# KILL ALL MUTANTS!

## (Intro to Mutation Testing)

by Dave Aronson
T.Rex-2025@Codosaur.us




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

CURRENT TIME: ~26:00, want <= 30 including Q&A, so OK

# KILL ALL MUTANTS!

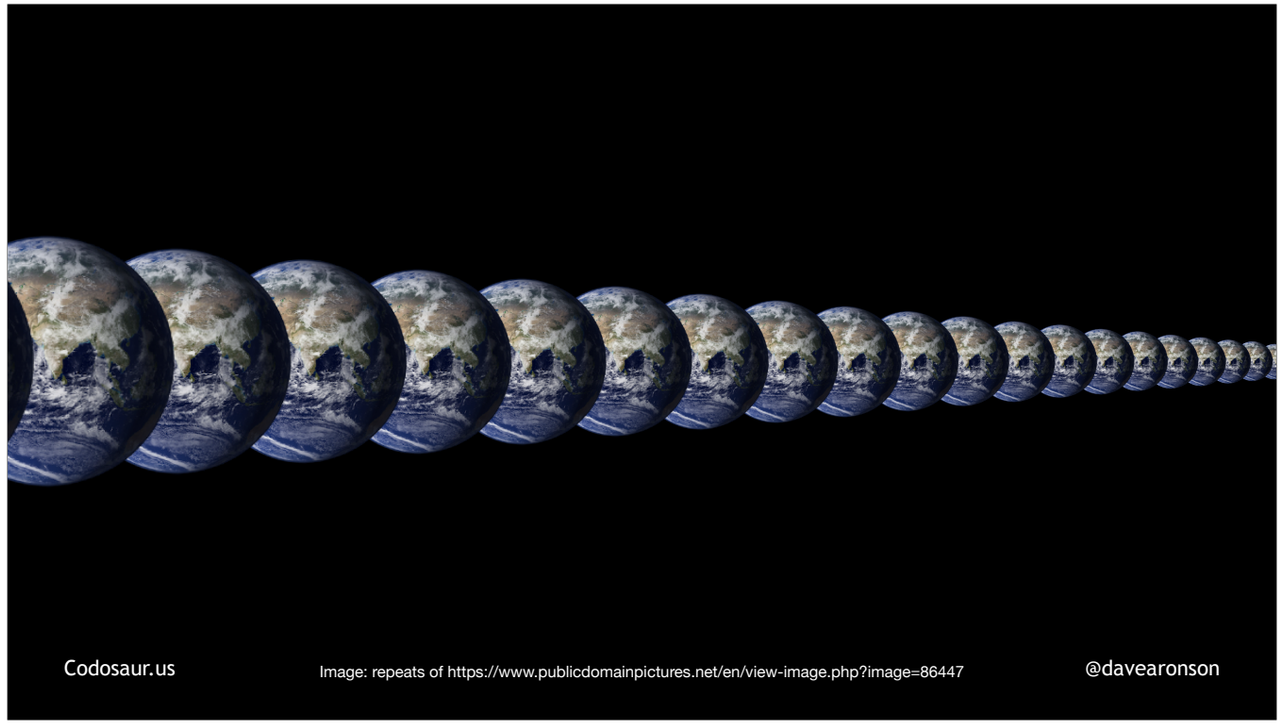
## (Intro to Mutation Testing)

by Dave Aronson
T.Rex-2025@Codosaur.us

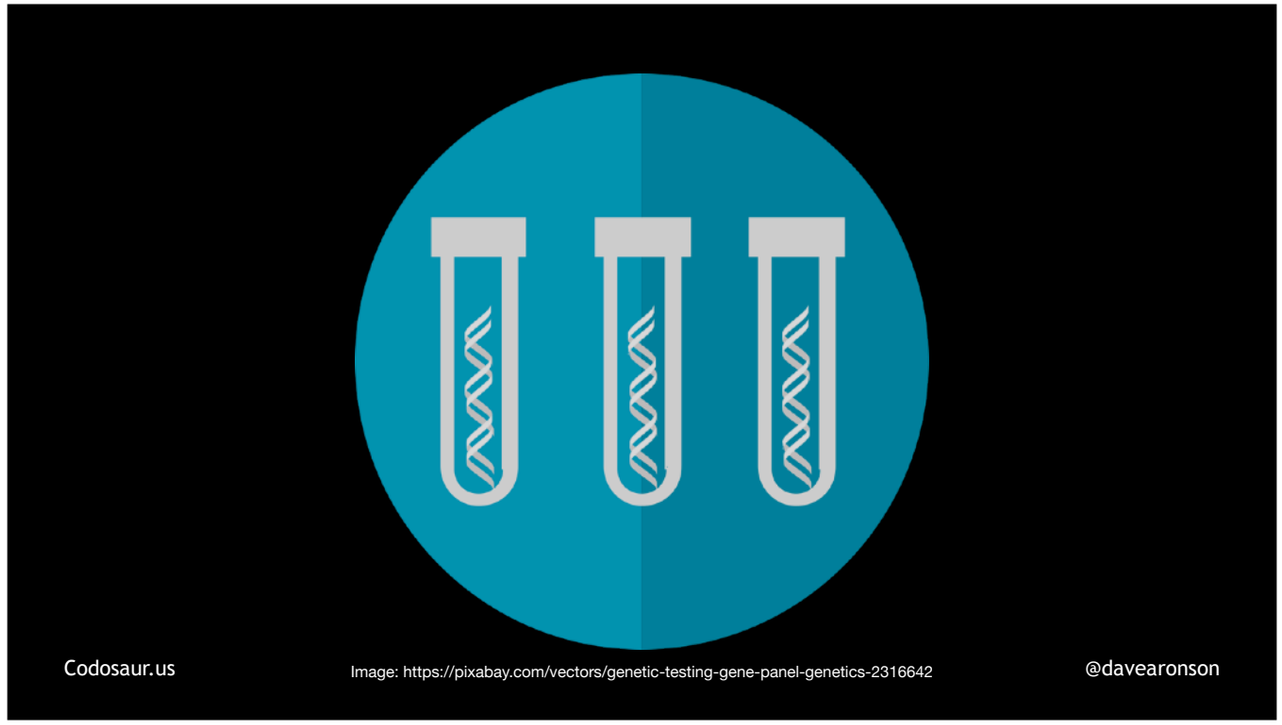



Hi everybody, I’m Dave Aronson, the T. Rex of Codosaurus, LLC.  Some of you may already know me from NoVa-Python, DC Tech, and other local meetups.  I’m here tonight to teach you to KILL MUTANTS!

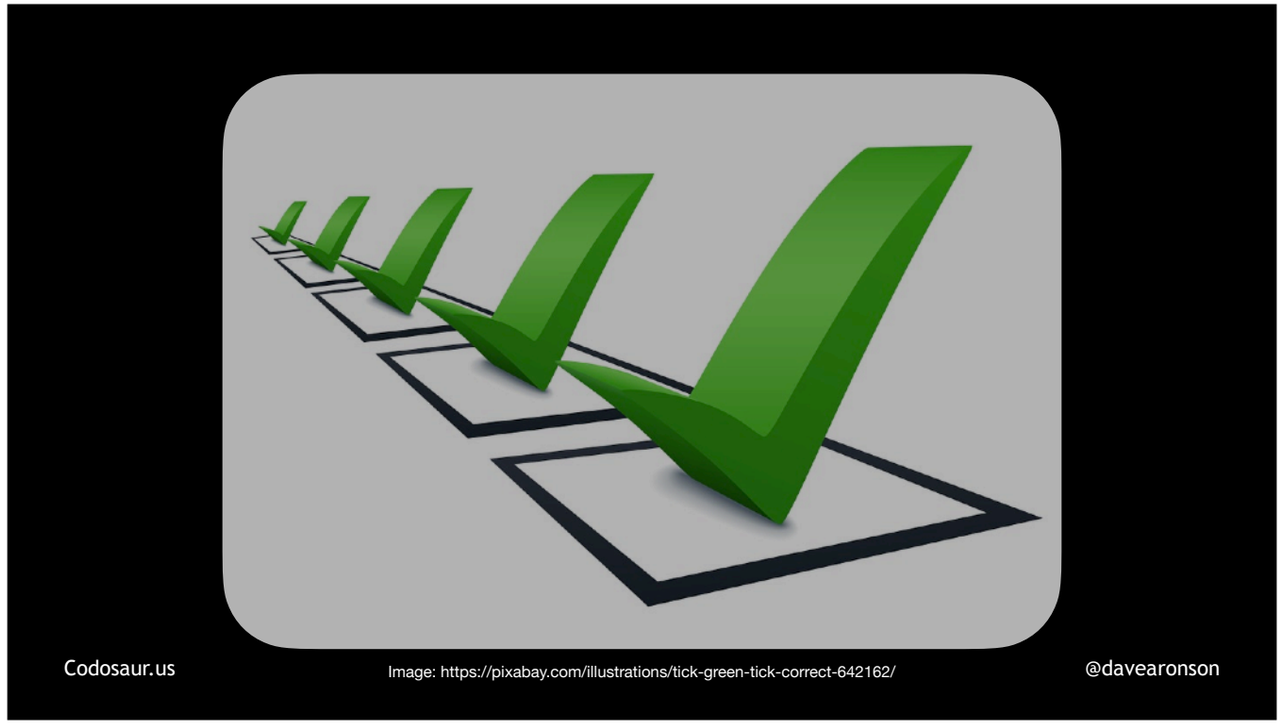
So, what on . . .



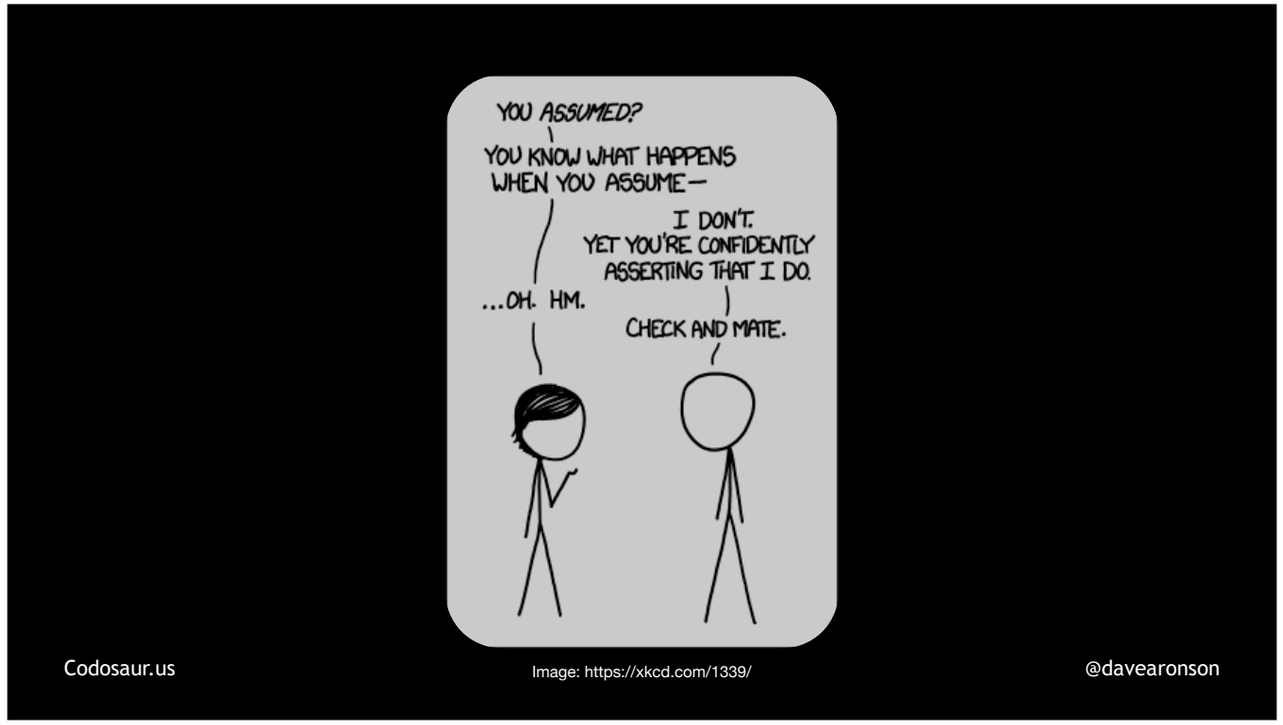

. . . Infinite Earths, makes . . .




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




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

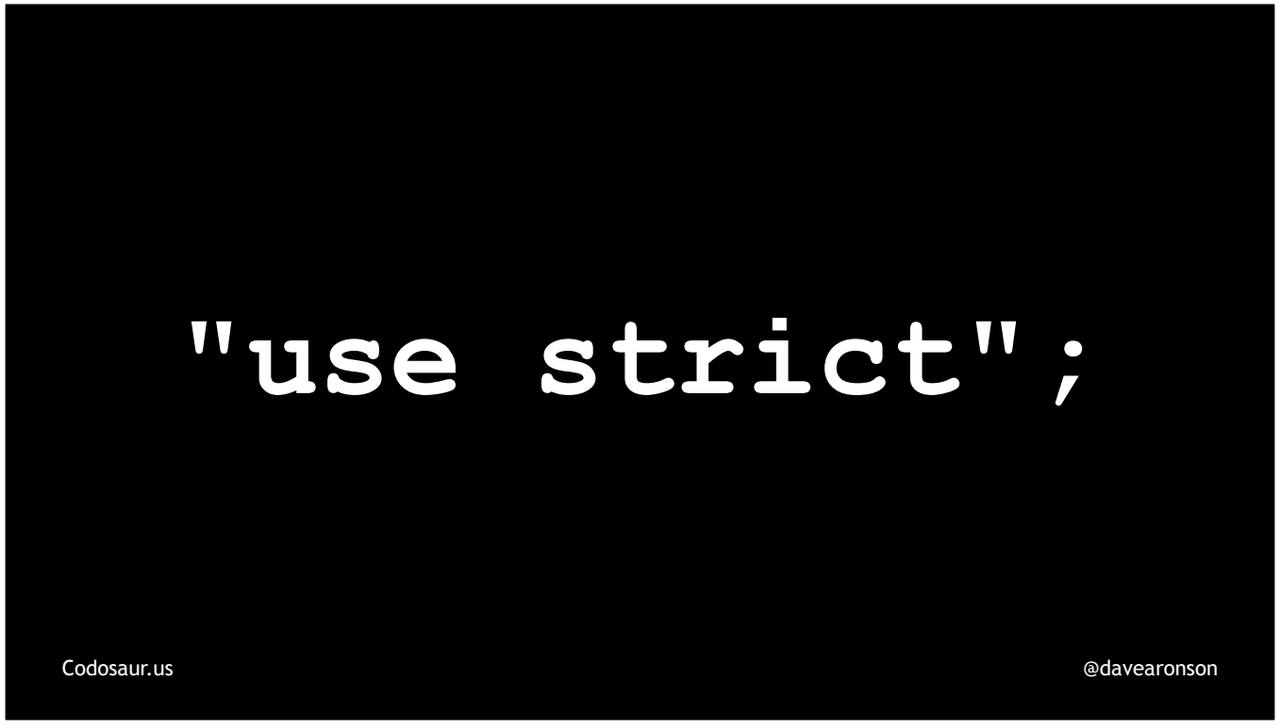



Codosaur.us

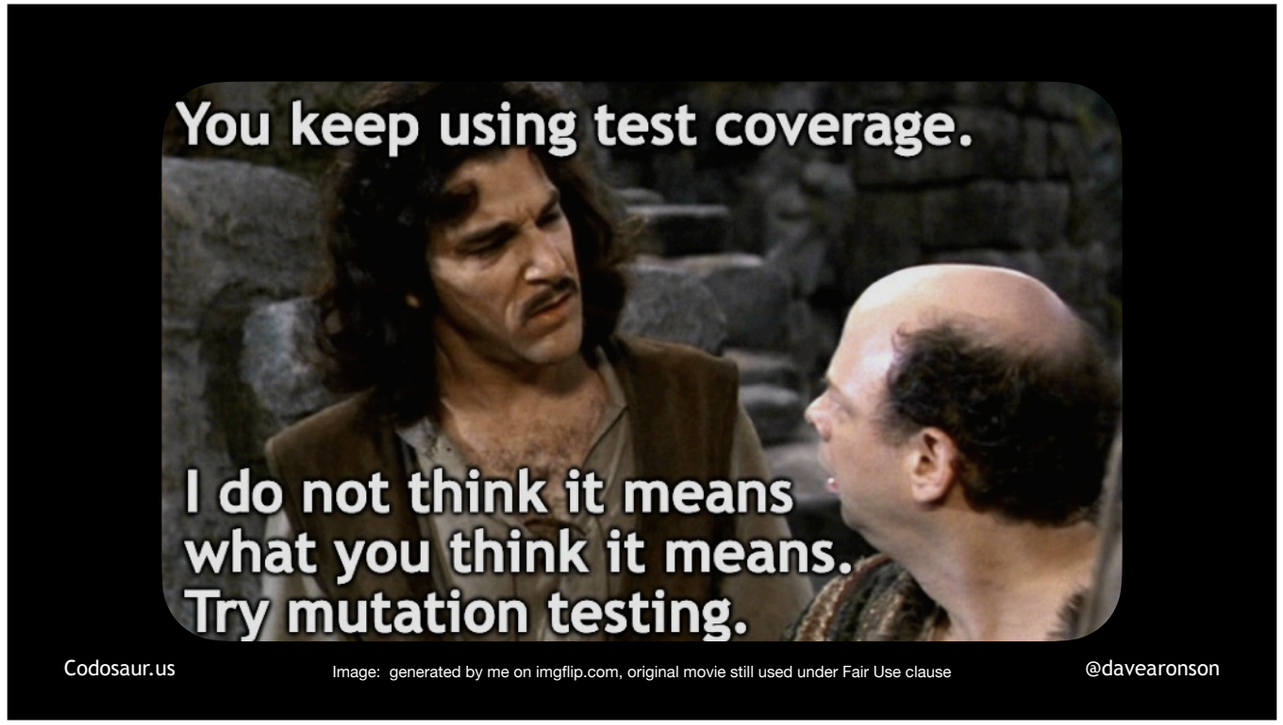
Image: https://xkcd.com/1339/

@davearonson

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




. . . *strict*.  Now you might think, “Isn’t that what test coverage is for?”




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

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


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

  def another_func:
    # whatever
```

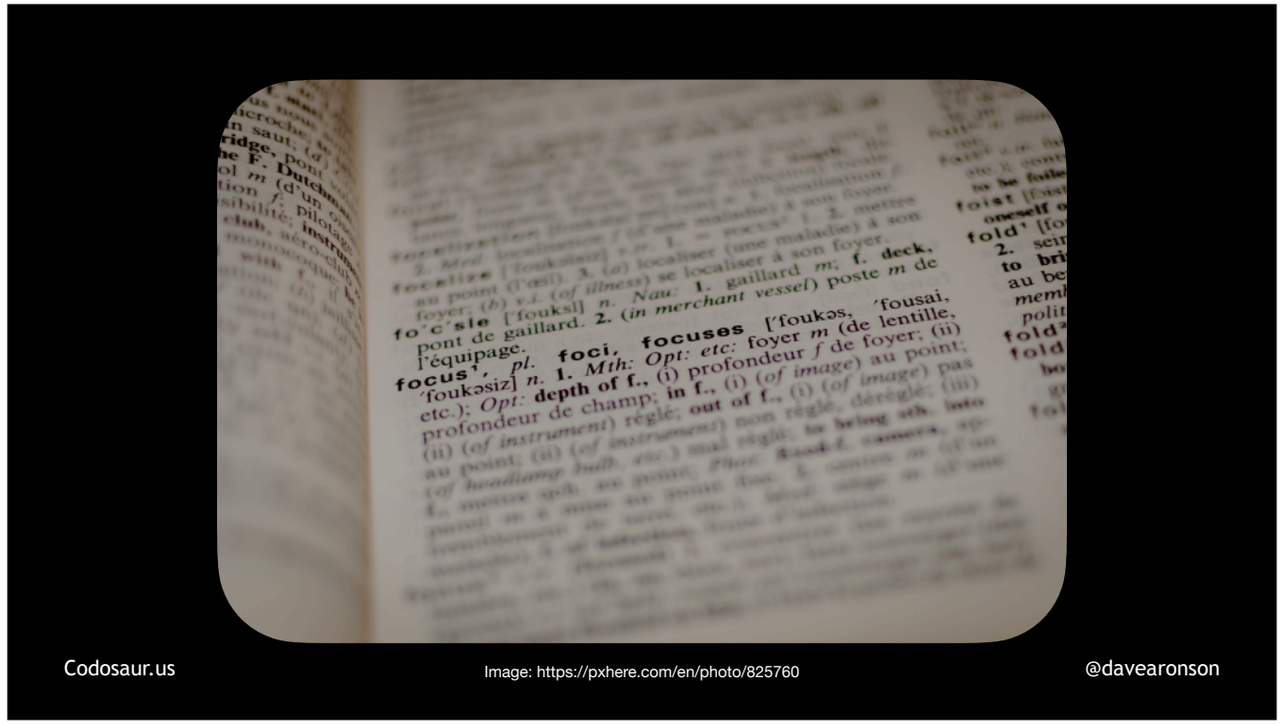
Codosaur.us @davearonson

. . . the code it claims is “covered”.  It tells us NOTHING about whether the *correctness* of the code *made any difference* to whether any test passed.  So how *can* we tell if the code really is *tested*, not just *run?*  That’s where mutation testing comes in.

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

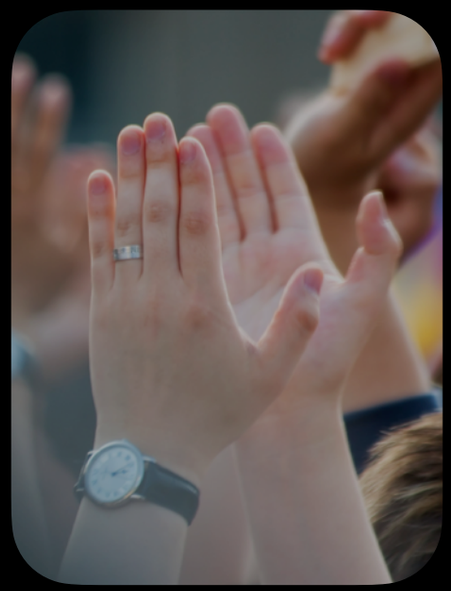

. . . find the gaps in our test suite, that let our code get away with unintended behavior.  Once we find gaps, we can close them by either adding tests or improving existing tests.  Lack of strictness comes mainly from *lack* of tests, or poorly *written* tests.

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




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

Mutation testing . . .



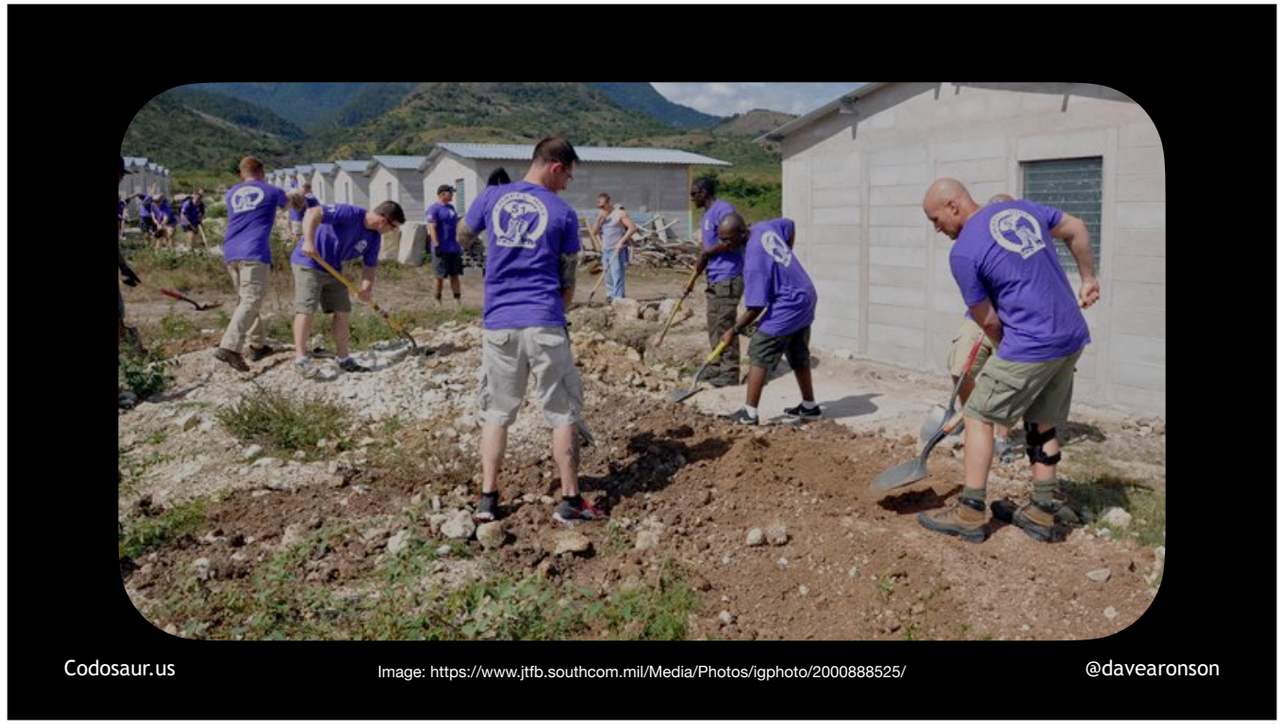
Codosaur.us

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

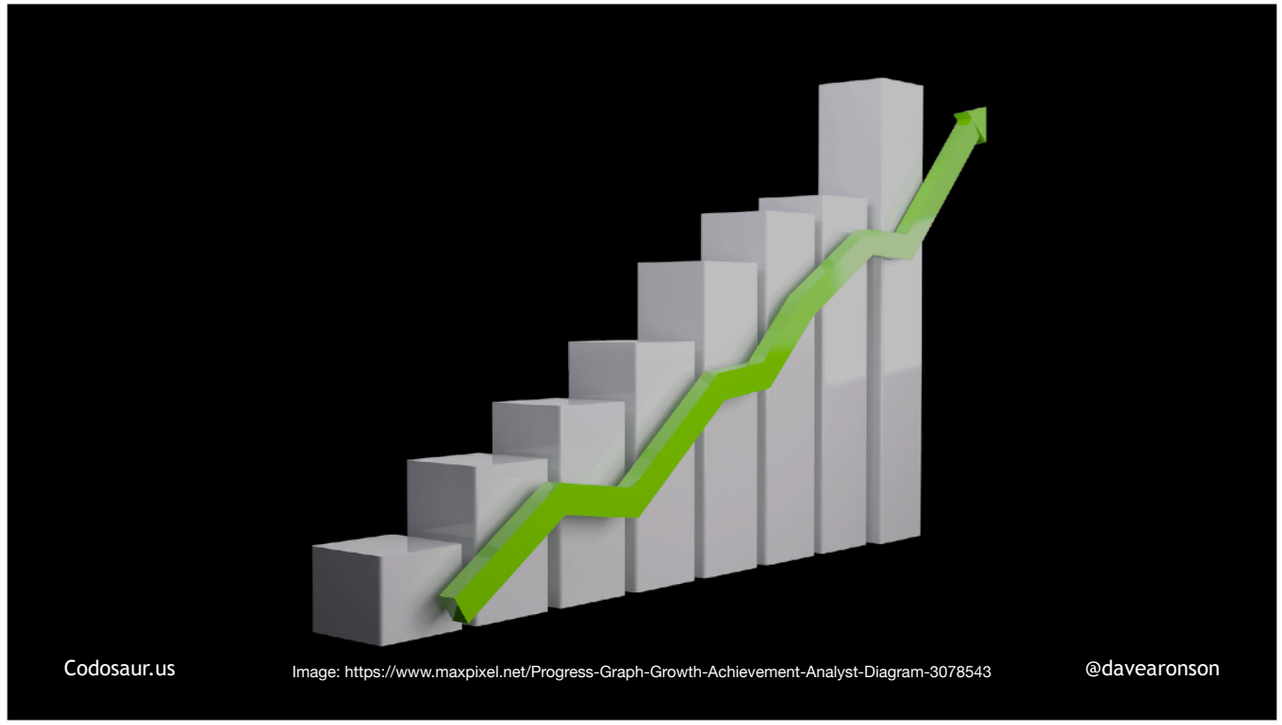
@davearonson

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

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



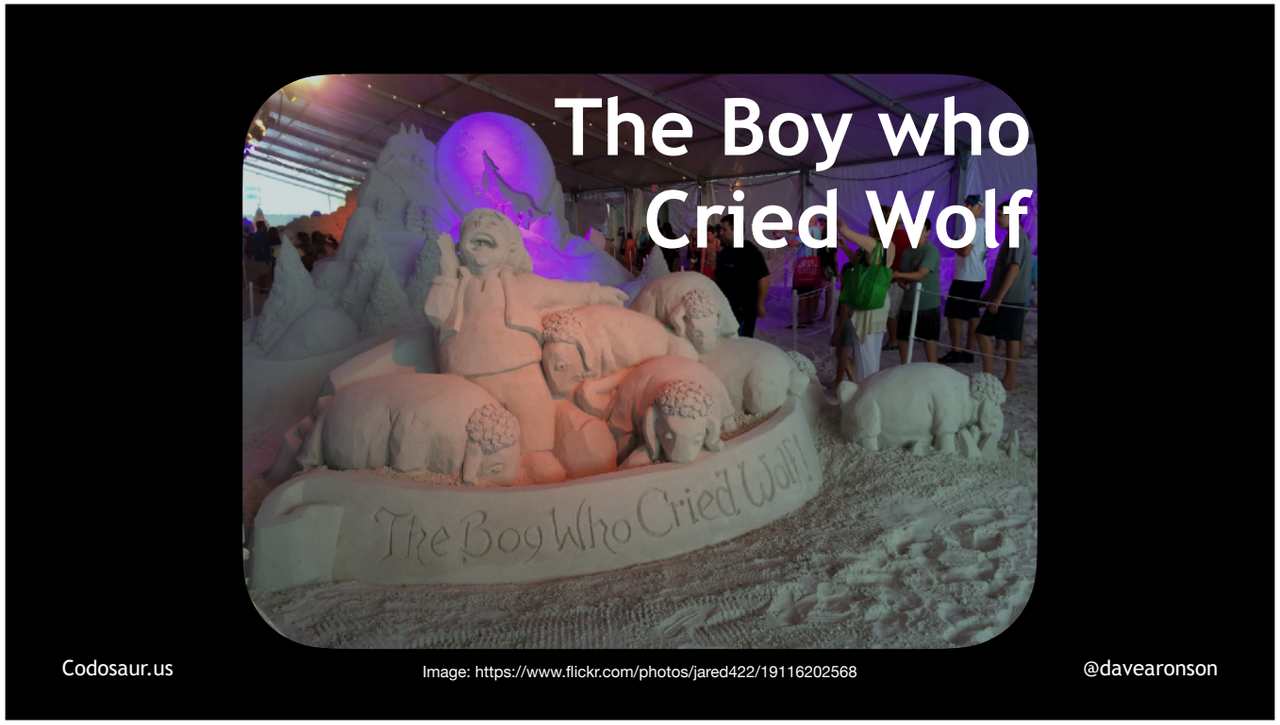

. . . hard labor for the CPU, so it's usually sssllloooowww.  We certainly won't want to sit and wait while a tool mutation-tests our whole codebase, for any non-trivial project.  But, we can let it run in the background, or while we're not even there, maybe over a lunch break for a smallish system, or a weekend for a large one.  Second, most tools let us just check specific methods, classes, files, and so on.  Third, some include an . . .




. . . incremental mode, so that we can test only the changes since the last time we ran the tool, or the last git commit, or the changes from the main branch, or some such milestone.  With filtering like that, we can test just the relevant changes, over a much shorter time.

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 surviving mutant is trying to tell us.  They're *usually* saying that our code is meaningless, or our tests are lax, or both, but it can be very hard to figure out exactly *how!*  Even worse, sometimes it's a . . .



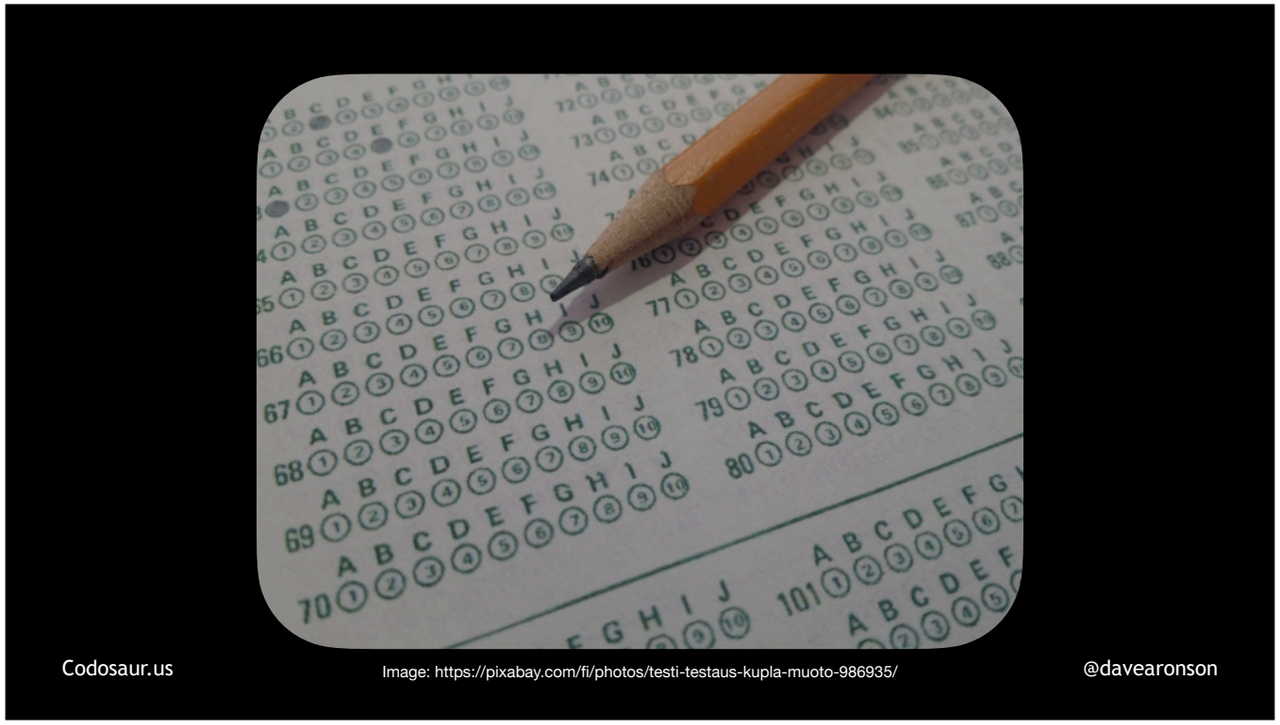

. . . false alarm, because the mutation didn't make a test fail, but it didn't make any actual difference in the first place.  It can still take quite a lot of time and effort to figure *that* out.

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


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

Now that we've seen some of the pros and cons, how does mutation testing work, unlike this guy?  First, our chosen tool . . .


. . . breaks our code apart into pieces to test.  Usually, these are our methods.  Then, for each method, it finds . . .




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


. . . makes mutants from the method.  To make mutants, the tool looks closely at the method to see how it can be changed, and for each tiny little way, the tool makes . . .


. . . one copy with that change, in other words, one mutant with *that mutation*.

Once our tool is done creating all the mutants it can for a given method, 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 method `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.  The tools generally don't give us quite all this information, let alone so neatly organized, but I find this to be a helpful conceptual model.

For each . . .

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

Codosaur.us

@davearonson

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

| Mutating method `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 method `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 method.

(PAUSE)  If any test . . .

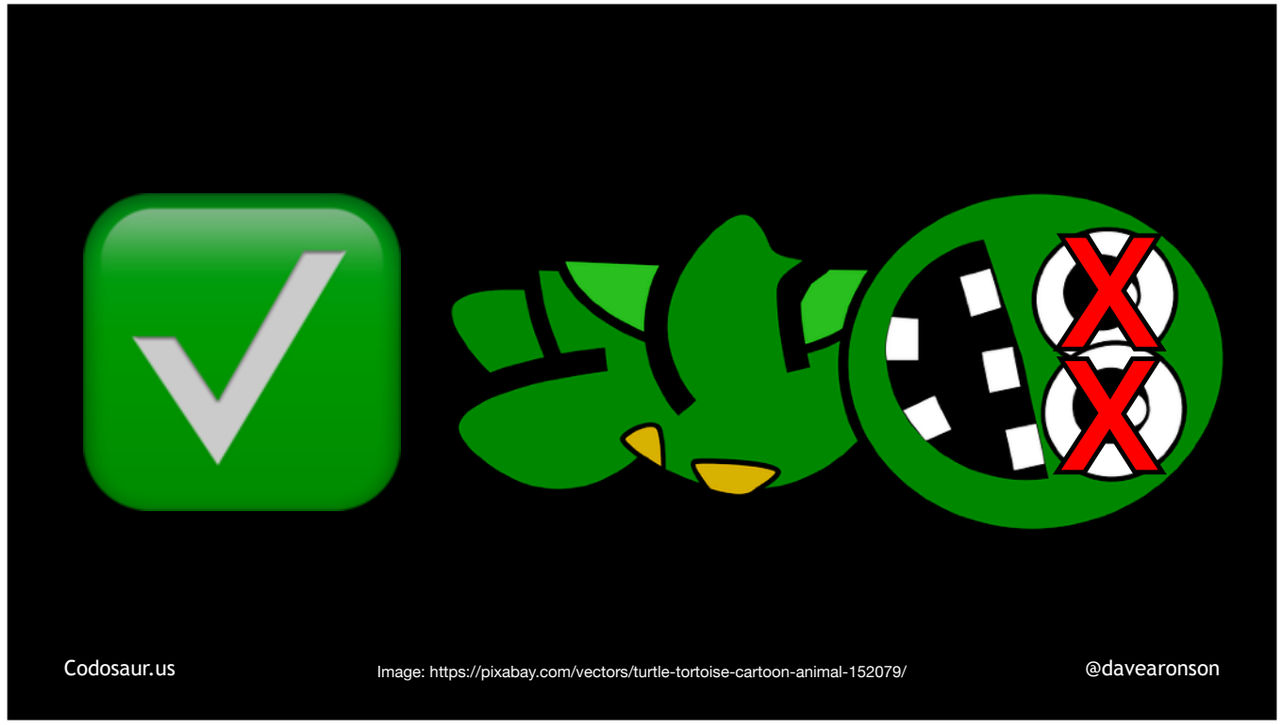
| Mutating method `whatever`, at `something.py:42` | | | | | | | | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|
| Test # / Mutant # | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | 10 | Result |
| 1 | ✔ | ✔ | ✔ | ✔ | ✘ | | | | | | In Progress |
| 2 | | | | | | | | | | | To Do |
| 3 | | | | | | | | | | | 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 our code is *meaningful* enough that the change that the tool made, to *create* this mutant, made a difference in the method's behavior, *and* that at least one test *noticed* that difference, and failed.  Then, the tool will . . .

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

Codosaur.us

@davearonson

. . . move on to the next one.  Once a mutant has made *one* test fail, we don't care how many more it *could* make fail.  Like so much in computers, we only care about ones and zeroes.

On the other claw, if a mutant . . .

| Mutating method `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 method `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 . . .

Codosaur.us

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

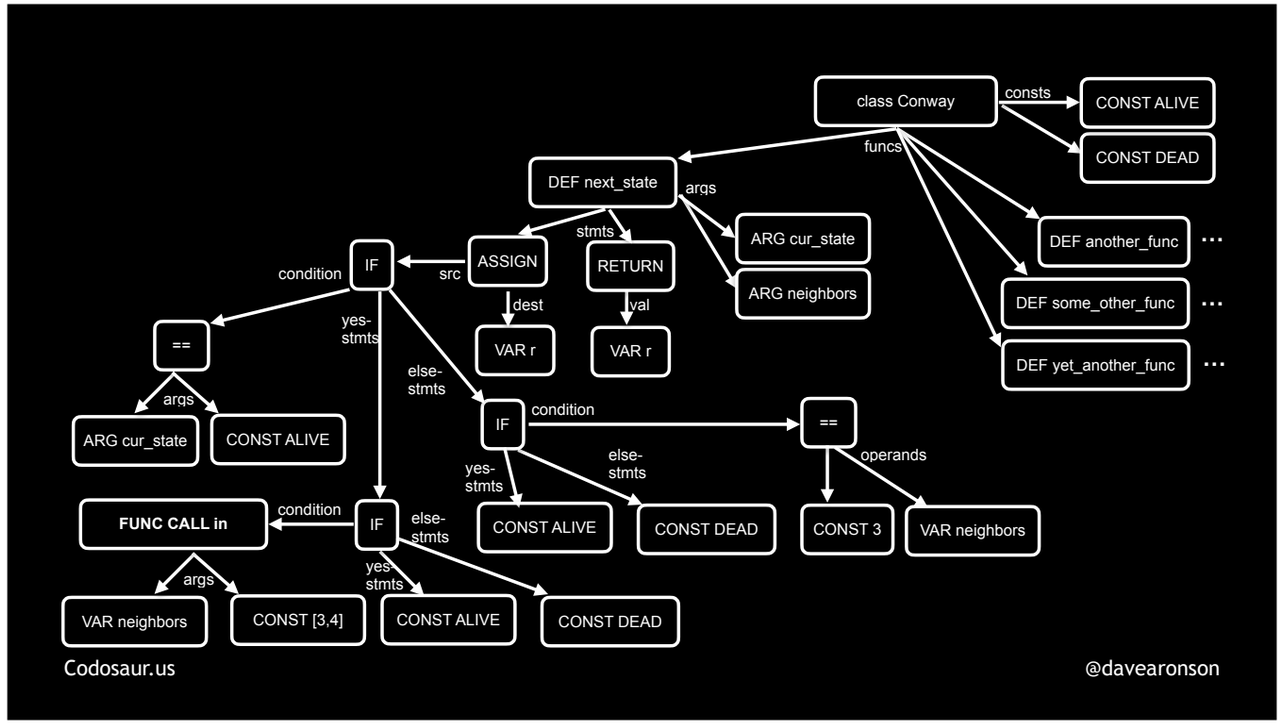
@davearonson

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

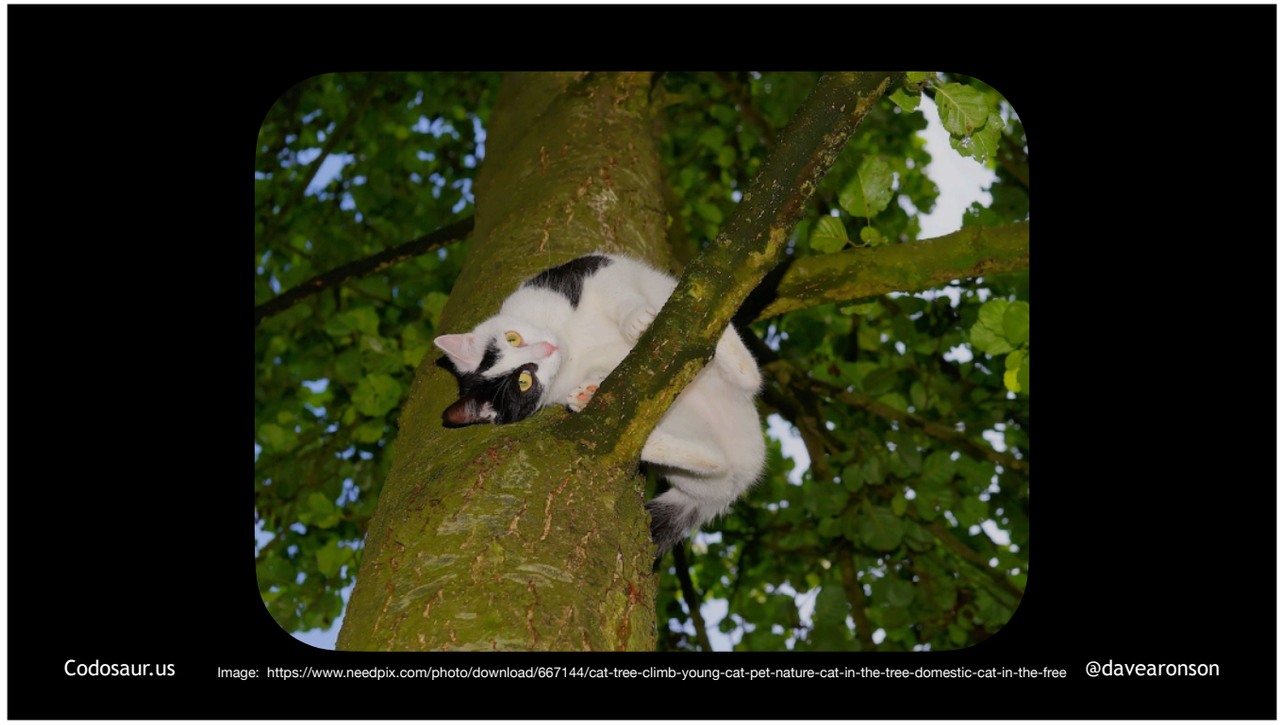
```
class Conway:
  ALIVE = "*"
  DEAD  = " "

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

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



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




. . . this Abstract Syntax Tree.  Then it . . .




. . . traverses the tree, looking for sub-trees, or branches if you will, that represent each method. After finding *them*, it looks for each one's *tests*. That usually relies mainly on us developers, either . . .

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

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



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

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

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

Codosaur.us @davearonson

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

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

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

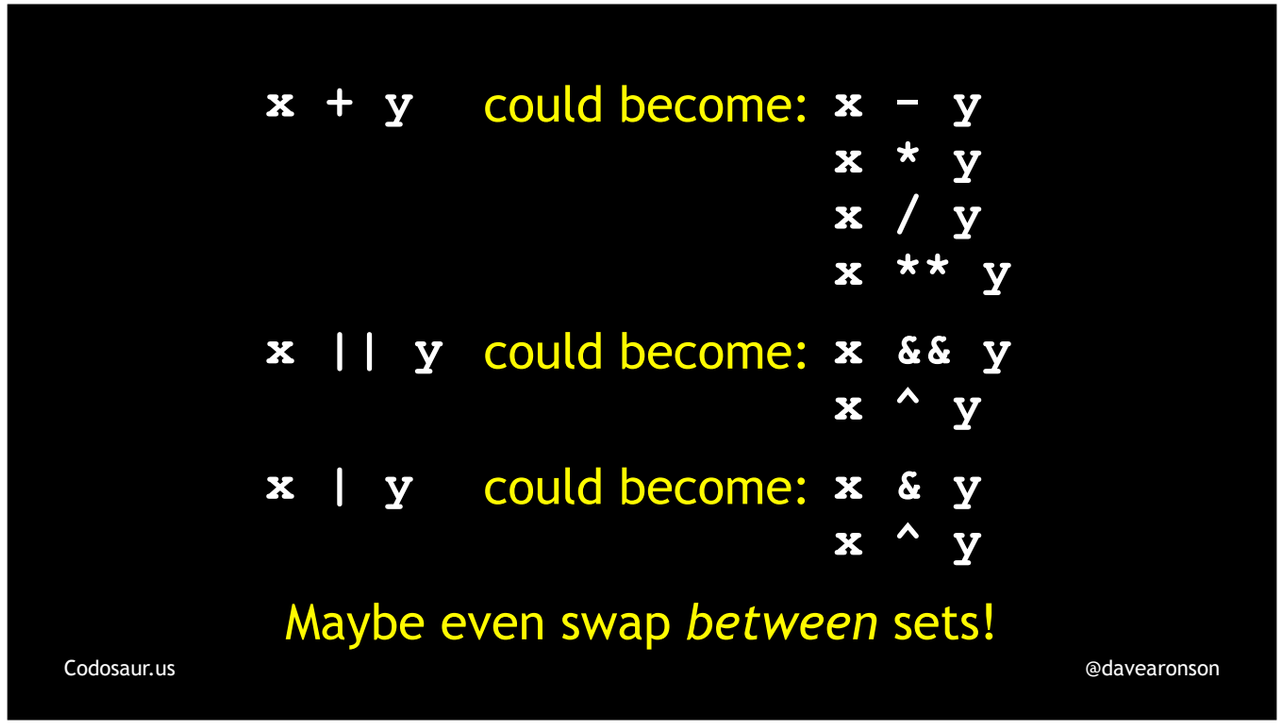
Codosaur.us

@davearonson

. . . the tool looking at what tests call what methods, out of our own codebase.  Next it makes the mutants.  To make mutants *from* an AST subtree, it . . .


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

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




It could change a mathematical, logical, or bitwise operator from one to another.  When the language allows, it could even cross these categories.

However, in the interests of speed, many programs restrict, or let *you* restrict, how different an operator it will apply, so they can stick within the category, or even try only the opposite, so x *times* y would *only* become x *divided by* y.

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

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

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

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

Codosaur.us

@davearonson

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

```
x < y
```

could become:

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

Codosaur.us

@davearonson

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

`x`

could become:

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

. . . or vice-versa!

Codosaur.us

@davearonson

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

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

could become:

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

or

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

Codosaur.us

@davearonson

It can remove an entire *statement or expression*.

```
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:

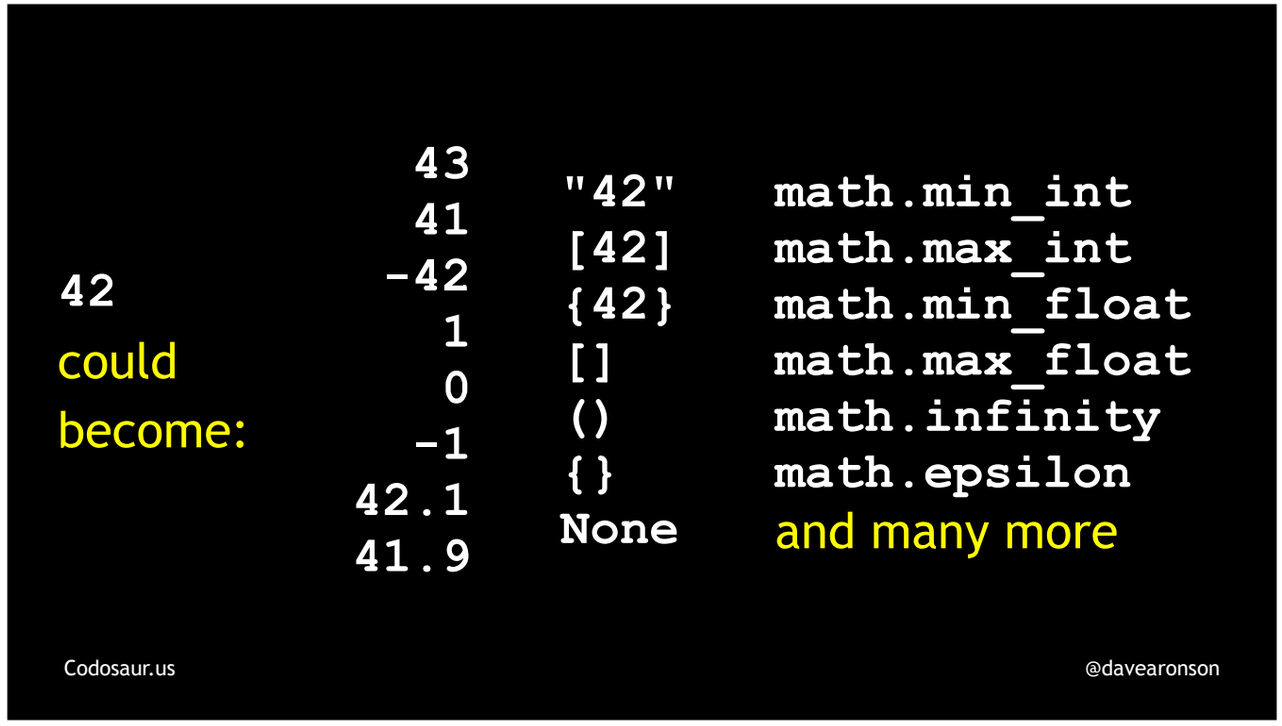
```
def f(x, y): return 0
def f(x, y): return sys.maxsize
def f(x, y): return "a string"
def f(x, y): return nil
def f(x, y): return x  # or y
def f(x, y): fail("kaboom")
def f(x, y): # nothing
```

etc.

Codosaur.us

@davearonson

It could replace a method's *entire contents* with returning a constant, or any of the arguments, or raising an error, or nothing at all.  In fact, removing the method's entire content is the basis of a very fast form, called Extreme Mutation Testing, that only does that.




It could change a value to some other value, such as changing 42 to any of these (though I realize Python doesn't actually have those math constants), and many more but I had to stop somewhere.  It could even change it to something of a different and possibly incompatible type, such as changing a number into a, if I may quote . . .

Codosaur.us

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

@davearonson

. . . Gollum, “string, or nothing!”

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

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

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

Codosaur.us @davearonson

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

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

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

```
x + y
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, again, many more but I had to stop somewhere.

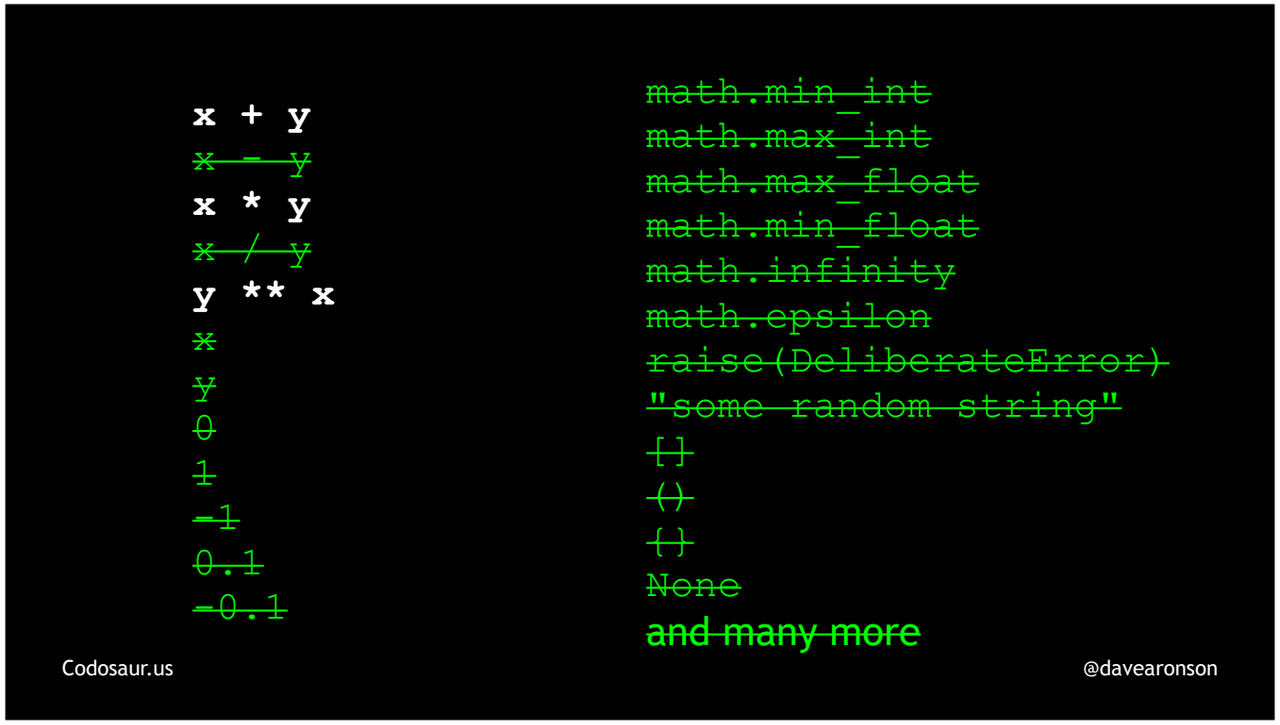
Now suppose we had only one test . . .

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

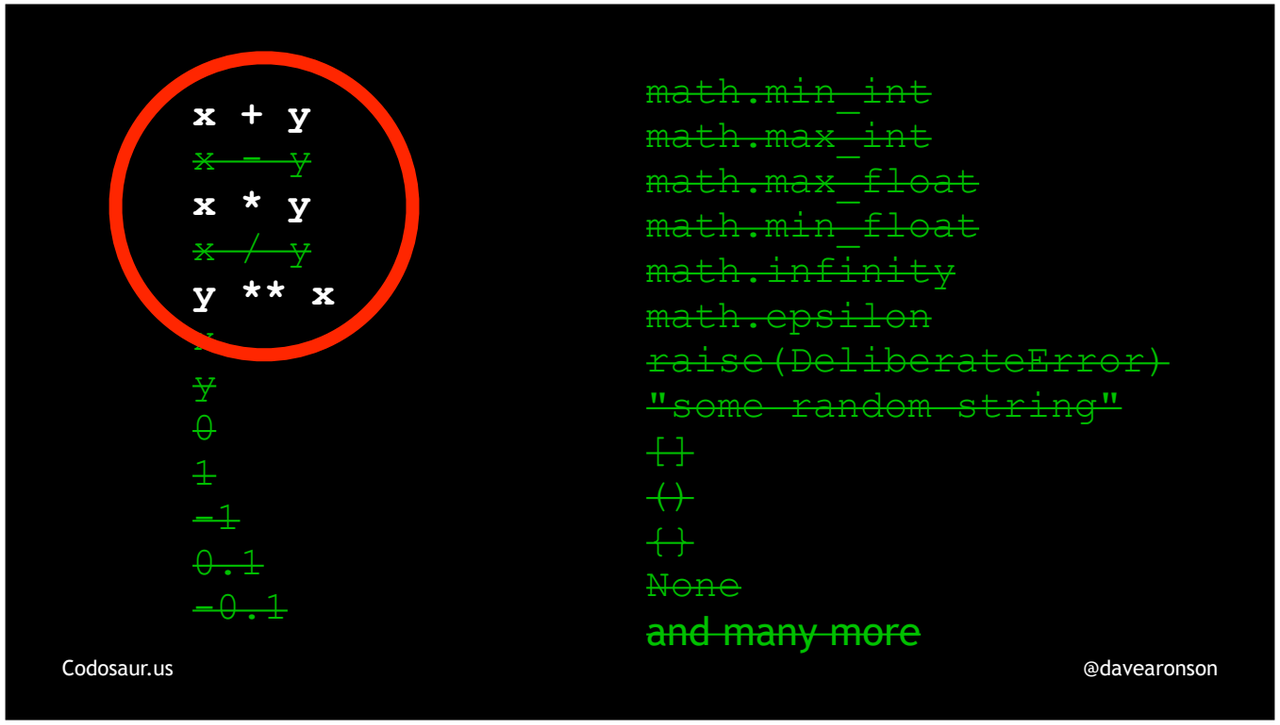
Codosaur.us

@davearonson

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




. . . here in crossed-out green.  But . . .




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

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

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

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

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

Codosaur.us

@davearonson

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

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

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

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

Codosaur.us

@davearonson

. . . the method called power, in . . .

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

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

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

Codosaur.us

@davearonson

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

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

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

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

Codosaur.us

@davearonson

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

And, that those four ways are: . . .

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

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

43 -    return x ** y
43 +    return x + y

43 -    return x ** y
43 +    return x * y

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

Codosaur.us

@davearonson

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

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

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

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

Codosaur.us

@davearonson

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

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

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

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

Codosaur.us

@davearonson

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

So what is . . .

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

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

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

Codosaur.us

@davearonson

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

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

Codosaur.us

@davearonson

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

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

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

43 -   return x ** y
43 +   return x + y

43 -   return x ** y
43 +   return x * y

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

Codosaur.us

@davearonson

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

```
mutant_power(x, y)

==

original_power(x, y)
```

Codosaur.us

@davearonson

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

the change:

```
43 -    return x ** y
43 +    return 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 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:

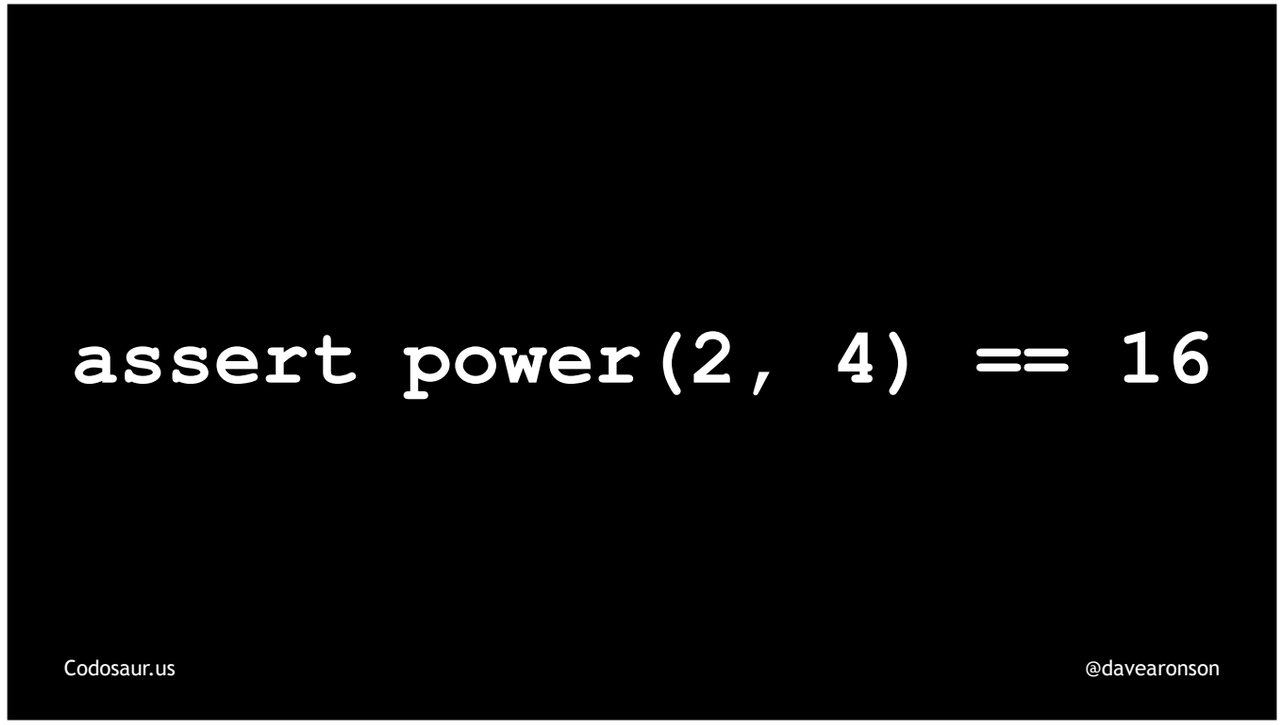
```
43 -    return x ** y
43 +    return x + y
```

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?  To do that, 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 in addition, two *plus* four is six, not six*teen*, so **this kills the plus mutant.** (See how that works?)

Better yet, two *times* four is eight, which is *also* not sixteen! We devs should certainly know our powers of two at least *that* well! So, this kills the "times" mutant as well. Killing one mutant often kills many other mutants of the same method.

But . . .


. . . the pair of argument-swapping mutants survive!  That’s because . . .

```
4 ** 2 == 16

2 ** 4 == 16
```

Codosaur.us

@davearonson

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

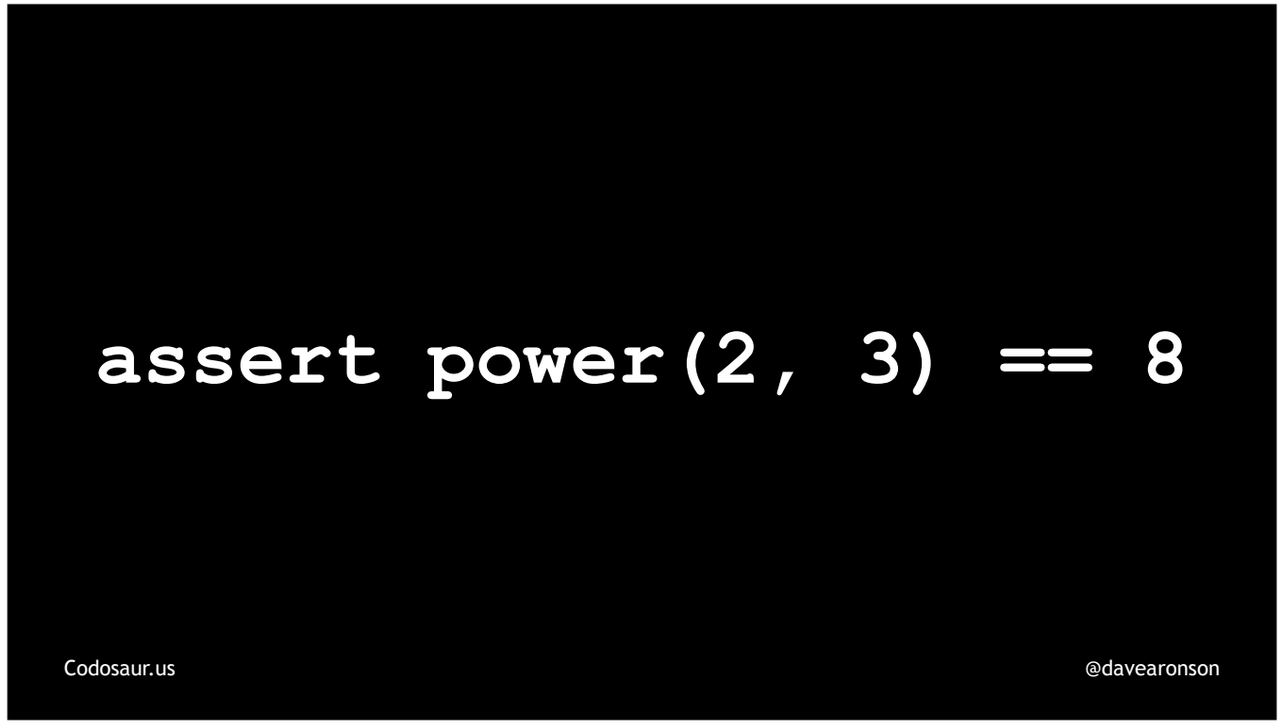

. . . attack these mutants separately, no need to kill all the mutants in one shot and be some kind of superhero about it.  To kill *them*, again, we can either add a test, or adjust an existing test, to something like . . .

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

Codosaur.us @davearonson

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


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




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


. . . *that* flerken!

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

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

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

Codosaur.us @davearonson

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

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

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

Codosaur.us

@davearonson

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

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!)

Hmmm . . . if a mutant that only goes through . . .

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

Codosaur.us @davearonson

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

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

in network.py:

```
max_chunk_size = 10_000
```

in test_send_message:

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

Codosaur.us @davearonson

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

in network.py:

```
max_chunk_size = 10_000
```

in test_send_message:

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

Codosaur.us @davearonson

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

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

in network.py:

```
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

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

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

in message.py:

```
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.  Now some other stuff is *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 *method* 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 . . .

```
def send_message(buf, len):
  # logging?
  res = send_bytes(buf, 0, len)
  # high-level error handling?
  # other record-keeping?
  return res
```

Codosaur.us @davearonson

. . . logging, error handling, and so on, needed in send_*message*, that we can't (or at least *shouldn't*) shove down the stack into send_*bytes*, but at the very least, the *looping* was redundant.  Fortunately, when it's this kind of problem, the usual solution is clear and easy, just rip out the extra junk that the mutant doesn't have.  This will also make our code more *maintainable*, by getting rid of useless cruft that just gets in the way of understanding it.

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


. . . 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?  Mutation testing has a surprisingly . . .

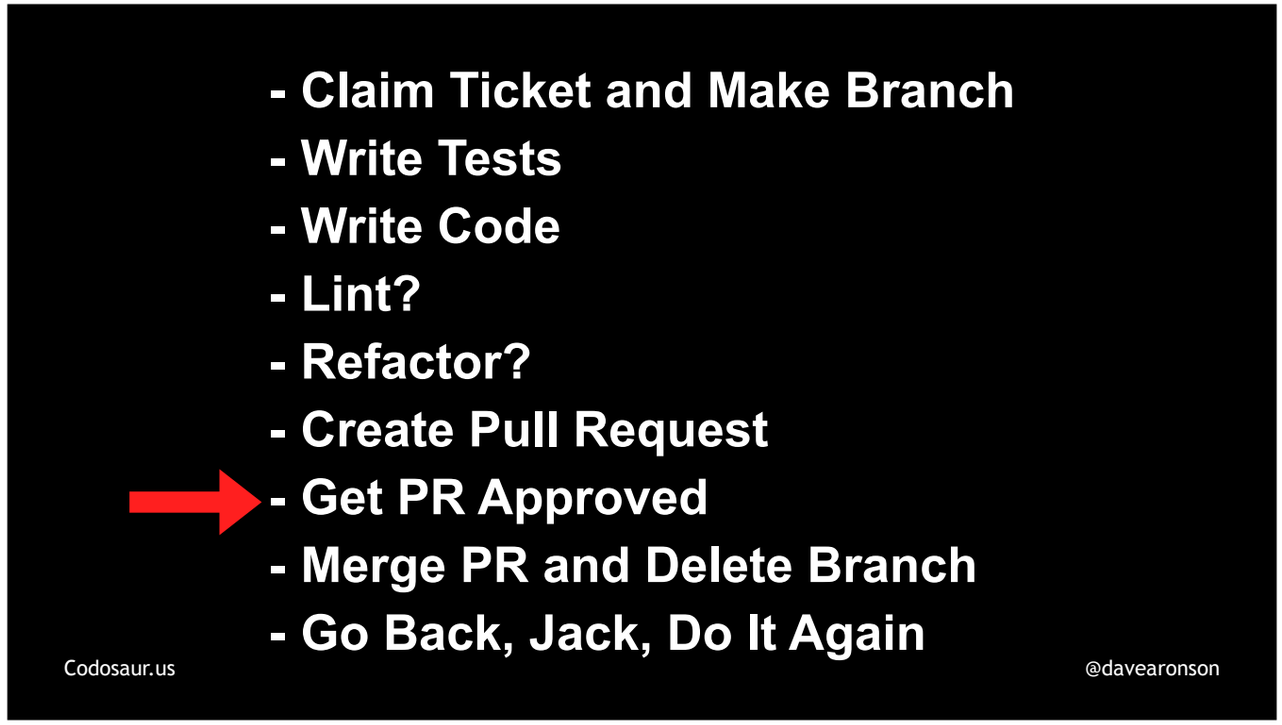

. . . long history -- at least in the context of computers.  It was first proposed in 1971, in Richard Lipton's term paper titled “Fault Diagnosis of Computer Programs”, at Carnegie-Mellon University.  The first *tool* appeared in 1980, as part of Timothy Budd's PhD work at Yale.  However, it wasn’t *practical* on typical developer-grade computers, until the early 2000s, with significant advances in CPU *speed*, multi-*core* CPUs, larger and cheaper memory, and so on.  Now, it’s practical even on fairly low-end, but still relatively modern, systems, like this 2020 MacBook Air, at least with an M1 chip, not Intel.

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

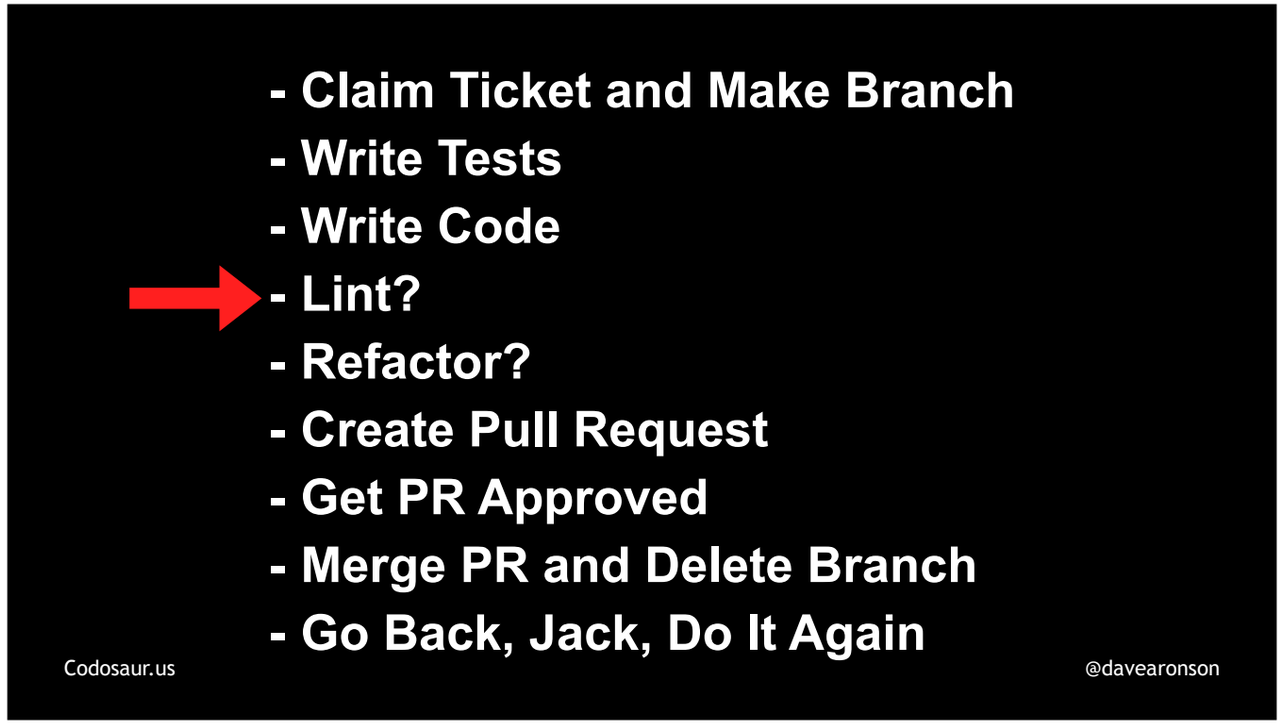
- Claim Ticket and Make Branch
- Write Tests
- Write Code
- Lint?
- Refactor?
- Create Pull Request
- Get PR Approved
- Merge PR and Delete Branch
- Go Back, Jack, Do It Again



. . . 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 accept, such as no surviving mutants on new code and no increase of them on old code.  Ideally this would be automated, as part of a CI pipeline, started automagically when the PR is created, stopping the job if the requirements 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 quality checking tools, to make sure my code is as good as possible, before making anyone else bother with it.

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


. . . here is a list of tools for Python, just in alphabetical order, no endorsements meant with the ordering.  It's been a while since I've done any mutation testing in Python, so some of these may be outdated by now, and there may be new ones.

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

# 😀 Checks that code is meaningful



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

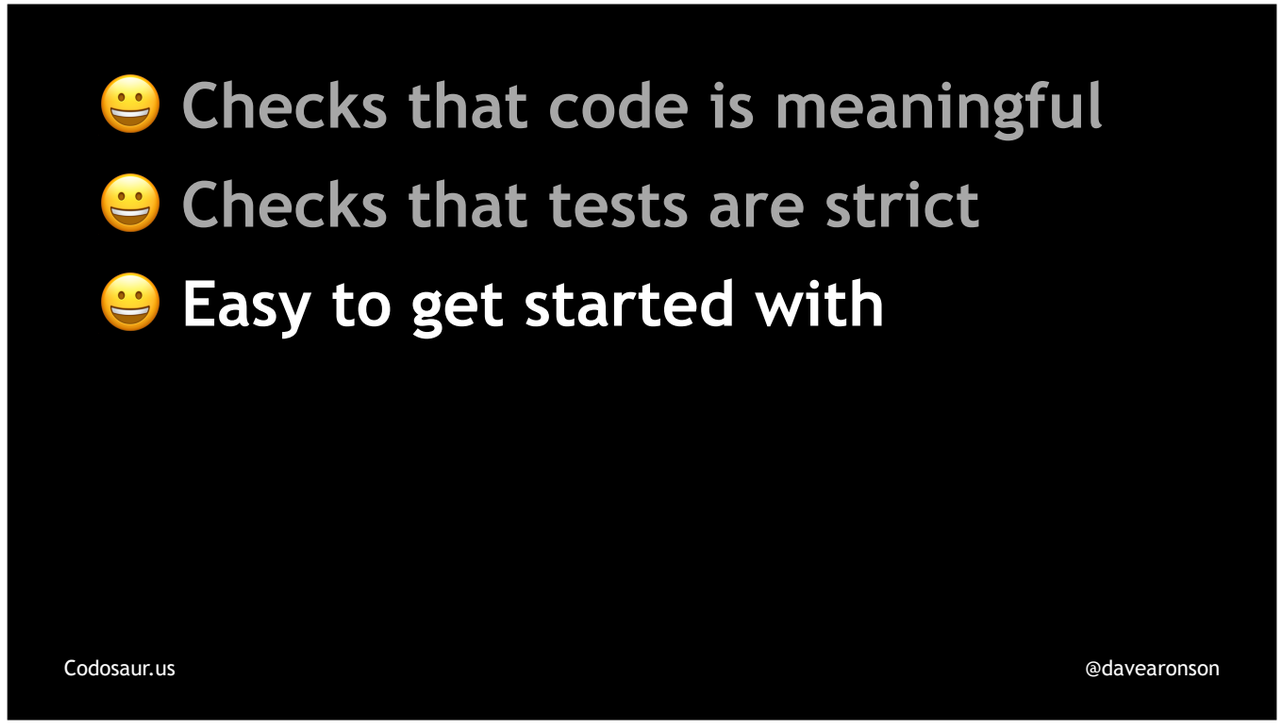
😀 Checks that code is meaningful

😀 **Checks that tests are strict**

Codosaur.us

@davearonson

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




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

😀 Checks that code is meaningful

😀 Checks that tests are strict

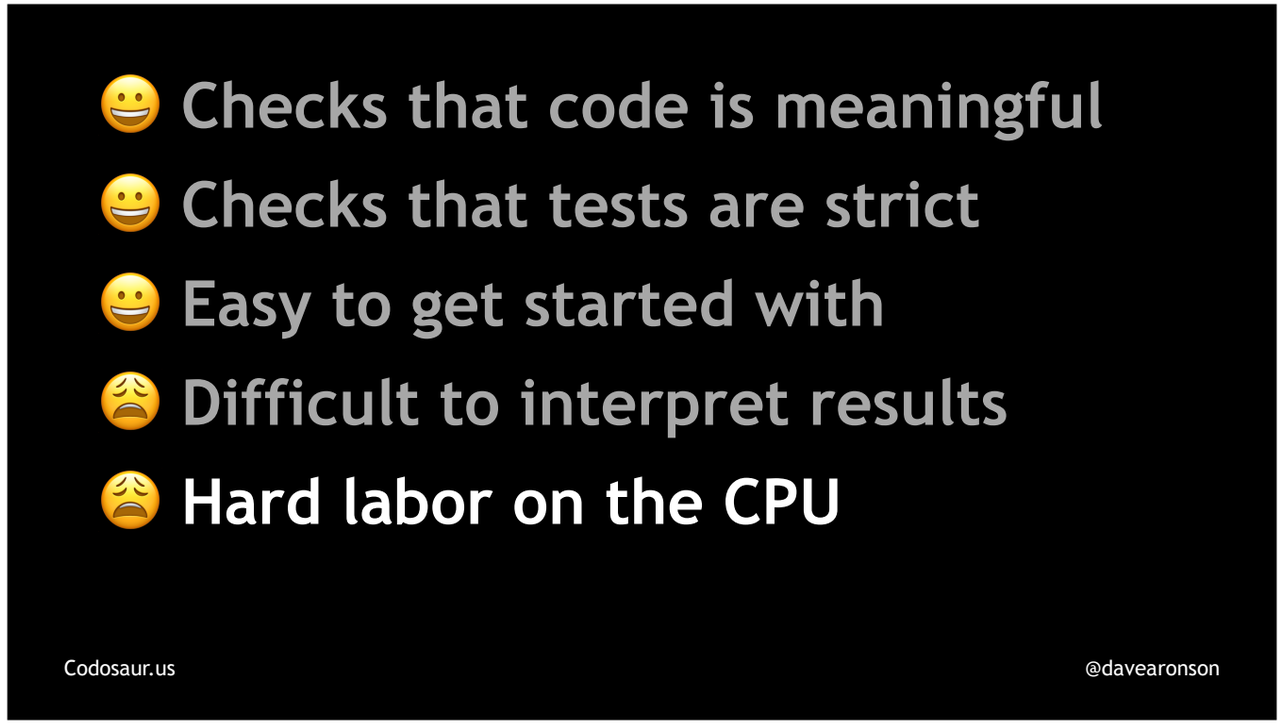
😀 Easy to get started with

😩 **Difficult to interpret results**

Codosaur.us

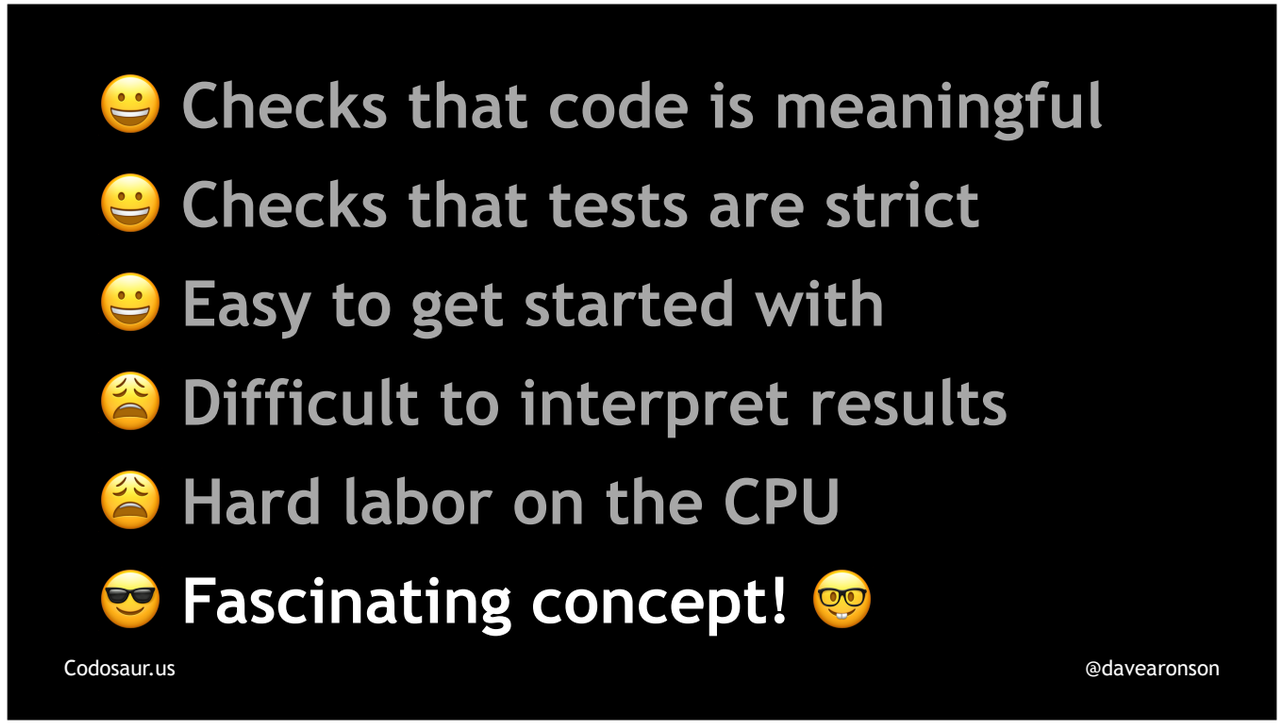
@davearonson

. . . 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 . . .




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

If you have any questions, . . .


. . . I'll take some now, and if you think of anything later, there's my contact info, plus the URL for the slides, complete with a full script, which I've *mostly* stuck to.  Any questions?