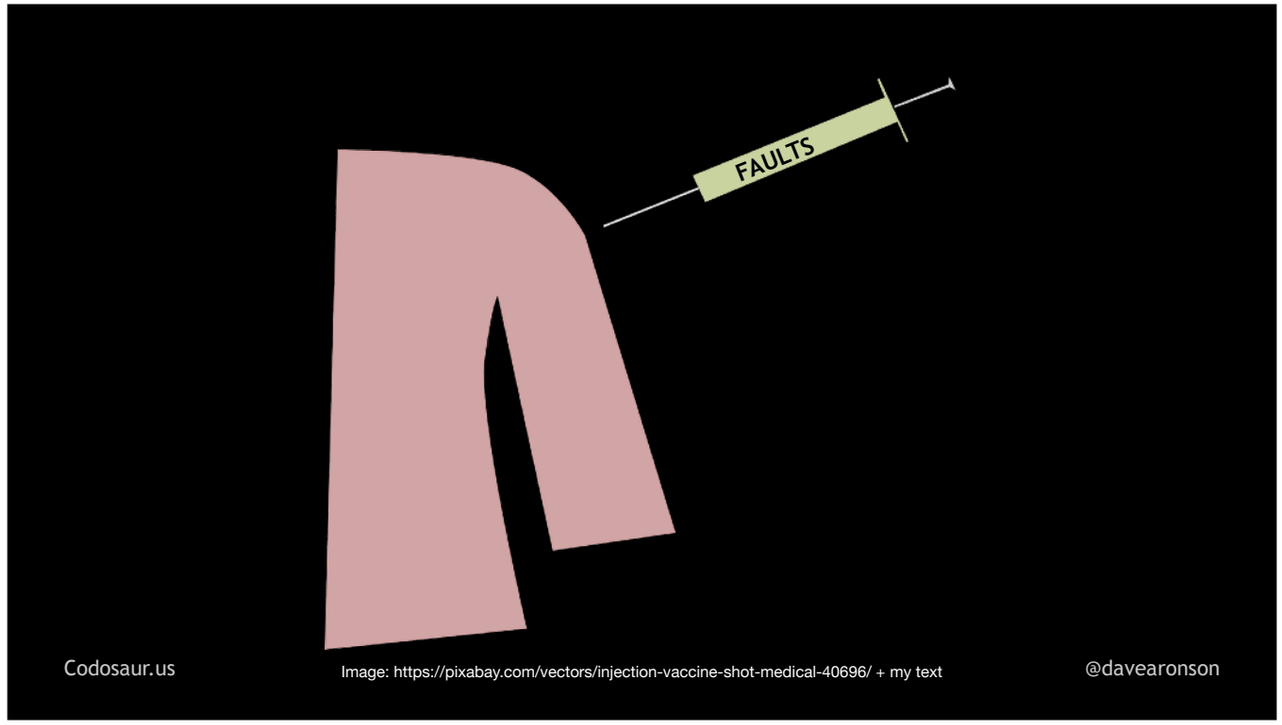
. . . *mutates* copies of our code, hence the name, in order to create test failures, also known as . . .



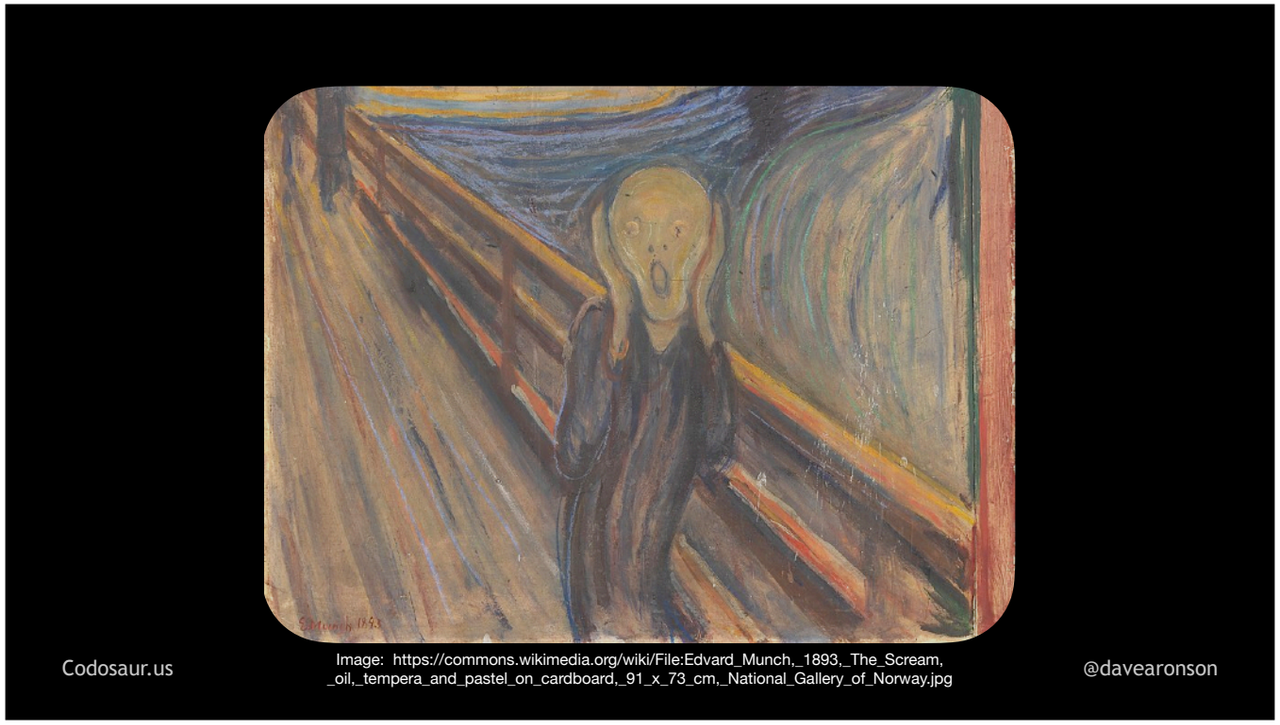
. . . faults. So, mutation testing can be categorized as a *“fault-based”* testing technique, which means that it’s related to something you might already know:

. . . Chaos Monkey, from Netflix. But the way mutation testing does it, is sort of . . .

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

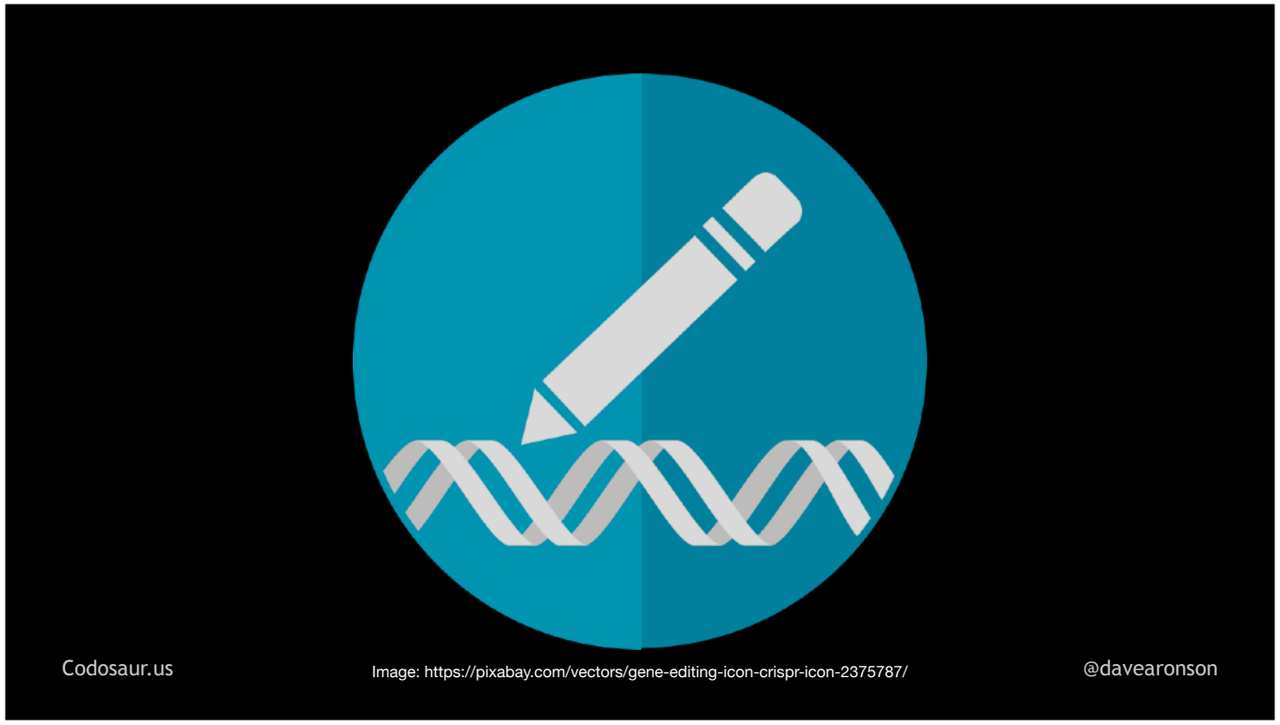


. . . injecting faults into Netflix's production network. (QUICK-CUT TO NEXT SLIDE!)



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

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



. . . copies of our code, not our actual network, and it does this in our . . .

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

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

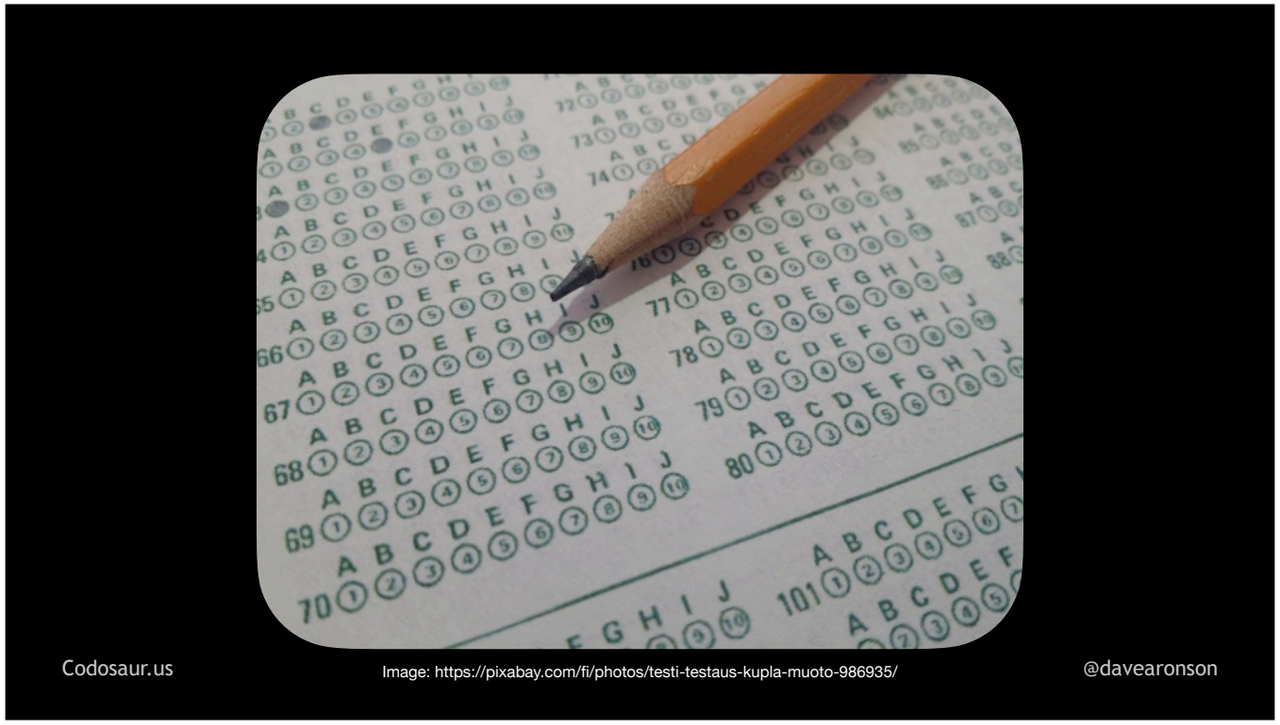
Codosaur.us @davearonson

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

. . . there *is* a problem! Remember, every change should make *at least* one test *fail*.

So how does mutation testing *work?* First, the tool . . .

. . . breaks our code apart into pieces to test, usually our functions. Then, for each one, it tries to find . . .



. . . the *tests* that cover that function, and . . .

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

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

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

Codosaur.us

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

@davearonson

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

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

Codosaur.us

@davearonson

This chart represents the progress of our tool. Most don't give us all this data, let alone so neatly organized, but it's a useful *conceptual* model. For each . . .

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

Codosaur.us

@davearonson

. . . mutant, derived from . . .

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

Codosaur.us

@davearonson

. . . the same function, the tool runs the function's . . .

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

Codosaur.us

@davearonson

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

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

Codosaur.us

@davearonson

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

(PAUSE) If any test . . .

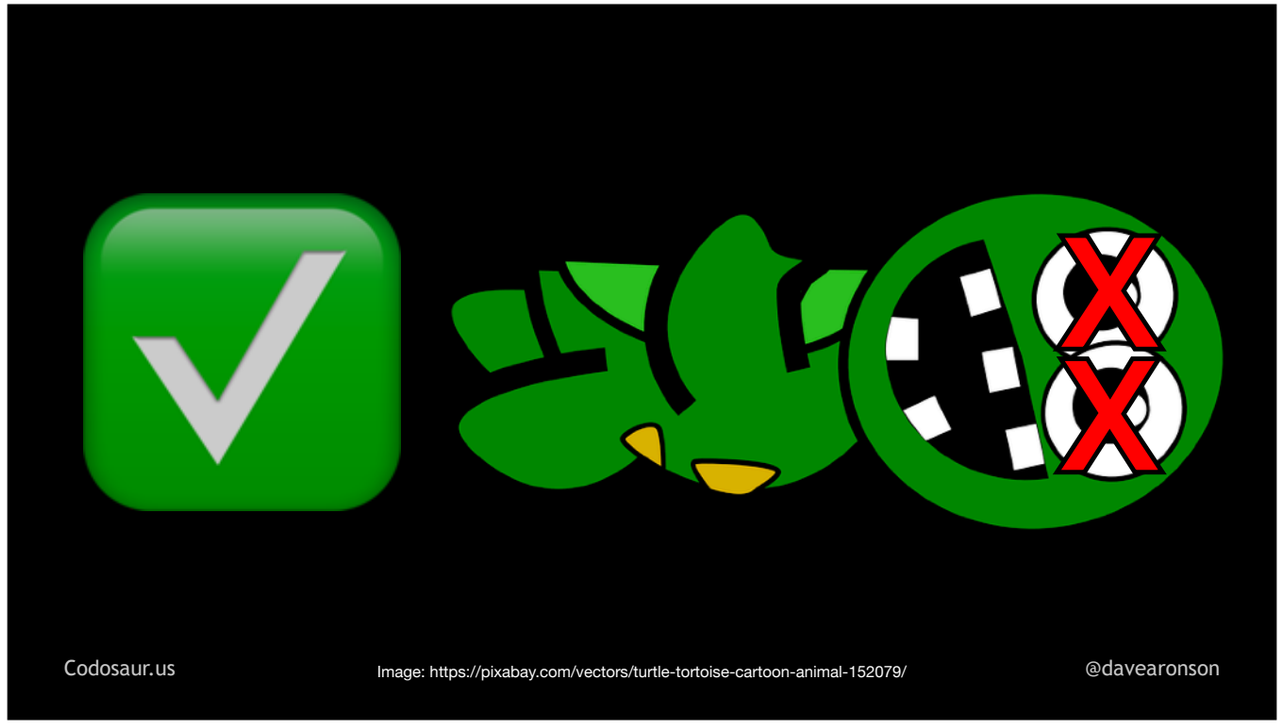
Mutating function `whatever`, at `something.py:42`											
Test # / Mutant #	1	2	3	4	5	6	7	8	9	10	Result
1	✔	✔	✔	✔	✘						In Progress
2											To Do
3											To Do
4											To Do
5											To Do

Codosaur.us

@davearonson

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

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



. . . *good* thing. It means that the change that the tool made, to *create* this mutant, *made a difference* in the function's behavior, *and* that at least one test would *notice* that difference, and fail. Then, the tool will . . .

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

Codosaur.us

@davearonson

. . . mark that mutant killed, . . .

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

Codosaur.us

@davearonson

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

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

Codosaur.us

@davearonson

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

On the other claw, if a mutant . . .

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

Codosaur.us

@davearonson

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

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

Codosaur.us

@davearonson

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

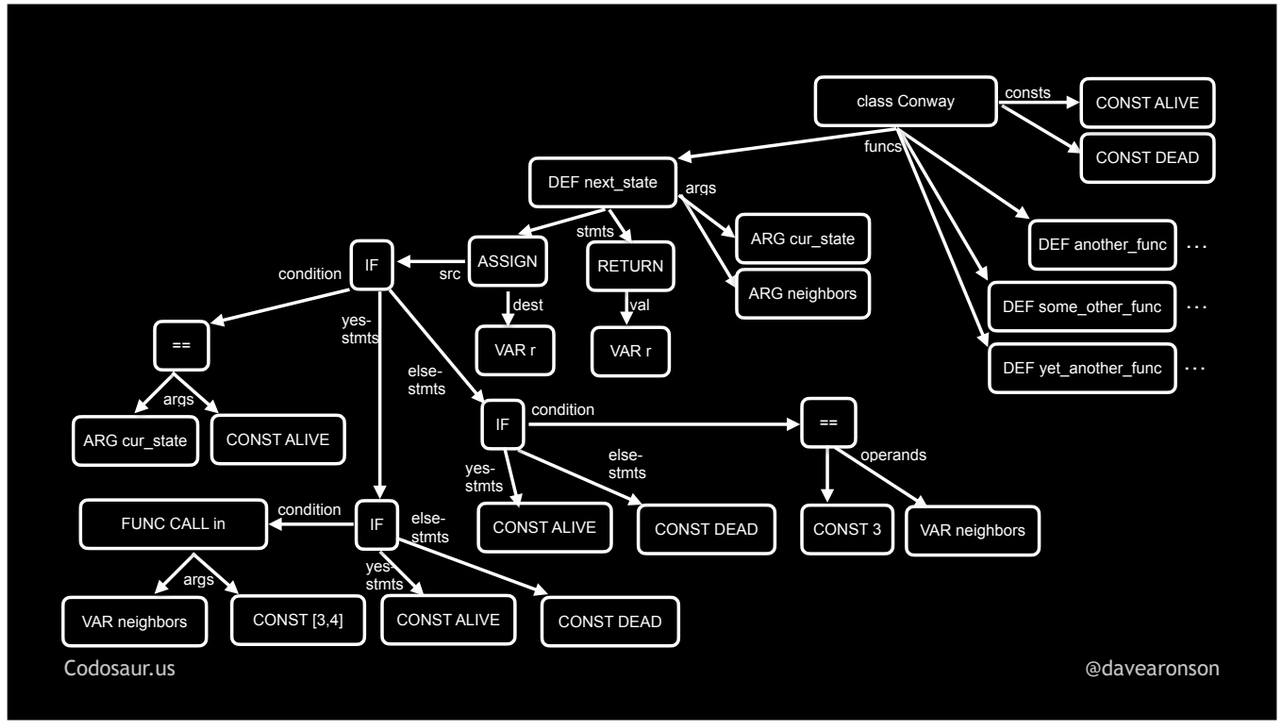
. . . superpower of mimicry, skilled enough to *fool our tests!* This usually means that our code is meaningless, or our tests are lax, or both — and now it’s up to us to figure out how.

Now let's peel back one . . .

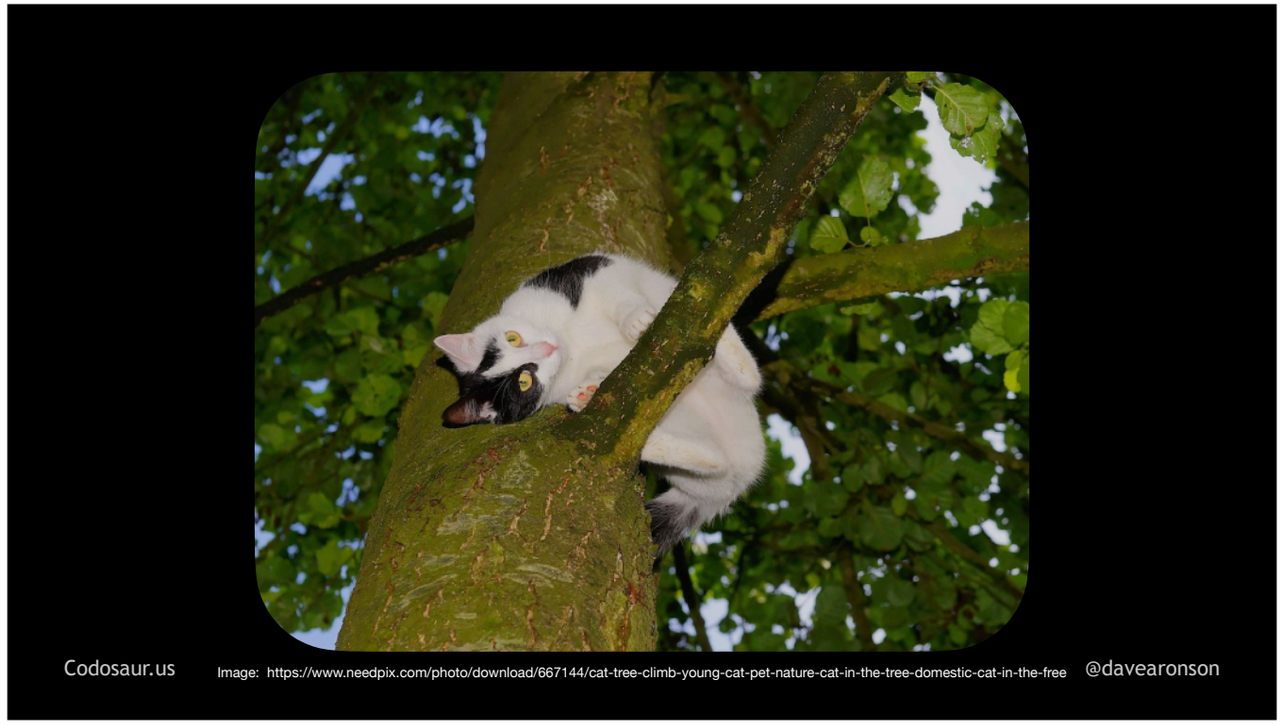
. . . layer of the onion, and look at some *technical details* of how this works. First, our tool parses . . .

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

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

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


. . . our code, usually into . . .



. . . an Abstract Syntax Tree, or AST. Next, the tool . . .



. . . traverses the tree, looking for sub-trees, or branches if you will, that represent each function. For some of the tools, finding the tests requires that we . . .

```
@mumu test-for MyClass.myfunc
def test_myfunc_turns_3_into_6:
  myfunc(3).must_equal 6

@mumu test-for MyClass.myfunc
def test_myfunc_turns_4_into_10:
  myfunc(4).must_equal 10
```


. . . annotate them, or . . .

```
def test_myfunc_turns_3_into_6:
  myfunc(3).must_equal 6


def test_myfunc_turns_4_into_10:
  myfunc(4).must_equal 10
```


. . . follow some naming convention, though the smarter ones can . . .

```
def test_myfunc_turns_3_into_6:
  myfunc(3).must_equal 6


def test_myfunc_turns_4_into_10:
  myfunc(4).must_equal 10
```

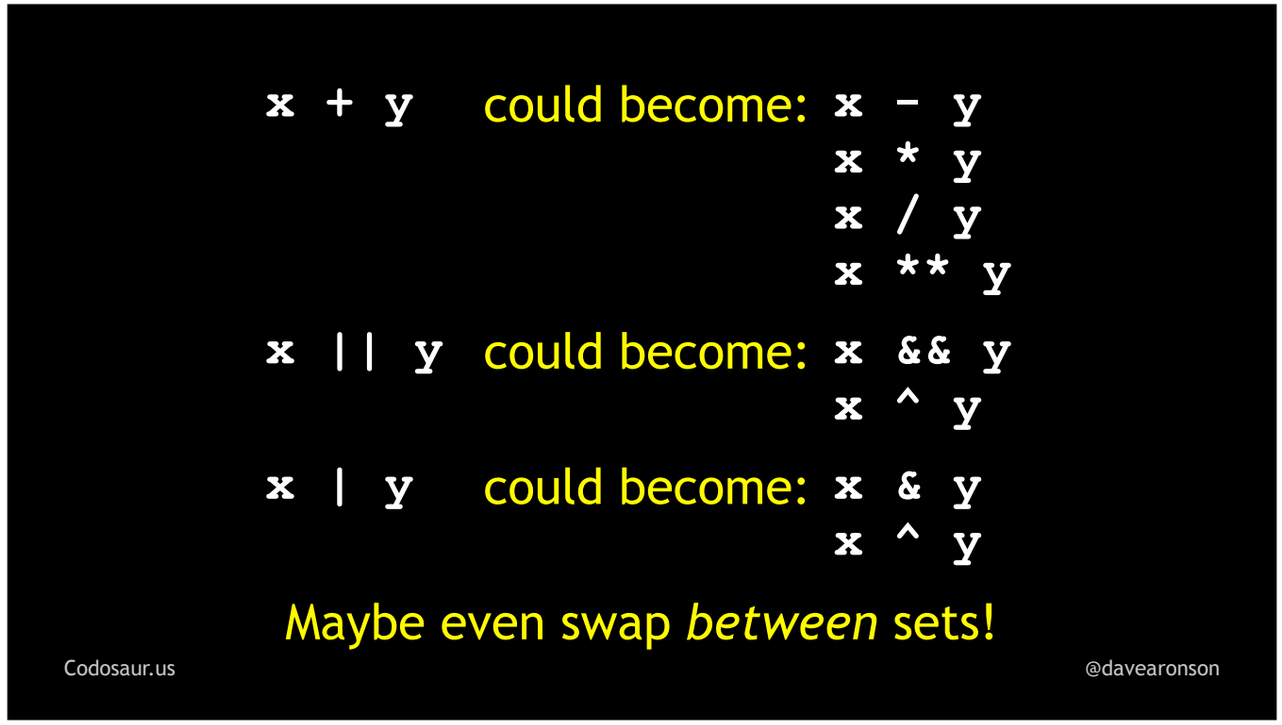
Codosaur.us

@davearonson

. . . look at what tests call what functions from our codebase. Next, to make *mutants* from an AST subtree, it . . .

. . . traverses that subtree, just like it did to the whole thing. But now, instead of looking for even *smaller* subtrees it can *extract*, like twigs or something, it's looking for *nodes* where it can *change* something. Each time it finds one, then for each way it can change that node, it makes one copy of the function's AST subtree, with that one node changed, in that one way, in other words, one mutant with that mutation.

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



It could change an operator from one to another.

`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 oper*ands* matters, it could *swap* them.

```
x < y
```

could become:

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

Codosaur.us

@davearonson

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

`x`

could become:

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

. . . or vice-versa!

It could insert *or remove* a *negation*.

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

could become:

```
foo(z)
```


It can remove an if-condition . . .

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

could become:

```
foo(z)
```

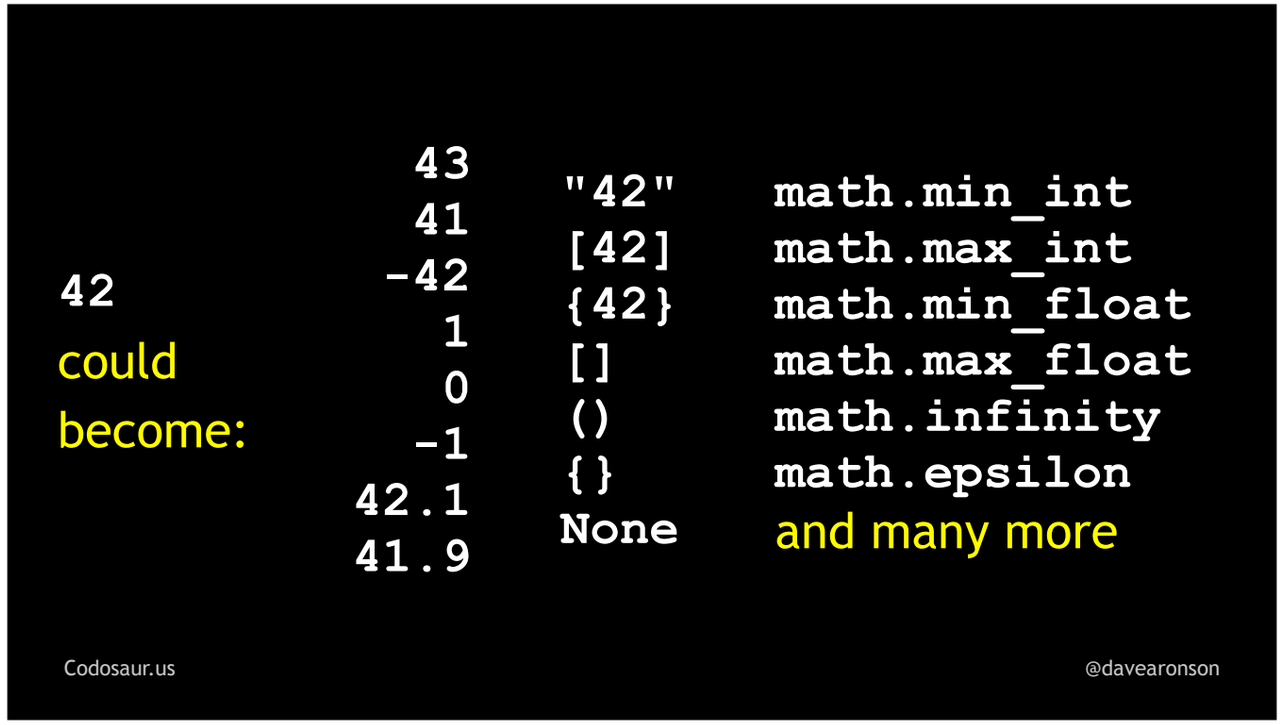

. . . or a looping condition.

```
def f(x, y): # lots of code here
```
could become:
```
def f(x, y): return 0
def f(x, y): return :math.max_int
def f(x, y): return "a string"
def f(x, y): return nil
def f(x, y): return x  # or y
def f(x, y): fail("kaboom")
def f(x, y): # nothing
```
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.



It could change a value to another one, even of an incompatible type, like 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* more types of changes, but I trust you get the idea!

Now let’s *finally* walk through some *examples!* We’ll start with an easy one. Suppose we have a function . . .

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

Codosaur.us @davearonson

. . . like so. Never mind *why*, it just makes a good simple example.

Think about what a mutant made from this might *return*. Mainly, it could return results such as . . .

```
x + y
x - y
x * y
x / y
y ** x
x
y
0
1
-1
0.1
-0.1
```

```
math.min_int
math.max_int
math.max_float
math.min_float
math.infinity
math.epsilon
raise(DeliberateError)
"some random string"
[]
()
{}
None
```

and many more

Codosaur.us

@davearonson

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

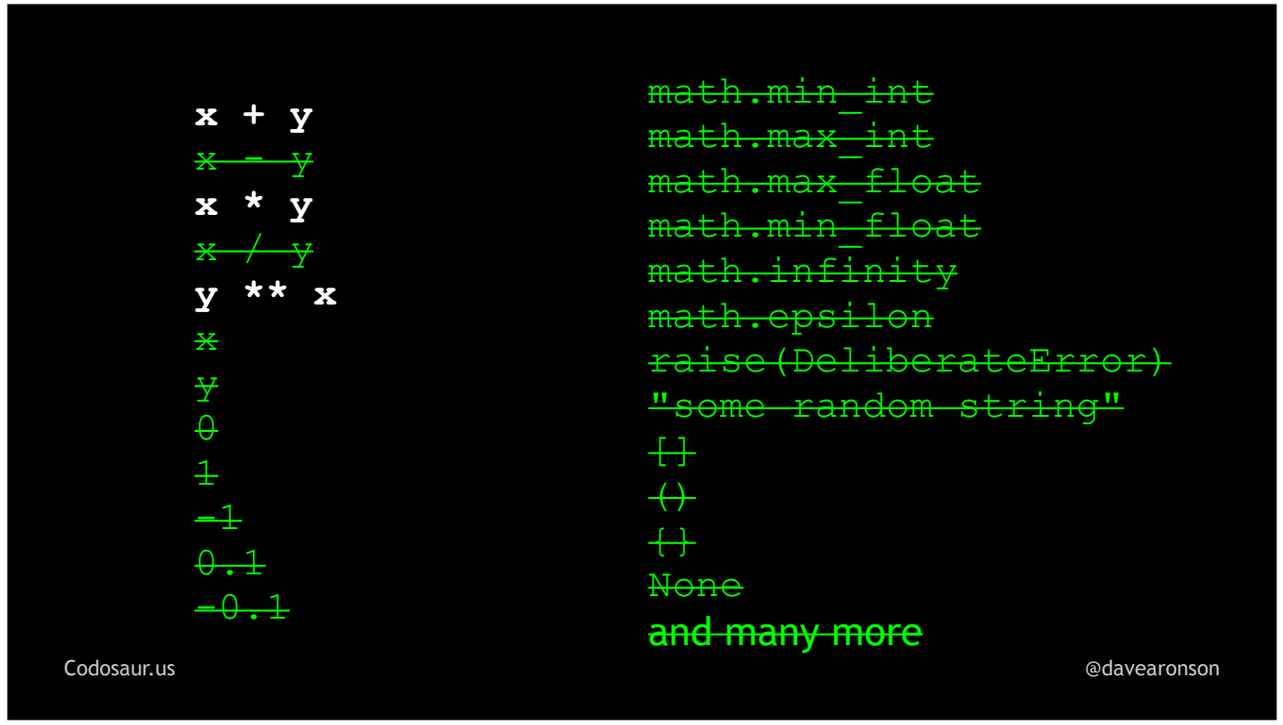
Now suppose we had only one test . . .

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

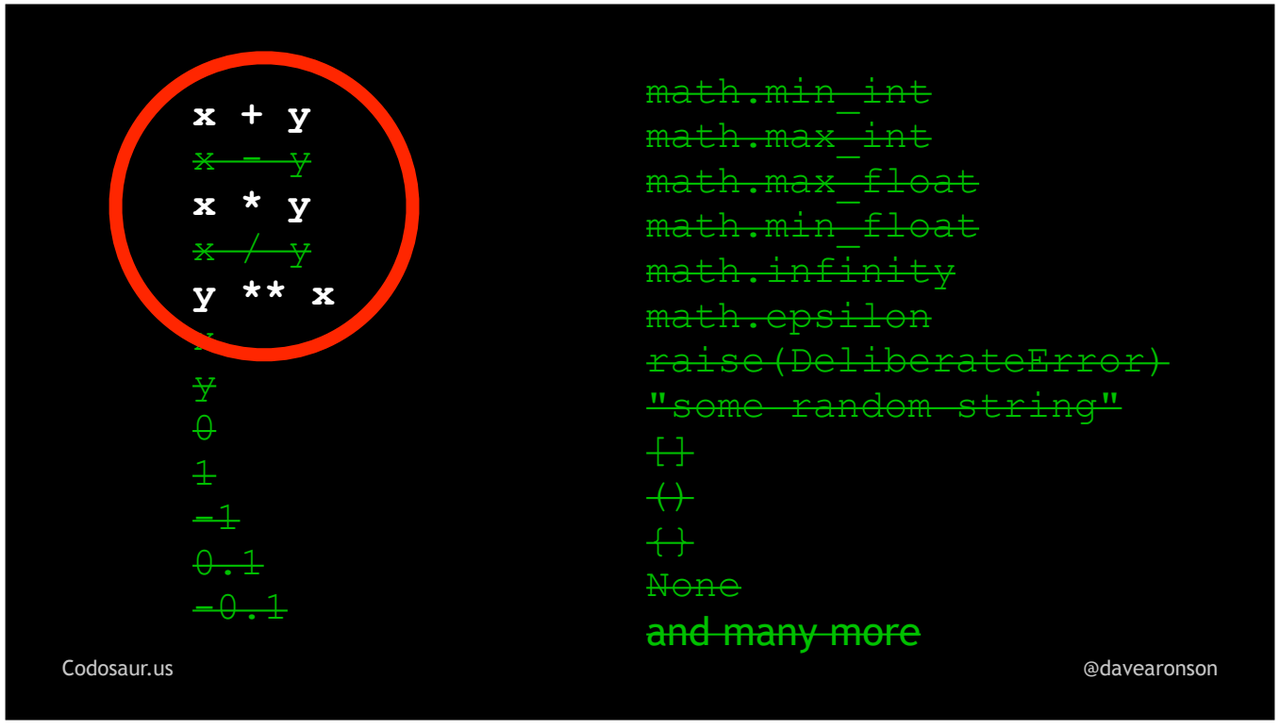
Codosaur.us

@davearonson

. . . like so. This is a rather poor test, and I think at least one reason why is clear to most of us, but even so, it would kill most of those mutants, the ones shown . . .



. . . here in crossed-out green. The ones returning constants, are very unlikely to match. There's no particular reason a tool would put a 4 there, as various significant numbers. Changing the exponentiation into subtraction gets us zero, dividing them gets us one, returning either one alone gets us two, and the mismatched types and deliberate errors will at *least* make the test not pass. But . . .



. . . addition, multiplication, and exponentiation in the reverse order, all get us the correct answer. So, mutants based on *these* mutations will *"survive"* our test. This is the usual way mutants survive, by . . .

```
mutant_power(x, y)

==

original_power(x, y)
```

Codosaur.us

@davearonson

. . . returning the same result as the original function, for the inputs we've used. Or they have the same side effect — whatever our tests are looking at. To determine how that *happens*, for a given mutant, it helps to take a closer look at its mutation, *along with* a test it passes. Let's start with . . .

the change:

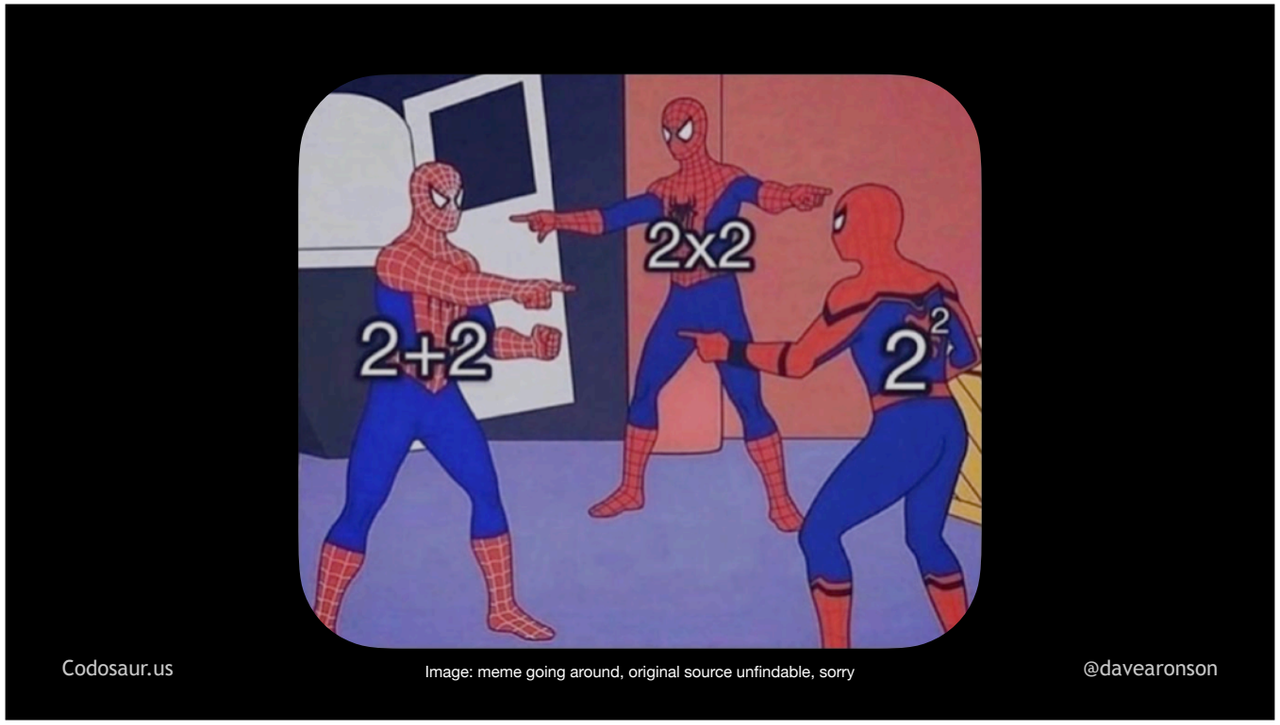
```
43 -    x ** y
43 +    x + y
```

our test:

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

Codosaur.us @davearonson

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



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

So how can we *kill* . . .

the change:

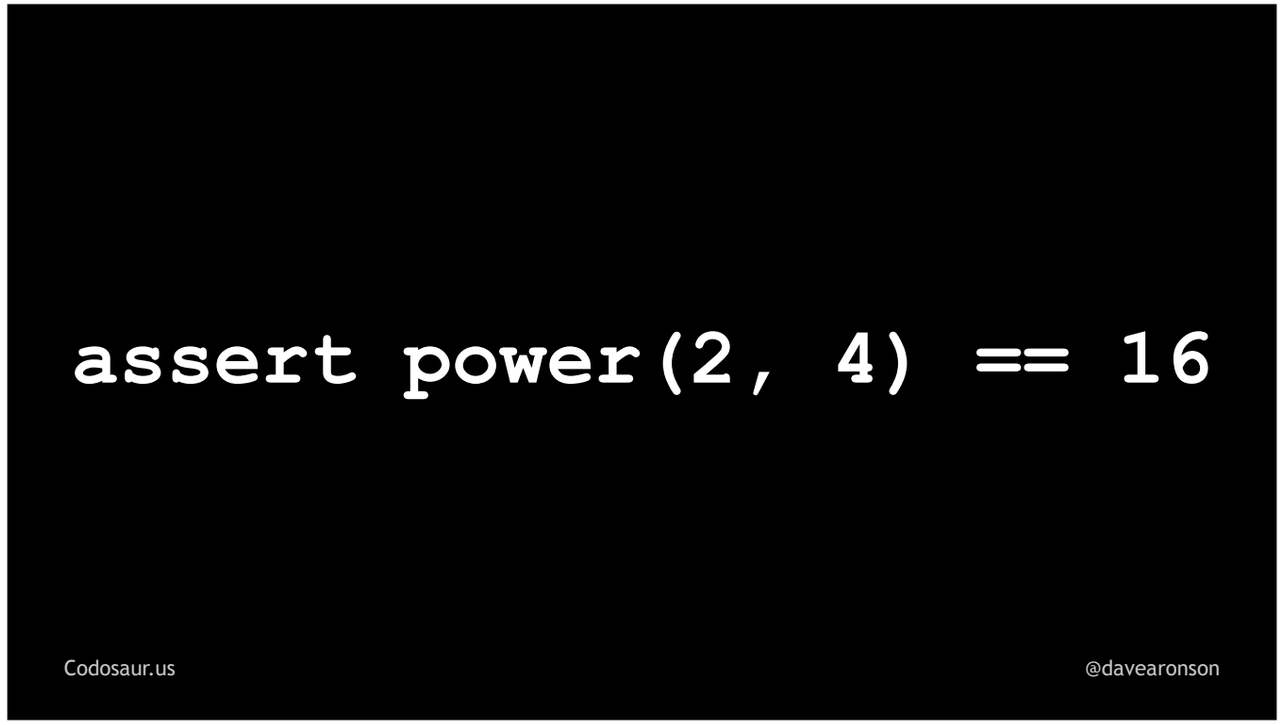
```
43 -    x ** y
43 +    x + y
```

our test:

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

Codosaur.us @davearonson

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



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

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

But . . .

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

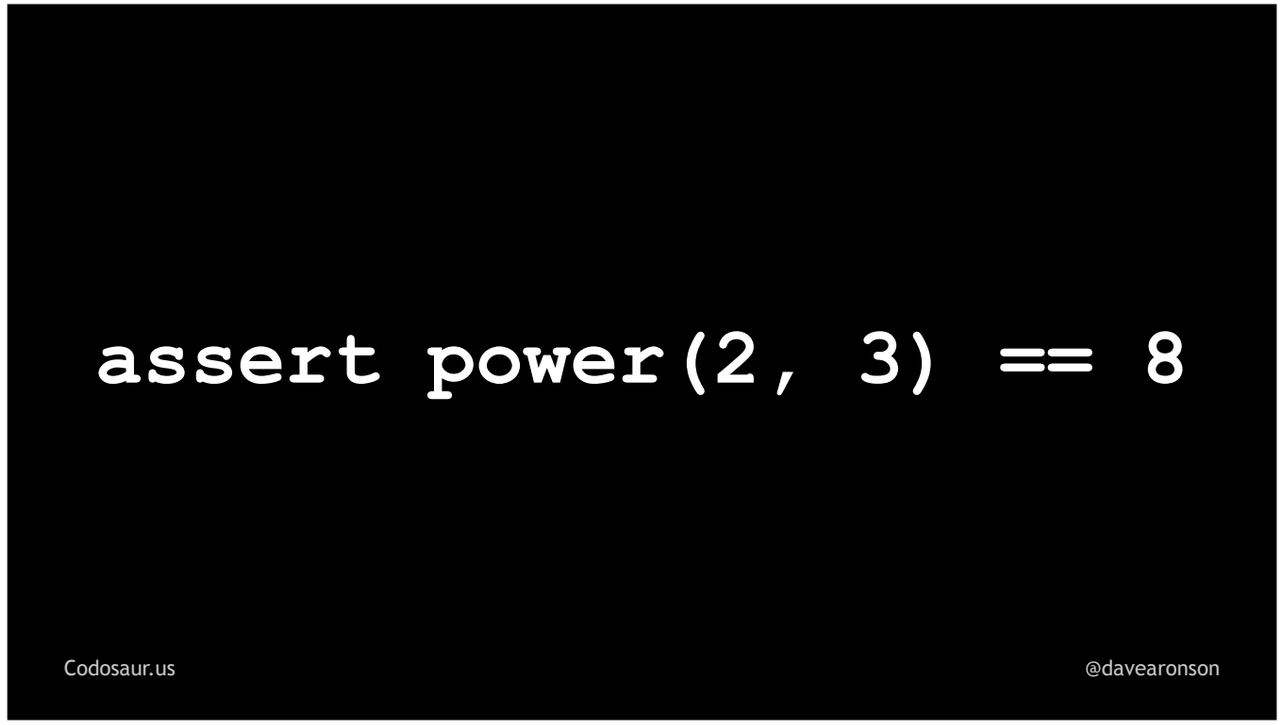
```
2 ** 4 == 16

4 ** 2 == 16
```

Codosaur.us

@davearonson

. . . two to the fourth and four squared, are both sixteen. But we can . . .

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



. . . assert that two to the *third* power is *eight*. Three squared is nine, not eight, so **this kills the argument-swapping mutants**. Better yet, two *plus* three is five, two *times* three is six, and both of those are not eight, so the "plus" and "times" mutants *stay* dead, even if this were still our one and only test. (PAUSE!) With these inputs, the correct operation is the only simple common one that yields the correct answer. This isn't the *only* solution, though; there are *lots* of ways to skin . . .

. . . *that* flerken!

This may make mutation testing sound simple, but this was a downright trivial example. So let's look at a more *complex* one!

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

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

Codosaur.us @davearonson

. . . like so. send_*message* uses send_*bytes* to send as many bytes as send_bytes *could* send, like a woodchuck, looping to pick up where it left off, until the message is all sent.

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

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

Codosaur.us @davearonson

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

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

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

If a mutant that only goes through . . .

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

Codosaur.us @davearonson

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

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

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

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


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

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

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

Codosaur.us @davearonson

. . . we're only testing with a *three* byte message. (PAUSE!)

The obvious fix is to deliberately use a message larger than our maximum chunk size, and at least with the fake messaging we're using here, we can easily construct one . . .

in module Network:

```
max_chunk_size = 10_000
```

in test_send_message:

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

Codosaur.us @davearonson

. . . like so.

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

in module Message:

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

in test_send_message:

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

Codosaur.us @davearonson

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

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

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

Codosaur.us @davearonson

. . . this. Some other stuff is now *clearly* redundant, because we only needed it to support the looping. If we *also* remove that, then it boils down to . . .

```
def send_message(buf, len):
  return send_bytes(buf, 0, len)
```

Codosaur.us @davearonson

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

Now that we've seen examples of finding both bad tests and redundant code, I'll address a couple of . . .

. . . Frequently Asked Questions. First, this all sounds pretty weird, deliberately making tests fail, to prove that the code succeeds! Where did this whole . . .

Codosaur.us

Image: https://commons.wikimedia.org/wiki/File:New_York_Comic_Con_2015_-_Bizarro_%2821931796858%29.jpg

@davearonson

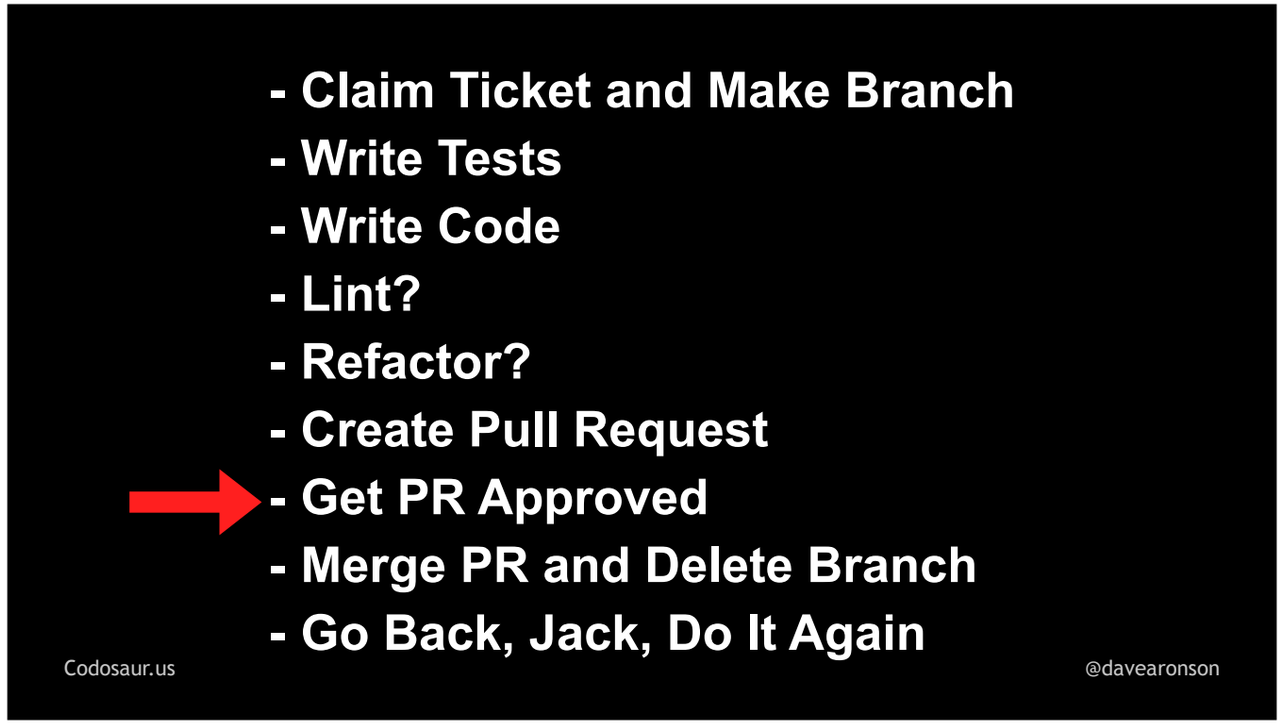
. . . bizarro idea come from? Mutation testing has a surprisingly . . .

. . . long history — at least in the context of computers. It was first proposed in 1971, in Richard Lipton's term paper, “Fault Diagnosis of Computer Programs”, at Carnegie-Mellon University. The first *tool* didn't appear until 1980, in Timothy Budd's PhD work at Yale. But it wasn’t *practical* on typical computers of the day, until the early 2000s, with significant advances in CPU *speed*, multi-*core* CPUs, larger and cheaper memory, and so on. But now, it’s practical even on fairly low-end modern systems, like this 2020 MacBook Air.

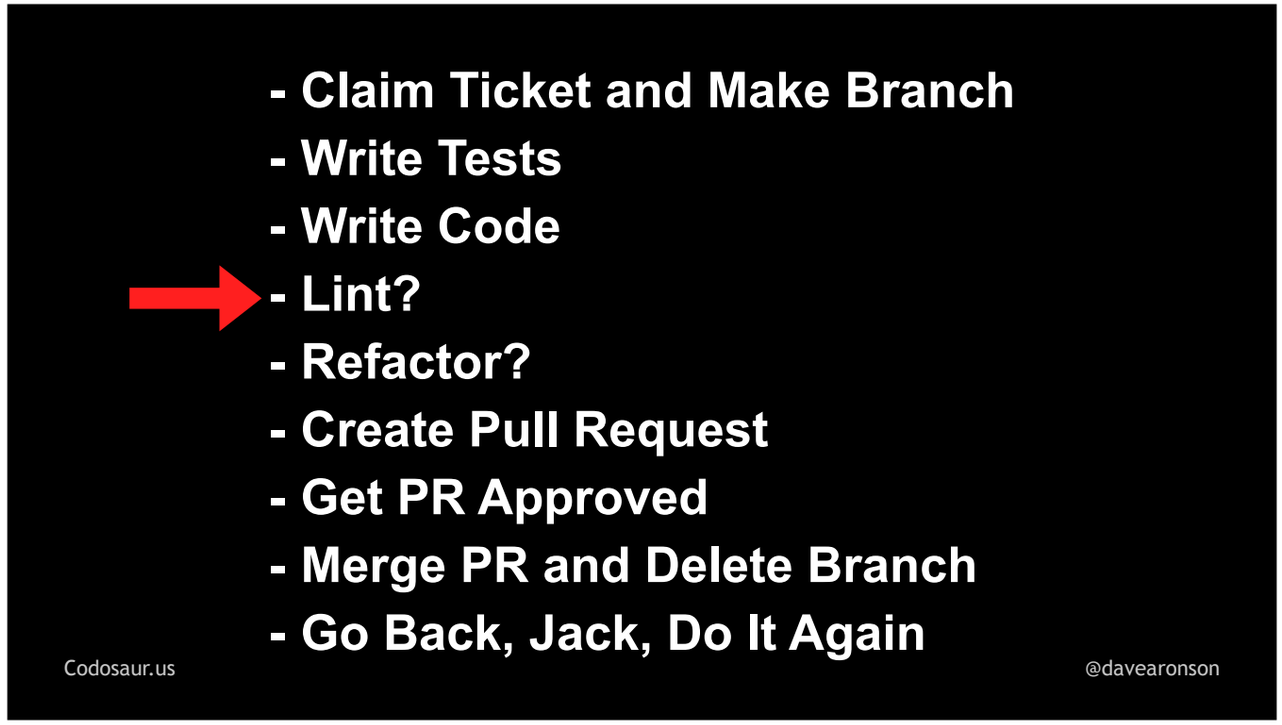
Another common question is: where should we fit this into . . .

- Claim Ticket and Make Branch
- Write Tests
- Write Code
- Lint?
- Refactor?
- Create Pull Request
- Get PR Approved
- Merge PR and Delete Branch
- Go Back, Jack, Do It Again

. . . our development process? Mainly, I think it belongs at *least* . . .



. . . here, as part of the requirements for a Pull Request (or whatever your process uses) to be approved. You can set some standards for what you're willing to tolerate, such as no net *increase* in surviving mutant count. Ideally this would be automated, as part of a CI pipeline, kicked off when the PR is created, and block it if the criteria aren't met. That said, I personally would also do it in my *own* work as part of . . .



. . . the Linting step, where I apply all sorts of other quality checking tools.

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
Dart:	mutation_test
Elixir:	darwin, exavier, exmen, mutation, Muzak [Pro]
Erlang:	mu2
Etherium:	vertigo
FORTRAN-77:	Mothra (written in mid 1980s!)
Go:	go-mutesting, gremlins, ooze
Haskell:	fitspec, muCheck
Java:	jumble, major, metamutator, muJava, pit/pitest, and many more
JavaScript:	stryker, ~~grunt-mutation-testing~~
Pharo:	MUTALK
PHP:	infection, ~~humbug~~, pest
PL/SQL:	MuPLSQL
Python:	cosmic-ray, mutmut, mutpy, pester, xmutant
Ruby:	mutant, ~~mutest~~, ~~heckle~~
Rust:	mutagen
Scala:	scalamu, stryker4s
Smalltalk:	mutalk
Solidity:	RegularMutator
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. Don’t worry about pictures, the last slide has the URL for the whole deck. The tools I know are outdated, are crossed out.

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

😀 Checks that code is meaningful

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

😀 Checks that code is meaningful

😀 Checks that tests are strict

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

😀 Checks that code is meaningful

😀 Checks that tests are strict

😀 Easy to get started with

Codosaur.us

@davearonson

easy to get started with, but it's . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

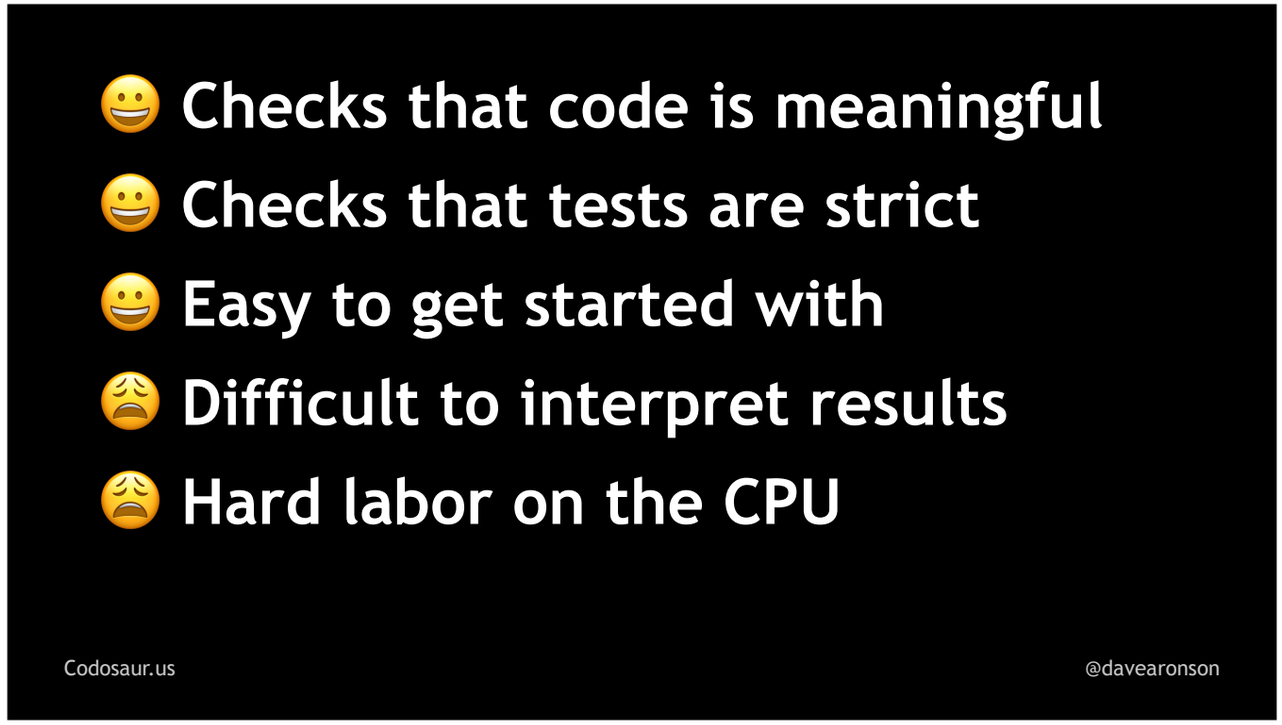
😀 Easy to get started with

😩 Difficult to interpret results

Codosaur.us

@davearonson

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



. . . easy on the CPU.

Even if these drawbacks mean it might not be a good fit for our current projects, I still think it's just . . .

😀 Checks that code is meaningful

😀 Checks that tests are strict

😀 Easy to get started with

😩 Difficult to interpret results

😩 Hard labor on the CPU

😎 Fascinating concept! 🤓

Codosaur.us

@davearonson

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

If you have any questions, . . .

. . . we have about five minutes, and if you think of something later, there's my contact info, plus, as promised, the URL for the slides, complete with a full script, which I've *mostly* stuck to. Any questions?