When the abyss gazes back: staring down Python's surprising internals

David Wolever
@wolever

- See README.txt for prep
- Thank you for coming!
- Exciting for me because I've been behind the scenes helping speakers get to the stage for years, but this is my first time up here myself.
- If you could tweet me some photos, that'd be great!

Overall, Python’s a pretty great language. It’s got a wonderful community, tons of packages, simple and straight forward syntax, and for the most part isn’t very surprising.

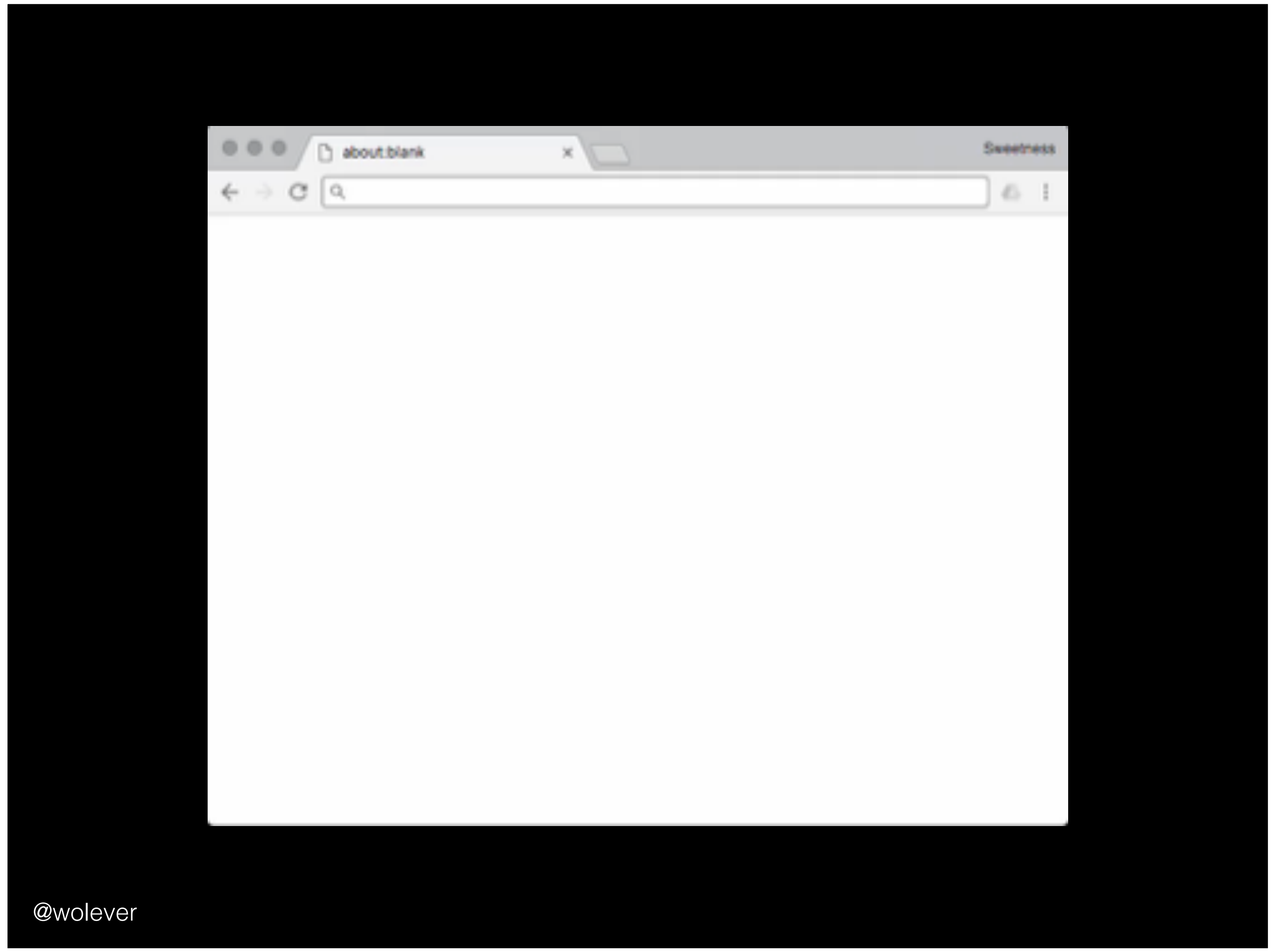
But only for the most part.

```
>>> nan = float("nan")
>>> nan is nan
True
>>> nan == nan
False
>>> nan in (nan, )
True
```

@wolever

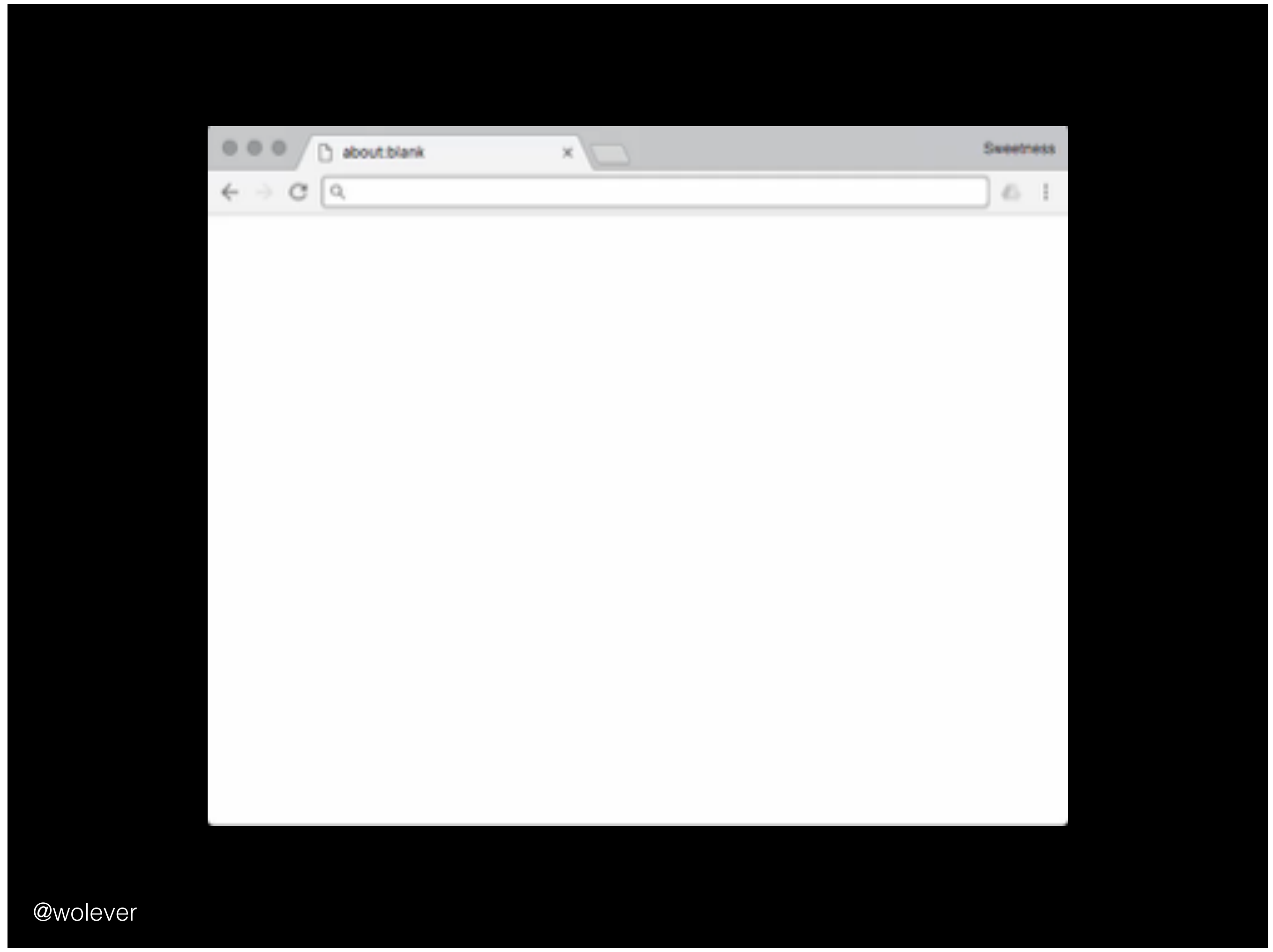
Every now and then you'll run into a weird problem. Something you can't explain, something your coworkers can't explain, and sometimes even your really smart friend you met at a meetup once can't explain.

And this is where the rubber really hits the road. Where you get to draw on all your years of experience, put those thousands of dollars you spent on school to work, and prove to yourself and the world that you're a Real Programmer™.



(video of googling and clicking a stack overflow link)

But I’m going to tell you that



(video of googling and clicking a stack overflow link)

But I’m going to tell you that

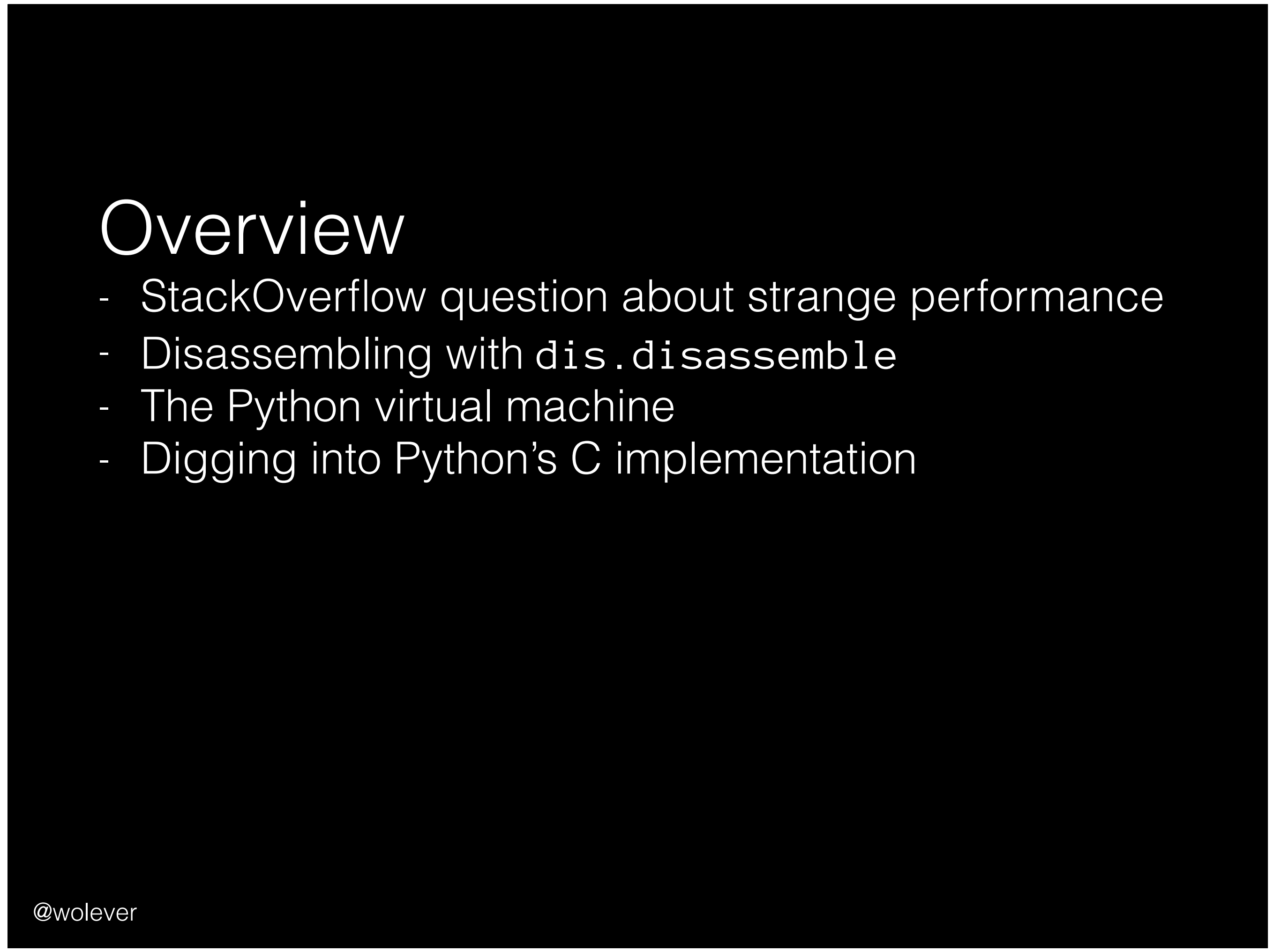
there is another way

@wolever

there is another way.

It might not be a better way, or even a very good way.

But it’s definitely an interesting way.



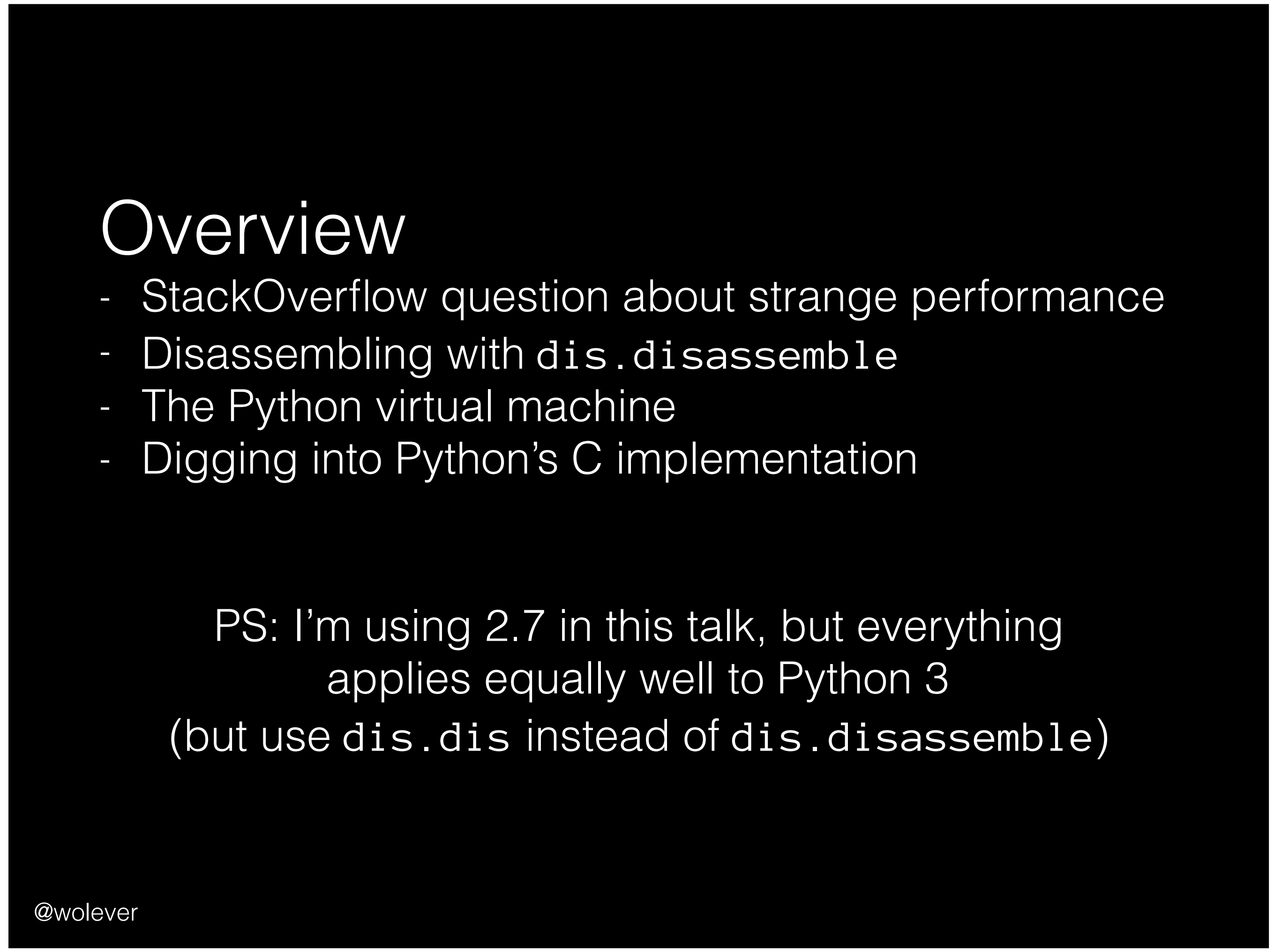
I’m going to be telling the story of answering a StackOverflow about strange performance, working mostly from first principals.

I’m going to show you how to use dis.disassemble read Python byte code, talk a little bit about the Python virtual machine, and then dive into Python’s C implementation.

By the time we’re done, I hope you’ll have some new trivia you can use to impress your friends at parties, and some practical tools you can use in your day-to-day development.

And if this all seems like old-hat to you, I won’t be offended if you want to go somewhere else :)

… and on that note, I’m going to be using Python 2.7, but everything applies equally well to Python 3.



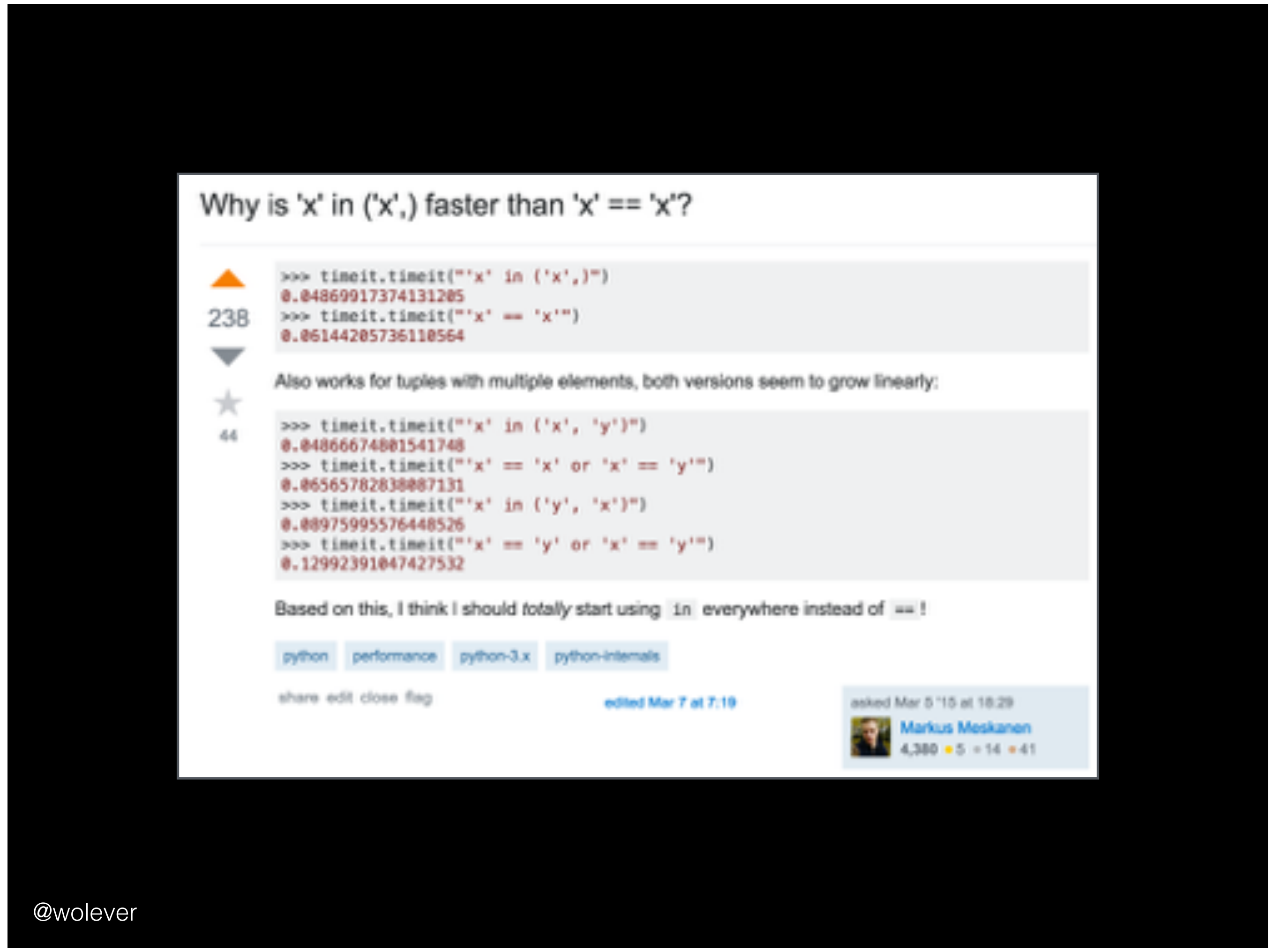
I'm going to be telling the story of answering a StackOverflow about strange performance, working mostly from first principals.

I'm going to show you how to use dis.disassemble read Python byte code, talk a little bit about the Python virtual machine, and then dive into Python's C implementation.

By the time we're done, I hope you'll have some new trivia you can use to impress your friends at parties, and some practical tools you can use in your day-to-day development.

And if this all seems like old-hat to you, I won't be offended if you want to go somewhere else :)

… and on that note, I'm going to be using Python 2.7, but everything applies equally well to Python 3.



Here’s the StackOverflow question that caught my eye.

At first it seems very strange – equality is about the simplest operation you could perform, yet it’s (marginally) slower than creating a tuple and testing for membership!

```
In [1]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [2]: %timeit 'x' == 'x'
10000000 loops, best of 3: 31.3 ns per loop

In [3]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 29.5 ns per loop

In [4]: %timeit 'x' == 'x'
10000000 loops, best of 3: 30.7 ns per loop
```

@wolever

Of course, the first thing I did was fire up IPython and check this for myself… and I was able to consistently reproduce the result.

And by the way…

```
In [5]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [6]: %timeit "x" * 10000
The slowest run took 10.08 times longer than the
fastest. This could mean that an intermediate
result is being cached
1000000 loops, best of 3: 213 ns per loop

In [7]: %timeit open("/dev/null").close()
100000 loops, best of 3: 3.86 µs per loop

In [8]: %timeit open("/dev/zero").read(1024**2)
10000 loops, best of 3: 93.8 µs per loop
```

@wolever

… ipython’s %timeit magic is incredibly useful. It automatically figures out how many iterations to run…

```
In [5]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [6]: %timeit "x" * 10000
The slowest run took 10.08 times longer than the
fastest. This could mean that an intermediate
result is being cached
1000000 loops, best of 3: 213 ns per loop

In [7]: %timeit open("/dev/null").close()
100000 loops, best of 3: 3.86 µs per loop

In [8]: %timeit open("/dev/zero").read(1024**2)
10000 loops, best of 3: 93.8 µs per loop
```

@wolever

… based on the speed of the operation you're timing …

```
In [5]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [6]: %timeit "x" * 10000
The slowest run took 10.08 times longer than the
fastest. This could mean that an intermediate
result is being cached
1000000 loops, best of 3: 213 ns per loop

In [7]: %timeit open("/dev/null").close()
100000 loops, best of 3: 3.86 µs per loop

In [8]: %timeit open("/dev/zero").read(1024**2)
10000 loops, best of 3: 93.8 µs per loop
```

@wolever

```
In [5]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [6]: %timeit "x" * 10000
The slowest run took 10.08 times longer than the
fastest. This could mean that an intermediate
result is being cached
1000000 loops, best of 3: 213 ns per loop

In [7]: %timeit open("/dev/null").close()
100000 loops, best of 3: 3.86 µs per loop

In [8]: %timeit open("/dev/zero").read(1024**2)
10000 loops, best of 3: 93.8 µs per loop
```

@wolever

```
In [5]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [6]: %timeit "x" * 10000
The slowest run took 10.08 times longer than the
fastest. This could mean that an intermediate
result is being cached
1000000 loops, best of 3: 213 ns per loop

In [7]: %timeit open("/dev/null").close()
100000 loops, best of 3: 3.86 µs per loop

In [8]: %timeit open("/dev/zero").read(1024**2)
10000 loops, best of 3: 93.8 µs per loop
```

@wolever

… instead of blindly picking one million, like the Python 2 timeit module (this is better in Python 3, though).

But getting back to our problem:

```
In [1]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [2]: %timeit 'x' == 'x'
10000000 loops, best of 3: 31.3 ns per loop

In [3]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 29.5 ns per loop

In [4]: %timeit 'x' == 'x'
10000000 loops, best of 3: 30.7 ns per loop
```

@wolever

We’ve been able to reproduce the result…

```
In [1]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [2]: %timeit 'x' == 'x'
10000000 loops, best of 3: 31.3 ns per loop

In [3]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 29.5 ns per loop

In [4]: %timeit 'x' == 'x'
10000000 loops, best of 3: 30.7 ns per loop
```

@wolever

… but this raises the question: if we’re not using Google …

where do we start?

@wolever

dis.disassemble!

dis.disassemble lets you disassemble Python code and see the underlying byte code.

Now, you’ve probably heard Python talked about as an interpreted language, in contrast with compiled languages like C++ or Java.

But this isn’t strictly true; Python does have a compiler which is automatically run over every .py file when it’s imported or executed. The compiler takes plain Python code – the stuff you write – and compiles it to Python Byte Code.

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

And dis.disassemble lets you disassemble that bytecode, showing a roughly human-readable translation.

Here’s an example!

Now, there’s a lot going on here, so let’s walk through it one step at a time

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

First, the three lines you're already familiar with: importing dis, setting a variable, defining a function

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2     0 LOAD_CONST          1 ('Hello, %s')
        3 LOAD_GLOBAL         0 (what)
        6 BUILD_TUPLE         1
        9 BINARY_MODULO
       10 STORE_FAST          0 (msg)

  3    13 LOAD_FAST           0 (msg)
       16 PRINT_ITEM
       17 PRINT_NEWLINE
       18 LOAD_CONST          0 (None)
       21 RETURN_VALUE
```

@wolever

Next we’ve got the call to dis.disassemble… and this is where things start to get interesting

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2     0 LOAD_CONST          1 ('Hello, %s')
        3 LOAD_GLOBAL         0 (what)
        6 BUILD_TUPLE         1
        9 BINARY_MODULO
       10 STORE_FAST          0 (msg)

  3    13 LOAD_FAST           0 (msg)
       16 PRINT_ITEM
       17 PRINT_NEWLINE
       18 LOAD_CONST          0 (None)
       21 RETURN_VALUE
```

@wolever

What is this func_code attribute?

To understand, we need to dig into function objects a little bit:

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)

        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

What is this func_code attribute?

To understand, we need to dig into function objects a little bit:

What is this func_code attribute?

To understand, we need to dig into function objects a little bit:

function objects

```
In [6]: say_hello
Out[6]: <function say_hello at 0x…>

In [7]: dir(say_hello)
Out[7]:[
    …,
    'func_code',
    'func_closure',
    # exercise for the reader:
    'func_globals',
    'func_defaults',
]
```

@wolever

Functions, like everything else in Python, are objects with a bunch of attributes, and we can use the `dir` builtin to list the attributes.

There are a whole bunch of fascinating things in there – after this talk I’d encourage you to try and figure out what func_globals and func_defaults do – but for now…

function objects

```
In [6]: say_hello
Out[6]: <function say_hello at 0x…>

In [7]: dir(say_hello)
Out[7]:[
    …,
    'func_code',
    'func_closure',
    # exercise for the reader:
    'func_globals',
    'func_defaults',
]
```

@wolever

… we're going to start with func_code (in Python 3, __code__).

```
In [6]: say_hello.func_code
Out[6]: <code object say_hello at 0x…, file
  "<ipython-input>", line 1>

In [7]: dir(say_hello.func_code)
Out[7]:[
    'co_argcount', 'co_cellvars', 'co_code',
    'co_consts', 'co_filename', 'co_firstlineno',
    'co_flags', 'co_freevars', 'co_lnotab',
    'co_name', 'co_names', 'co_nlocals',
    'co_stacksize', 'co_varnames',
]
```

@wolever

func_code is the object which describes the code associated with the function, and it has some interesting information associated with it:

```
In [6]: say_hello.func_code
Out[6]: <code object say_hello at 0x…, file
  "<ipython-input>", line 1>

In [7]: dir(say_hello.func_code)
Out[7]:[
    'co_argcount', 'co_cellvars', 'co_code',
    'co_consts', 'co_filename', 'co_firstlineno',
    'co_flags', 'co_freevars', 'co_lnotab',
    'co_name', 'co_names', 'co_nlocals',
    'co_stacksize', 'co_varnames',
]
```

@wolever

… like the function’s name, the file in which it was defined (which, in this case, is ipython), and even the line number.

```
In [6]: say_hello.func_code
Out[6]: <code object say_hello at 0x…, file
  "<ipython-input>", line 1>

In [7]: dir(say_hello.func_code)
Out[7]:[
    'co_argcount', 'co_cellvars', 'co_code',
    'co_consts', 'co_filename', 'co_firstlineno',
    'co_flags', 'co_freevars', 'co_lnotab',
    'co_name', 'co_names', 'co_nlocals',
    'co_stacksize', 'co_varnames',
]
```

@wolever

Again, I'd really encourage you to poke around inside func_code - there's some really neat stuff in there, and you can fool around with it to do some really, uh, interesting things.

But for now, we're going to be focusing on…

```
In [6]: say_hello.func_code
Out[6]: <code object say_hello at 0x…, file
  "<ipython-input>", line 1>

In [7]: dir(say_hello.func_code)
Out[7]:[
    'co_argcount', 'co_cellvars', 'co_code',
    'co_consts', 'co_filename', 'co_firstlineno',
    'co_flags', 'co_freevars', 'co_lnotab',
    'co_name', 'co_names', 'co_nlocals',
    'co_stacksize', 'co_varnames',
]
```

@wolever

… the co_code attribute.

(these are the same in Python 3)

```
In [8]: say_hello.func_code.co_code
Out[8]: 'd\x01\x00t\x00\x00f\x01\x00\x16}\x00\x00|
\x00\x00GHd\x00\x00S'
```

@wolever

co_code is a string containing the same compiled bytecode that you would find in a .pyc file. This byte code contains the full implementation of the function, which we can see if we use dis to disassemble it:

```
In [8]: say_hello.func_code.co_code
Out[8]: 'd\x01\x00t\x00\x00f\x01\x00\x16}\x00\x00|
\x00\x00GHd\x00\x00S'

In [9]: dis.disassemble_string(_8)
      0 LOAD_CONST          1 (1)
      3 LOAD_GLOBAL         0 (0)
      6 BUILD_TUPLE         1
      9 BINARY_MODULO
     10 STORE_FAST          0 (0)
     13 LOAD_FAST           0 (0)
     16 PRINT_ITEM
     17 PRINT_NEWLINE
     18 LOAD_CONST          0 (0)
     21 RETURN_VALUE
```

@wolever

Notice that this is the same disassembly as before, albeit without the line numbers or variable names.

The observant amount you, though, will notice…

```
In [8]: say_hello.func_code.co_code
Out[8]: 'd\x01\x00t\x00\x00f\x01\x00\x16}\x00\x00|
\x00\x00GHd\x00\x00S'

In [9]: dis.disassemble_string(_8)
```

Homework: where’s the `"Hello, %s!"`?

```
      9 BINARY_MODULO
     10 STORE_FAST           0 (0)
     13 LOAD_FAST            0 (0)
     16 PRINT_ITEM
     17 PRINT_NEWLINE
     18 LOAD_CONST           0 (0)
     21 RETURN_VALUE
```

@wolever

the conspicuous absence of the "Hello, %s" format string.

If it’s not in the byte code, where is it?

function objects

```
In [6]: say_hello
Out[6]: <function say_hello at 0x…>

In [7]: dir(say_hello)
Out[7]:[
    …,
    'func_code',
    'func_closure',
    # exercise for the reader:
    'func_globals',
    'func_defaults',
]
```

@wolever

And now, if we’re doing alright for time, I want to take a small detour into a second function attribute: …

Aside: `func_closure`

```
In [6]: say_hello
Out[6]: <function say_hello at 0x…>

In [7]: dir(say_hello)
Out[7]:[
    …,
    'func_code',
    'func_closure',
    # exercise for the reader:
    'func_globals',
    'func_defaults',
]
```

@wolever

… func_closure (or __closure__)

Aside: func_closure

```
In [10]: def hello_closure(what):
   ....:  msg = "Hello, %s!" %(what, )
   ....:  def hello_closure_inner():
   ....:    return msg
   ....:  return hello_closure_inner

In [11]: say_hello = hello_closure("World")

In [12]: say_hello()
Out[12]: 'Hello, World!'
```

@wolever

For the unfamiliar, a closure is a function which store references to variables which were in scope when the function was created, but aren’t part of the function its self.

For example:

Aside: `func_closure`

```
In [10]: def hello_closure(what):
   ....:   msg = "Hello, %s!" %(what, )
   ....:   def hello_closure_inner():
   ....:     return msg
   ....:   return hello_closure_inner

In [11]: say_hello = hello_closure("World")

In [12]: say_hello()
Out[12]: 'Hello, World!'
```

@wolever

The hello_closure_inner function references the "msg" variable, even though it’s not defined in the function, or passed in as an argument.

Aside: `func_closure`

```
In [10]: def hello_closure(what):
   ....:   msg = "Hello, %s!" %(what, )
   ....:   def hello_closure_inner():
   ....:     return msg
   ....:   return hello_closure_inner

In [11]: say_hello = hello_closure("World")

In [12]: say_hello()
Out[12]: 'Hello, World!'
```

@wolever

It’s defined here, outside the function

Aside: `func_closure`

```
In [10]: def hello_closure(what):
   ....:  msg = "Hello, %s!" %(what, )
   ....:  def hello_closure_inner():
   ....:    return msg
   ....:  return hello_closure_inner

In [11]: say_hello = hello_closure("World")

In [12]: say_hello()
Out[12]: 'Hello, World!'
```

@wolever

And the hello_closure_inner function can keep referencing that variable even after it’s been returned.

Aside: func_closure

```
In [10]: def hello_closure(what):
   ....:  msg = "Hello, %s!" %(what, )
   ....:  def hello_closure_inner():
   ....:    return msg
   ....:  return hello_closure_inner

In [11]: say_hello = hello_closure("World")

In [12]: say_hello()
Out[12]: 'Hello, World!'
```

@wolever

So, how does that work in Python?

Aside: function objects

```
In [12]: say_hello()
Out[12]: 'Hello, World!'

In [13]: say_hello.func_closure
Out[13]: (<cell at 0x…: str object at 0x…>, )

In [14]: say_hello.func_closure[0].cell_contents
Out[14]: 'World'
```

@wolever

The func_closure attribute!

It contains a tuple of all the variables that are being "closed over"; that is, "used by the function".

(Well, actually it’s a tuple of "cells" which reference the values being closed over… this makes it possible for the containing scope to update the value of the variable… but that’s a story for another time)

One neat consequence of this is that it is actually possible (… at least in theory) to serialize closures. But that’s a *bad idea* and you *definitely* shouldn’t do it… and you really, *absolutely* shouldn’t turn it into a package and put that package up on PyPI.

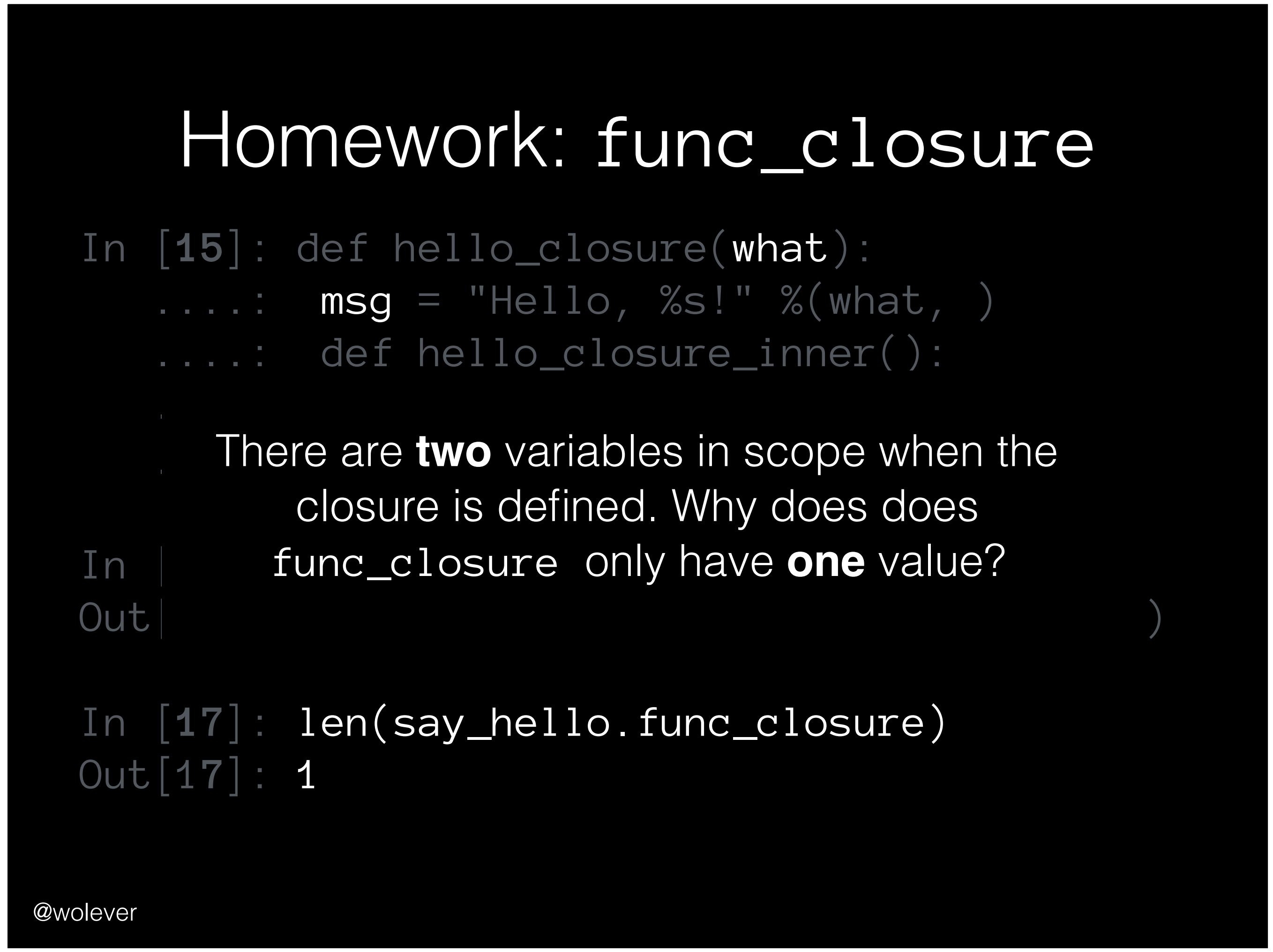
Now, your homework…

Homework: func_closure

```
In [15]: def hello_closure(what):
    ....:  msg = "Hello, %s!" %(what, )
    ....:  def hello_closure_inner():
    ....:    return msg
    ....:  return hello_closure_inner

In [16]: say_hello.func_closure
Out[16]: (<cell at 0x…: str object at 0x…>, )

In [17]: len(say_hello.func_closure)
Out[17]: 1
```

@wolever

… is to figure out why, even though there are two variables that are in scope…



… when the closure is defined, why does the func_closure tuple have only one value?

WELL, that was a fun digression, but getting back on track:

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

This is the line we were looking at before we got distracted.

Now we know a little bit more about what it means to disassemble code, so let’s start looking at the output of the disassembly:

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

Even if you’ve never seen disassembled code before, it’s not too hard to guess what’s going on:

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

The 'hello' format string is loaded (whatever "loading" means)

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

The value of the 'what' variable is loaded

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2       0 LOAD_CONST               1 ('Hello, %s')
          3 LOAD_GLOBAL              0 (what)
          6 BUILD_TUPLE              1
          9 BINARY_MODULO
         10 STORE_FAST               0 (msg)

  3      13 LOAD_FAST                0 (msg)
         16 PRINT_ITEM
         17 PRINT_NEWLINE
         18 LOAD_CONST               0 (None)
         21 RETURN_VALUE
```

@wolever

A tuple is created

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2     0 LOAD_CONST          1 ('Hello, %s')
        3 LOAD_GLOBAL         0 (what)
        6 BUILD_TUPLE         1
        9 BINARY_MODULO
       10 STORE_FAST          0 (msg)

  3    13 LOAD_FAST           0 (msg)
       16 PRINT_ITEM
       17 PRINT_NEWLINE
       18 LOAD_CONST          0 (None)
       21 RETURN_VALUE
```

@wolever

The modulo operator is used to format the string

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

and the result is stored in the 'msg' variable.

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2       0 LOAD_CONST               1 ('Hello, %s')
          3 LOAD_GLOBAL              0 (what)
          6 BUILD_TUPLE              1
          9 BINARY_MODULO
         10 STORE_FAST               0 (msg)

  3      13 LOAD_FAST                0 (msg)
         16 PRINT_ITEM
         17 PRINT_NEWLINE
         18 LOAD_CONST               0 (None)
         21 RETURN_VALUE
```

@wolever

Next, the value of the 'msg' variable is loaded

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

And then printed

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2      0 LOAD_CONST          1 ('Hello, %s')
         3 LOAD_GLOBAL         0 (what)
         6 BUILD_TUPLE         1
         9 BINARY_MODULO
        10 STORE_FAST          0 (msg)

  3     13 LOAD_FAST           0 (msg)
        16 PRINT_ITEM
        17 PRINT_NEWLINE
        18 LOAD_CONST          0 (None)
        21 RETURN_VALUE
```

@wolever

And None is returned (because we didn't define an explicit return value)

```
In [1]: import dis
In [2]: what = "World"
In [3]: def say_hello():
   ...:     msg = "Hello, %s" %(what, )
   ...:     print msg
In [4]: dis.disassemble(say_hello.func_code)
  2       0 LOAD_CONST               1 ('Hello, %s')
          3 LOAD_GLOBAL              0 (what)
          6 BUILD_TUPLE              1
          9 BINARY_MODULO
         10 STORE_FAST               0 (msg)

  3      13 LOAD_FAST                0 (msg)
         16 PRINT_ITEM
         17 PRINT_NEWLINE
         18 LOAD_CONST               0 (None)
         21 RETURN_VALUE
```

@wolever

Now, to touch in a tiny bit more detail on what's going on here:

Aside: Stack Machines

@wolever

You’ve probably noticed that the byte code instructions take at most one argument.

This is because the byte code interpreter – also called a virtual machine – is a particular kind of machine called a stack machine: in a stack machine, instructions operate on values by pushing them to, or popping them from, a stack.

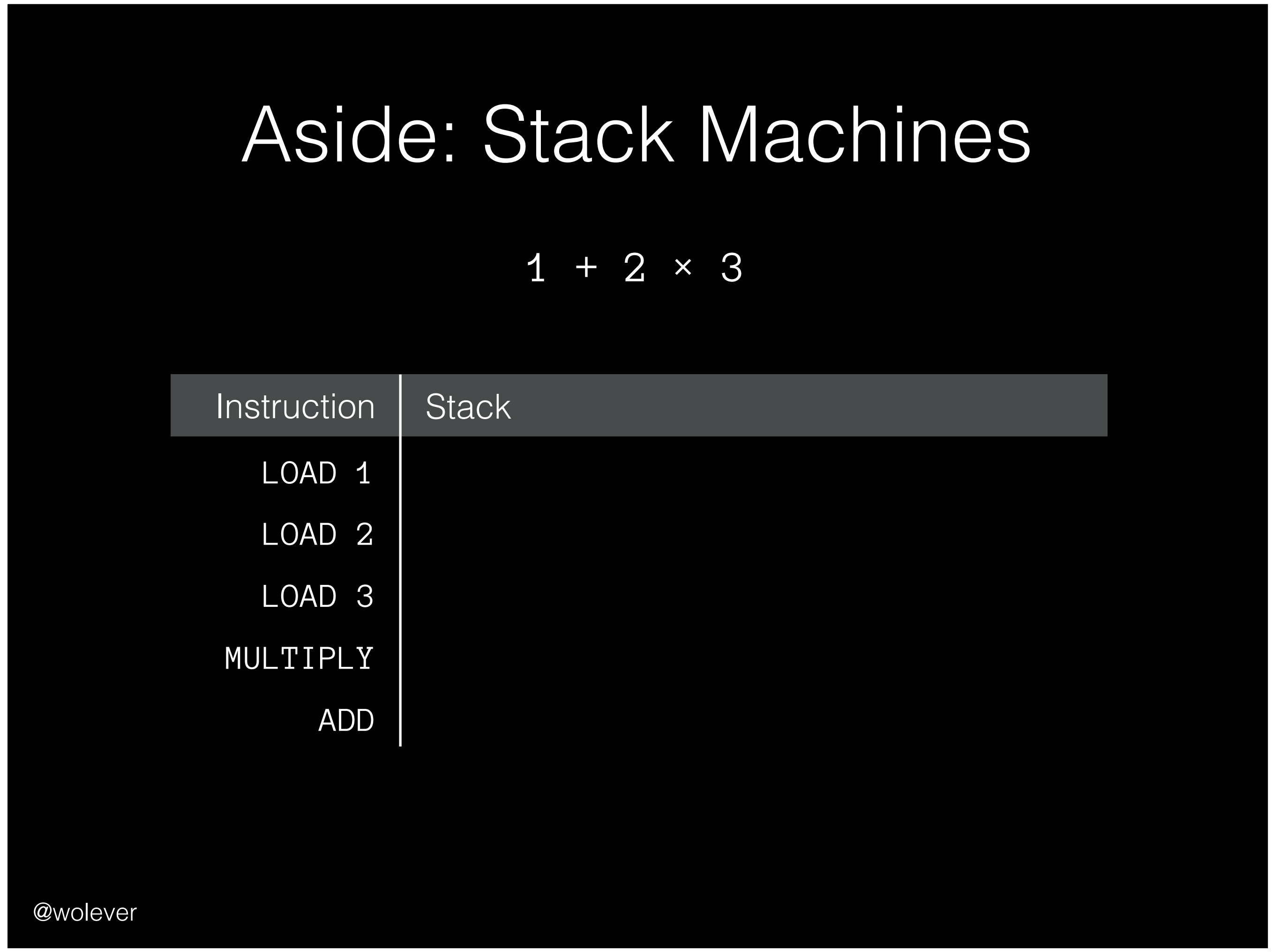
(this is in contrast with register machines, like the processor in your computer, where instructions can take multiple arguments, and pass values around in a bunch of different ways)

Aside: Stack Machines

1 + 2 × 3

@wolever

Consider evaluating the expression: 1 + 2 * 3:



It would be converted to these (fake) virtual machine instructions:

Aside: Stack Machines

`1 + 2 × 3`

Instruction	Stack
`LOAD 1`	`[1]`
`LOAD 2`	
`LOAD 3`	
`MULTIPLY`	
`ADD`	

@wolever

The one…

Aside: Stack Machines

`1 + 2 × 3`

Instruction	Stack
`LOAD 1`	`[1]`
`LOAD 2`	`[1, 2]`
`LOAD 3`	
`MULTIPLY`	
`ADD`	

@wolever

… two …

Aside: Stack Machines

1 + 2 × 3

Instruction	Stack
LOAD 1	[1]
LOAD 2	[1, 2]
LOAD 3	[1, 2, 3]
MULTIPLY	
ADD	

@wolever

… and three are pushed onto the stack, then …

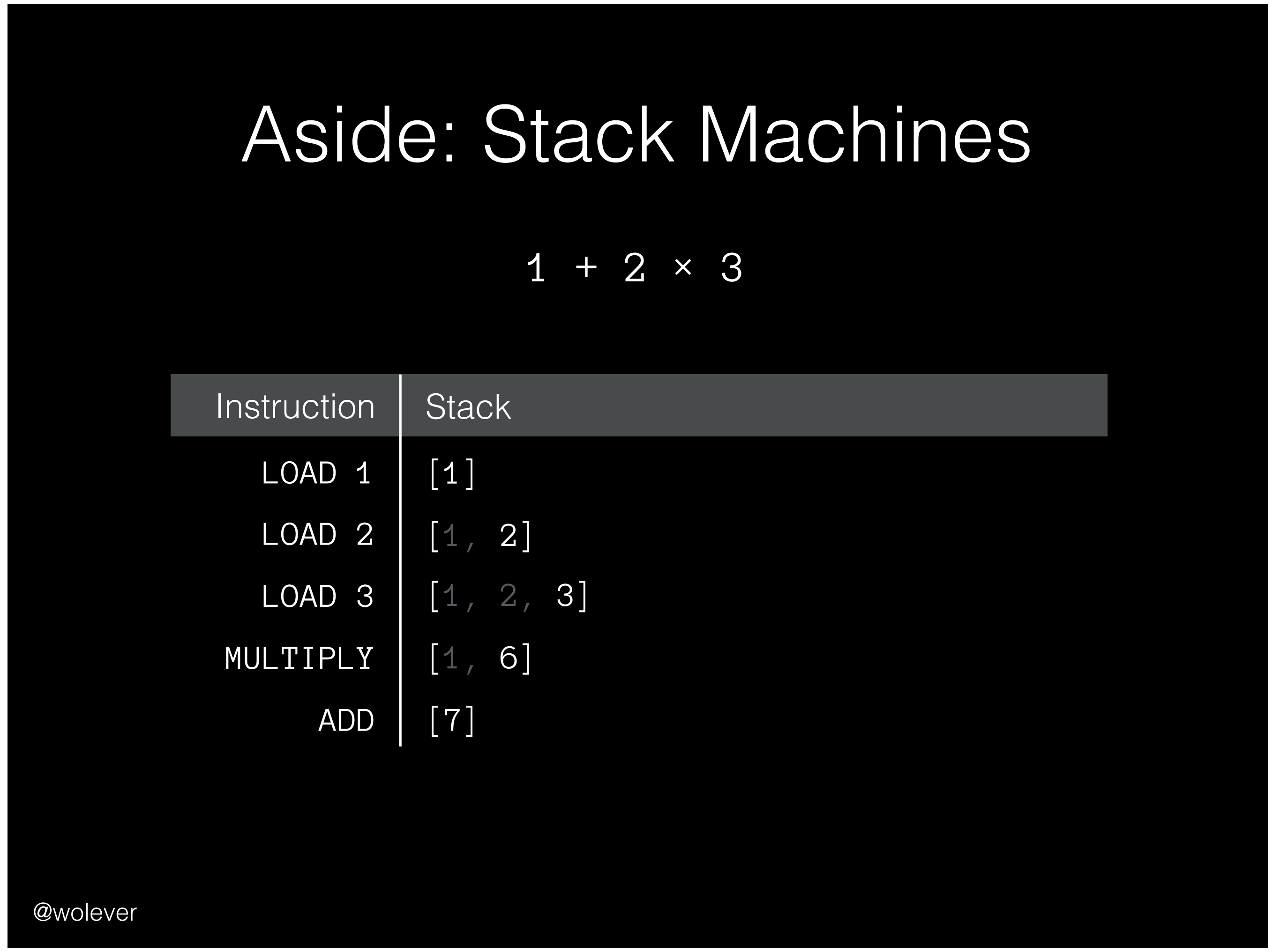
Aside: Stack Machines

`1 + 2 × 3`

Instruction	Stack
`LOAD 1`	`[1]`
`LOAD 2`	`[1, 2]`
`LOAD 3`	`[1, 2, 3]`
`MULTIPLY`	`[1, 6]`
`ADD`	

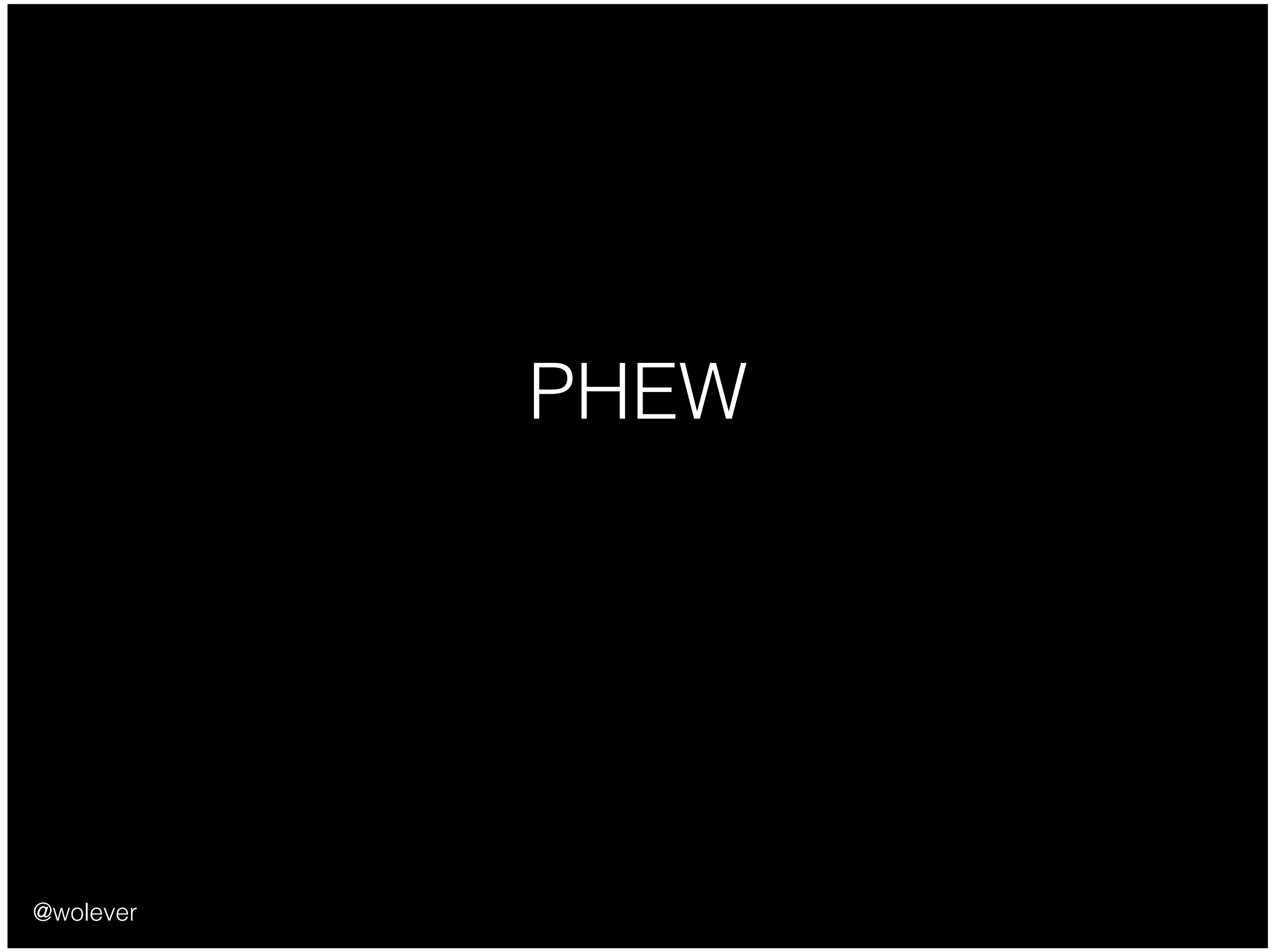
@wolever

… the multiply instruction pops the last two numbers – two and three – *off* the stack, multiplies them, and puts the result – six – onto the stack.



The ADD instruction is similar, popping six and one off the stack, adding them, and pushing the result – seven – back.

A stack machines are used because they’re very simple, very easy to implement, reason about, and optimize. In fact, in addition to Python, Java, PostScript, Etherium (a crypto currency), and Rubinious (a Ruby interpreter) are also implemented with Stack Machines.



PHEW! That was a lot.

Time to get back to the problem at hand:

```
In [1]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 30.9 ns per loop

In [2]: %timeit 'x' == 'x'
10000000 loops, best of 3: 31.3 ns per loop

In [3]: %timeit 'x' in ('x', )
10000000 loops, best of 3: 29.5 ns per loop

In [4]: %timeit 'x' == 'x'
10000000 loops, best of 3: 30.7 ns per loop
```

@wolever

If you remember, we’re trying to figure out why tuple membership is consistently a tiny bit faster than equality.

```
In [3]: import dis

In [4]: def in_():
   ....:     return "x" in ("x", )

In [5]: dis.disassemble(in_.func_code)
  2           0 LOAD_CONST               1 ('x')
              3 LOAD_CONST               2 (('x',))
              6 COMPARE_OP               6 (in)
              9 RETURN_VALUE
```

@wolever

To get started, we're going to disassemble each of the statements: first the tuple membership

```
In [6]: def eq():
   ...:     return "x" == "x"

In [7]: dis.disassemble(eq.func_code)
  2           0 LOAD_CONST               1 ('x')
              3 LOAD_CONST               1 ('x')
              6 COMPARE_OP               2 (==)
              9 RETURN_VALUE
```

@wolever

And second equality.

And by the way, if you were wondering, these numbers here are indexes into the functions co_constants tuple.

```
In [6]: def eq():
   ....:     return "x" == "x"

In [7]: dis.disassemble(eq.func_code)
  2           0 LOAD_CONST               1 ('x')
              3 LOAD_CONST               1 ('x')
              6 COMPARE_OP               2 (==)
              9 RETURN_VALUE
```

@wolever

And second equality.

And by the way, if you were wondering, these numbers here are indexes into the functions co_constants tuple.

```
In [6]: def eq():
   ....:     return "x" == "x"

In [7]: dis.disassemble(eq.func_code)
  2           0 LOAD_CONST               1 ('x')
              3 LOAD_CONST               1 ('x')
              6 COMPARE_OP               2 (==)
              9 RETURN_VALUE
```

See: eq.func_code.co_consts

@wolever

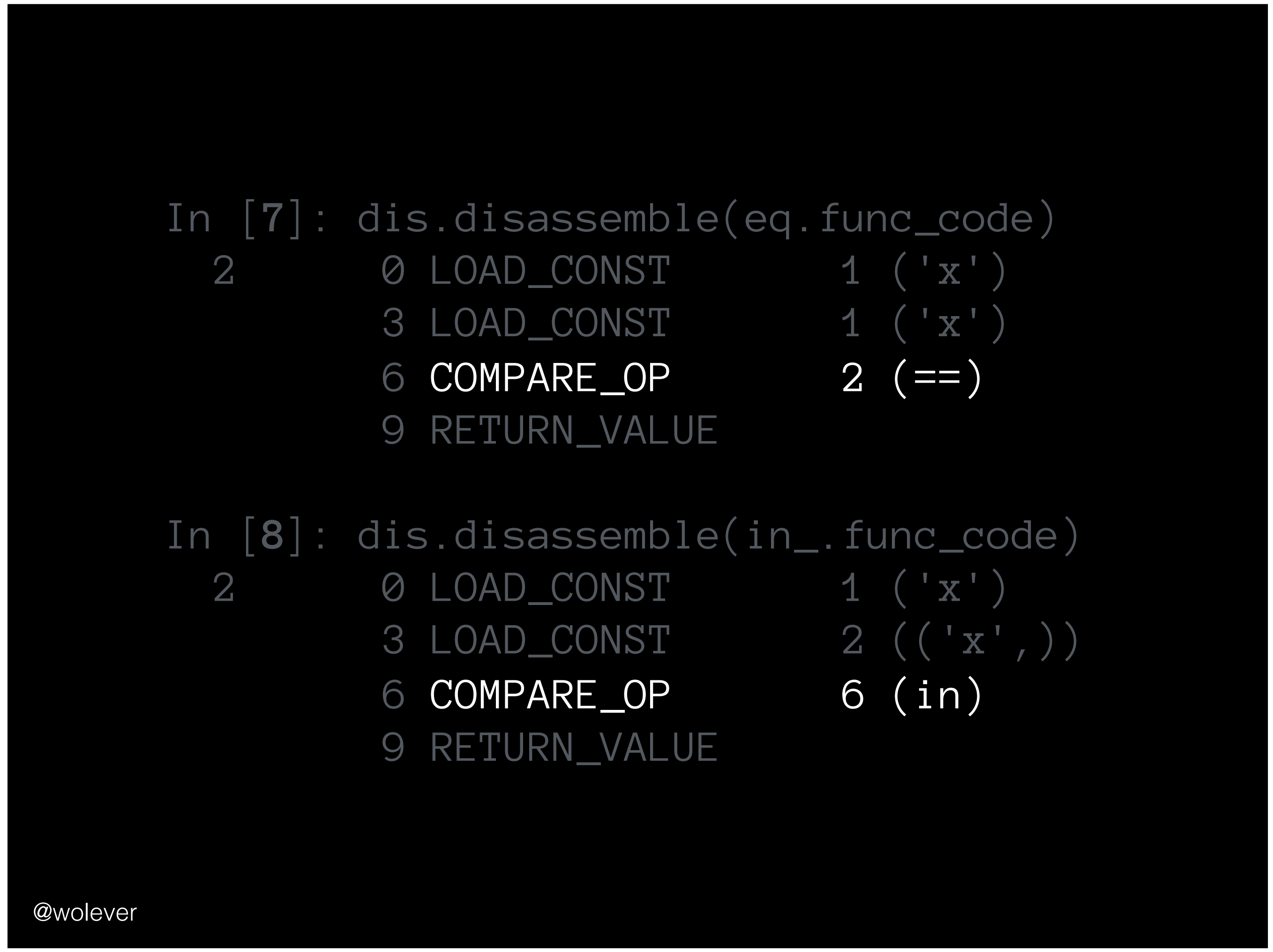
And second equality.

And by the way, if you were wondering, these numbers here are indexes into the functions co_constants tuple.

```
In [7]: dis.disassemble(eq.func_code)
  2           0 LOAD_CONST               1 ('x')
              3 LOAD_CONST               1 ('x')
              6 COMPARE_OP               2 (==)
              9 RETURN_VALUE

In [8]: dis.disassemble(in_.func_code)
  2           0 LOAD_CONST               1 ('x')
              3 LOAD_CONST               2 (('x',))
              6 COMPARE_OP               6 (in)
              9 RETURN_VALUE
```

@wolever

And comparing the two side-by-side, we can see that they’re virtually identical…



… except for the argument to COMPARE_OP.

What's going on past here?

To understand, we're going to dig into the source code of Python its self!

```
$ wget https://python.org/…/Python-2.7.12.tar.xz
$ tar xf Python-2.7.12.tar.xz
$ cd Python-2.7.12
$ ctags -R .
$ ack COMPARE_OP
Doc/library/dis.rst
668:.. opcode:: COMPARE_OP (opname)

Include/opcode.h
114:#define COMPARE_OP  107 /* Comparison operator */

Lib/compiler/pyassem.py
493:    def _convert_COMPARE_OP(self, arg):
…
```

@wolever

Fortunately for us, the Python source is very, very approachable.

(and again, I'm using 2.7, but 3.6 will be very similar)

We'll download a tarball, extract it, and search for that COMPARE_OP

There are a few things which come up…

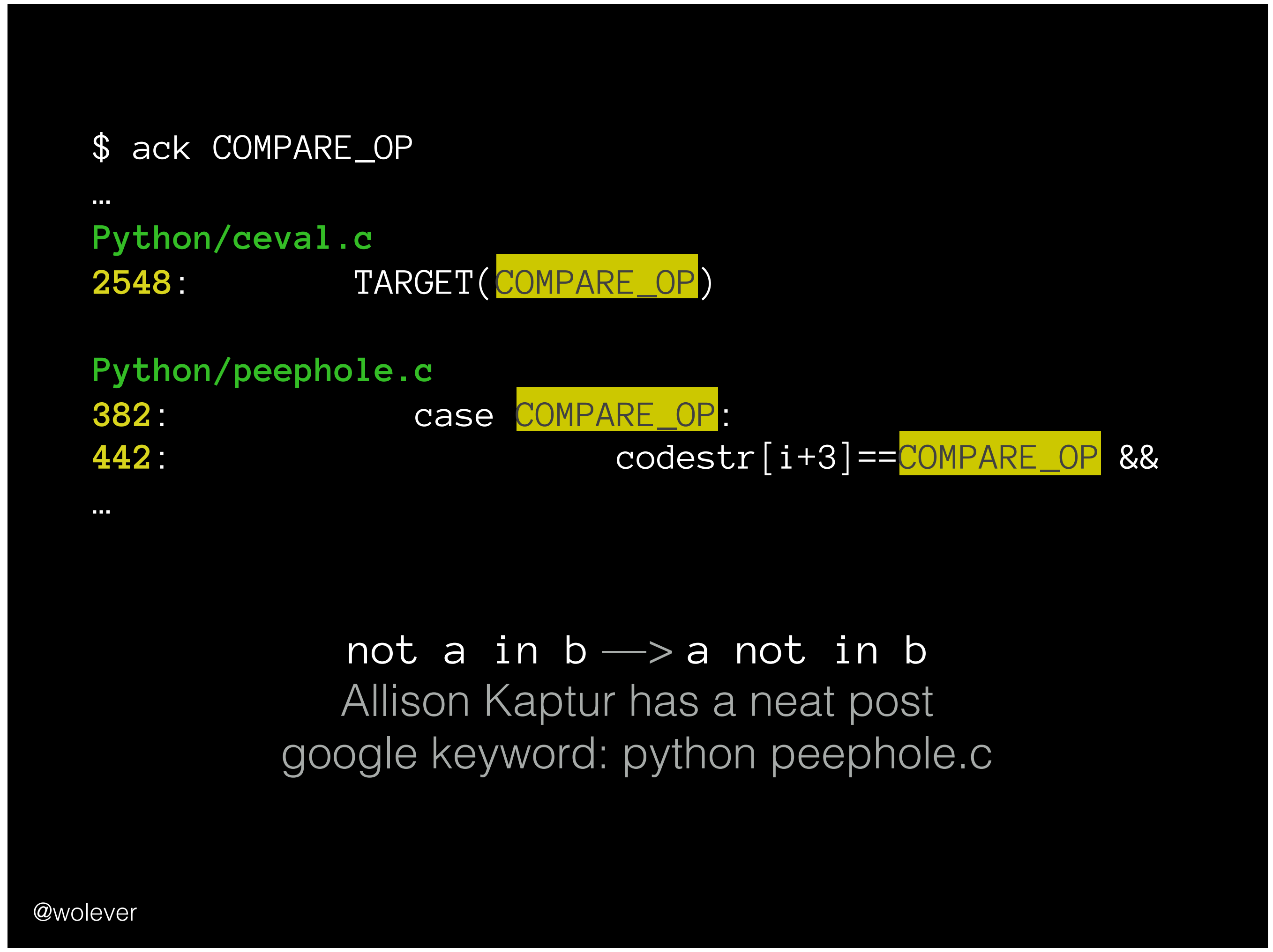
```
$ ack COMPARE_OP
…
Python/ceval.c
2548:        TARGET(COMPARE_OP)

Python/peephole.c
382:                case COMPARE_OP:
442:                            codestr[i+3]==COMPARE_OP &&
…
```

@wolever

Now, instead of going through all the different matches, I’m going to cheat a bit and just tell you:

peephole.c is very interesting - it performs in-place micro-optimizations on the byte code, things like transforming `not a in b` to `a not in b` (because `not in` is one operation, where `not a in b` is actually two). This is really cool - check out Allison’s blog post - but it’s not the file we want.

We want to start with ceval.c (no more comments)



Now, instead of going through all the different matches, I’m going to cheat a bit and just tell you:

peephole.c is very interesting - it performs in-place micro-optimizations on the byte code, things like transforming `not a in b` to `a not in b` (because `not in` is one operation, where `not a in b` is actually two). This is really cool - check out Allison’s blog post - but it’s not the file we want.

We want to start with ceval.c (no more comments)

```
In [7]: dis.disassemble(eq.func_code)
  2       0 LOAD_CONST          1 ('x')
          3 LOAD_CONST          1 ('x')
          6 COMPARE_OP          2 (==)
          9 RETURN_VALUE

In [8]: dis.disassemble(in_.func_code)
  2       0 LOAD_CONST          1 ('x')
          3 LOAD_CONST          2 (('x',))
          6 COMPARE_OP          6 (in)
          9 RETURN_VALUE
```

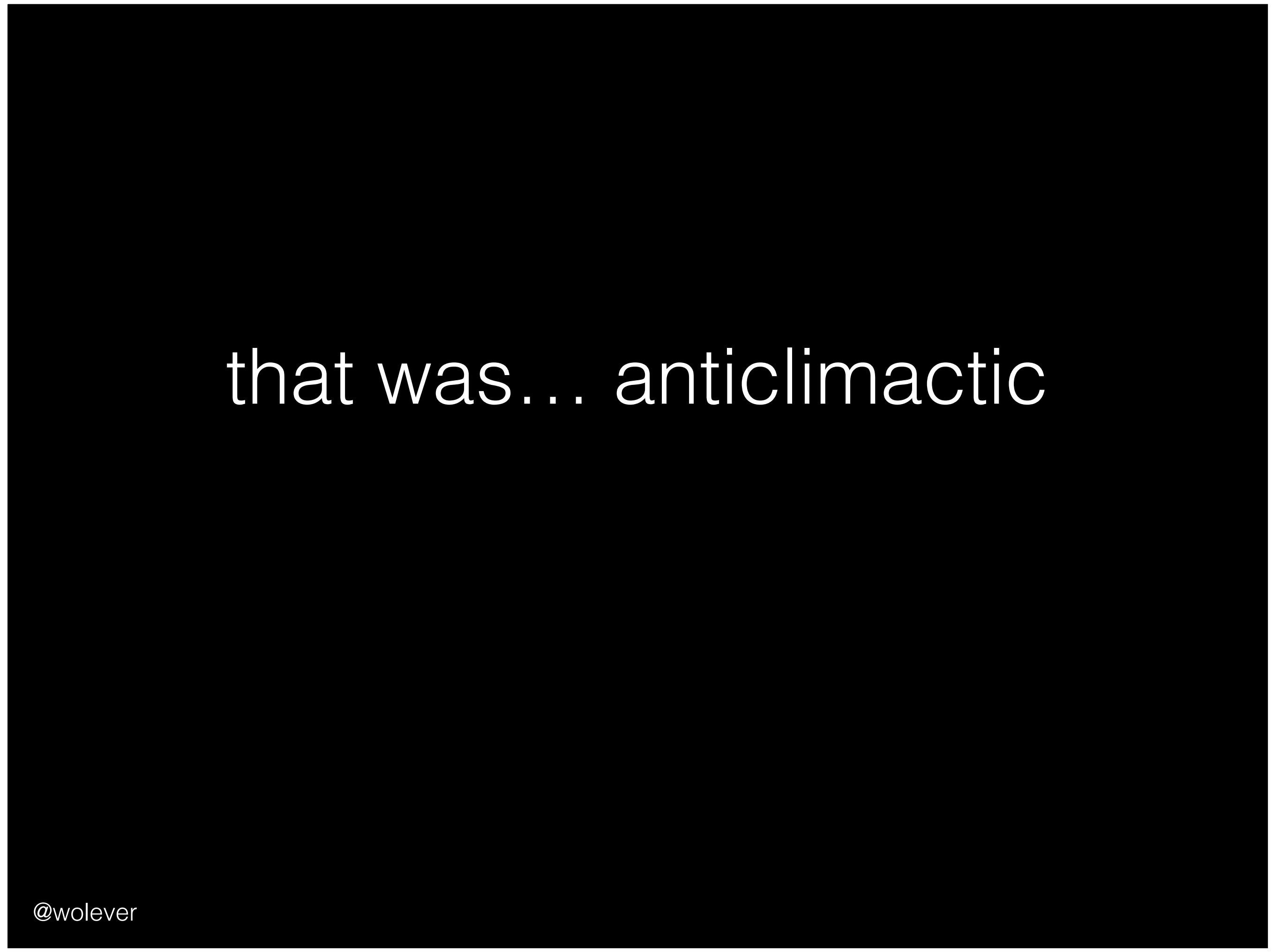
@wolever

Just before we pull that code up, a quick refresher: remember that both functions have virtually identical instructions, they only vary in the argument to COMPARE_OP: == VS in.

Now, to the code!

ceval.c

@wolever



I'm sorry if that wasn't nearly as exciting as you'd hoped for.

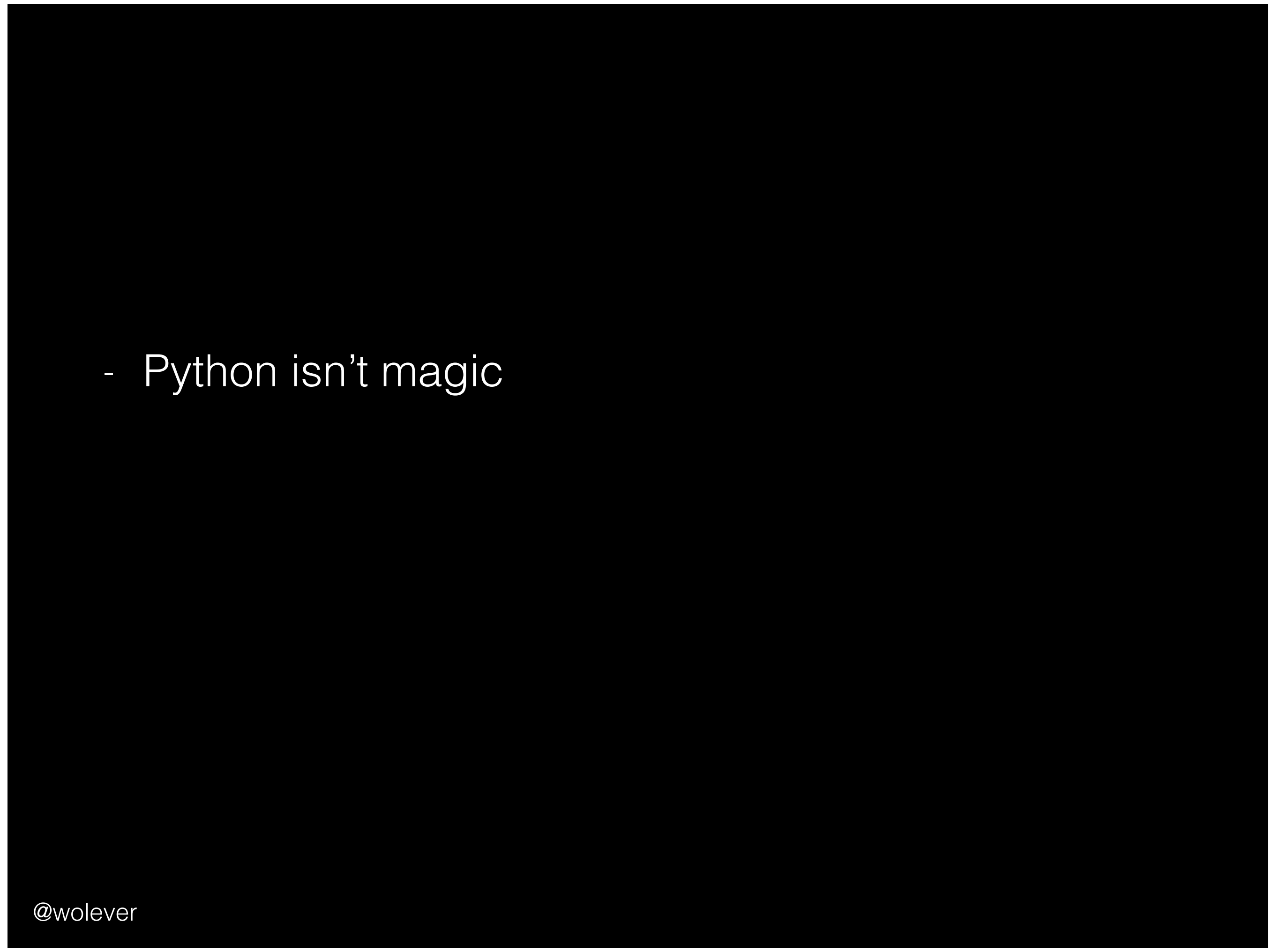
Programming rarely is.

But hopefully you have learned…

Python isn’t magic.

It feels that way sometimes, but now you know how to take a peek under the hood and see what’s going on behind the scenes.

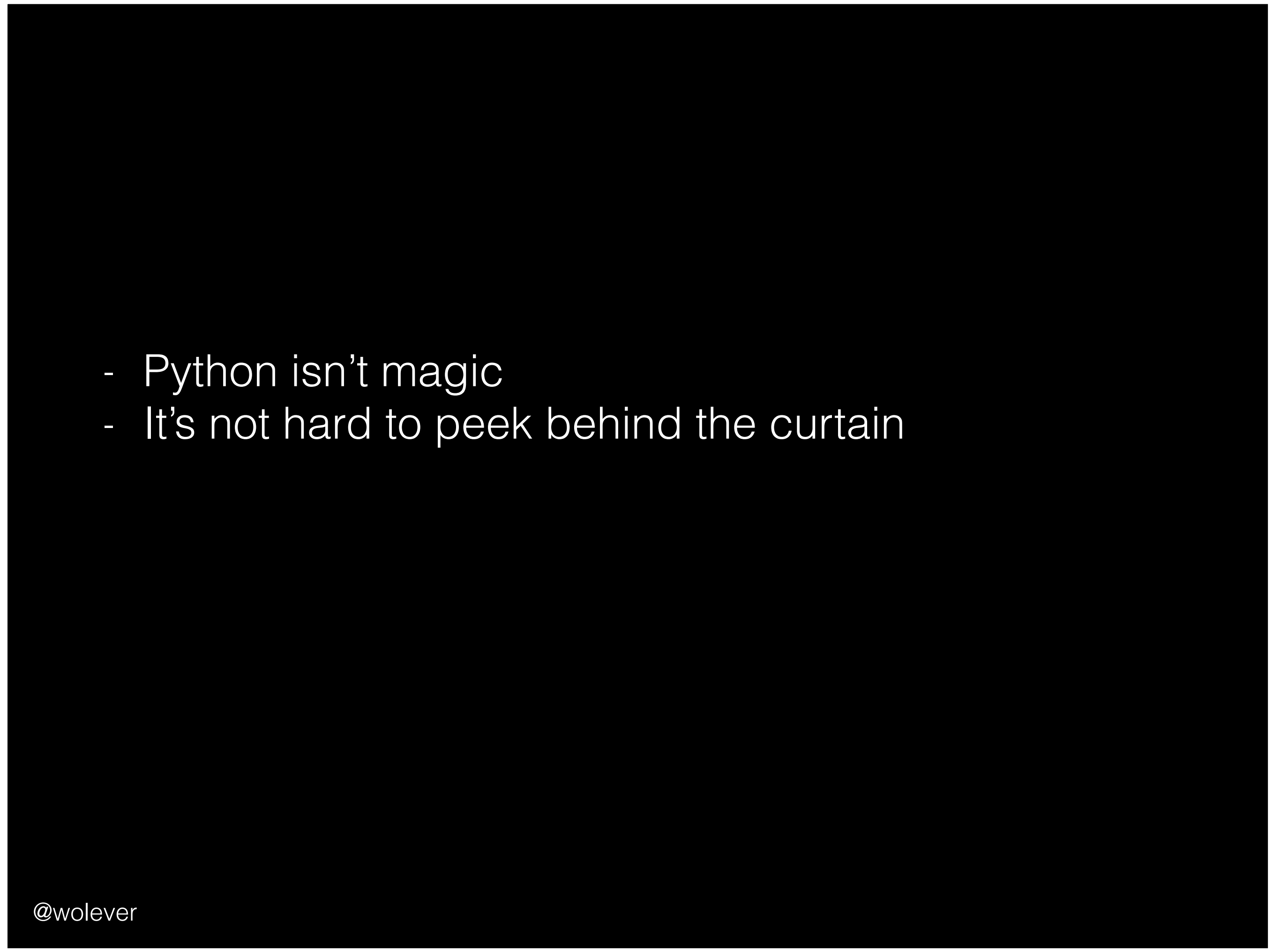
And hopefully you can see how this same technique can be used to solve more common problems: why your Django model isn’t saving, why Beautiful Soup isn’t matching a tag, or why urllib3 isn’t pooling the way you expect. See the "bonus content" slides at the end of this deck.



Python isn’t magic.

It feels that way sometimes, but now you know how to take a peek under the hood and see what’s going on behind the scenes.

And hopefully you can see how this same technique can be used to solve more common problems: why your Django model isn’t saving, why Beautiful Soup isn’t matching a tag, or why urllib3 isn’t pooling the way you expect. See the "bonus content" slides at the end of this deck.



Python isn't magic.

It feels that way sometimes, but now you know how to take a peek under the hood and see what's going on behind the scenes.

And hopefully you can see how this same technique can be used to solve more common problems: why your Django model isn't saving, why Beautiful Soup isn't matching a tag, or why urllib3 isn't pooling the way you expect. See the "bonus content" slides at the end of this deck.

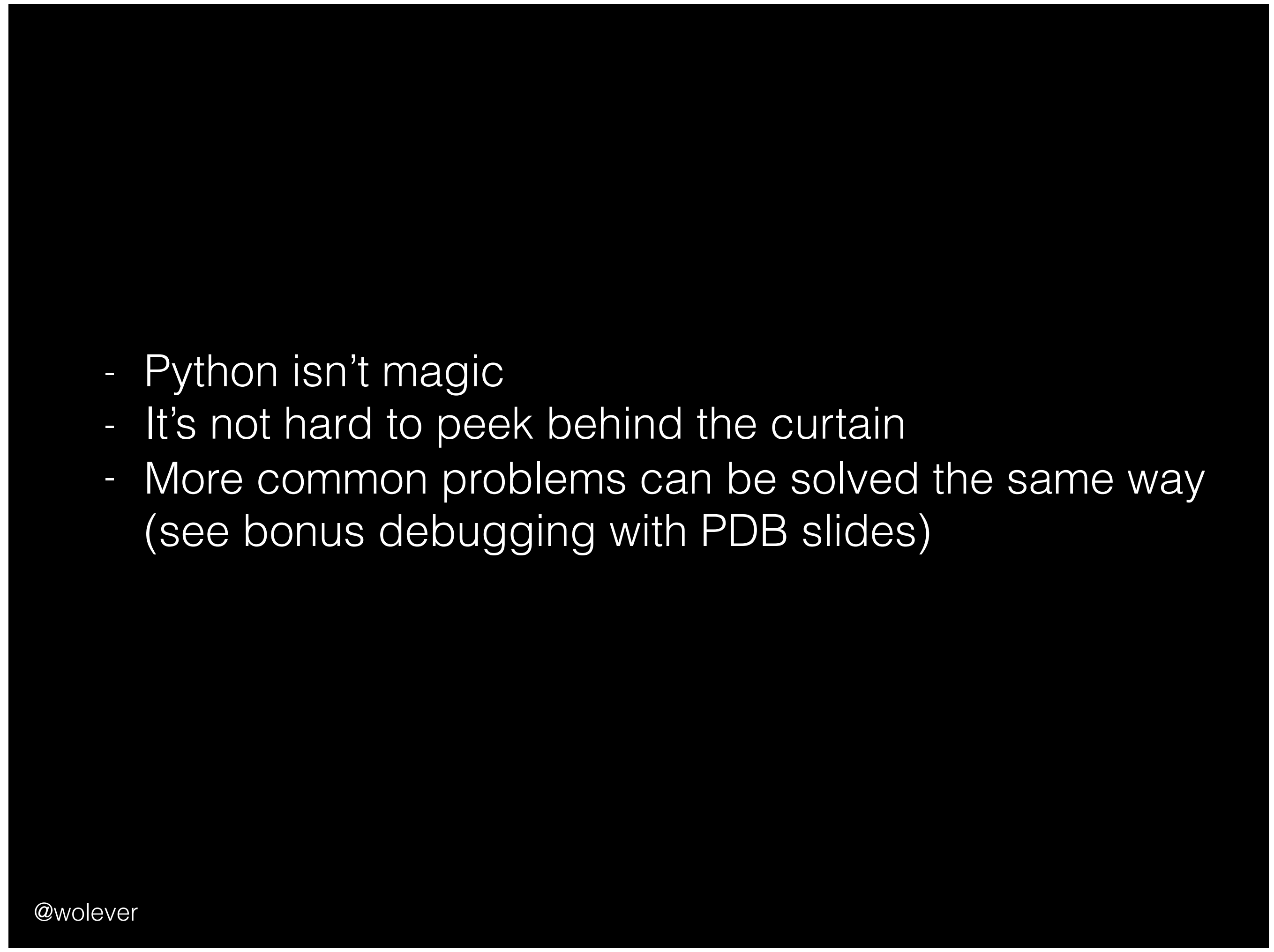
- Python isn’t magic
- It’s not hard to peek behind the curtain
- More common problems can be solved the same way (see bonus debugging with PDB slides)

@wolever

Python isn’t magic.

It feels that way sometimes, but now you know how to take a peek under the hood and see what’s going on behind the scenes.

And hopefully you can see how this same technique can be used to solve more common problems: why your Django model isn’t saving, why Beautiful Soup isn’t matching a tag, or why urllib3 isn’t pooling the way you expect. See the "bonus content" slides at the end of this deck.



Python isn’t magic.

It feels that way sometimes, but now you know how to take a peek under the hood and see what’s going on behind the scenes.

And hopefully you can see how this same technique can be used to solve more common problems: why your Django model isn’t saving, why Beautiful Soup isn’t matching a tag, or why urllib3 isn’t pooling the way you expect. See the "bonus content" slides at the end of this deck.

```
>>> nan = float("nan")
>>> nan is nan
True
>>> nan == nan
False
>>> nan in (nan, )
True
```

@wolever

Oh, and remember this code from the beginning?

Now you can see what’s going on: even though nan isn’t equal to nan, the tuple membership test ignores that and just checks identity.

And was this interesting? Want to try something for yourself?

Homework

You're seen methods added dynamically to objects before:

```
>>> p = Person()
>>> p.speak()
'Hello!'
>>> p.speak = lambda: "Bonjour!"
>>> p.speak()
'Bonjour!'
```

@wolever

Here's a bit of homework.

You know that you can dynamically add methods to objects…

Homework

Figure out why the second len(o) returns 42 instead of 17:

```
>>> class MyObject(object):
...  def __len__(self):
...    return 42
...
>>> o = MyObject()
>>> len(o)
42
>>> o.__len__ = lambda: 17
>>> len(o)
42
```

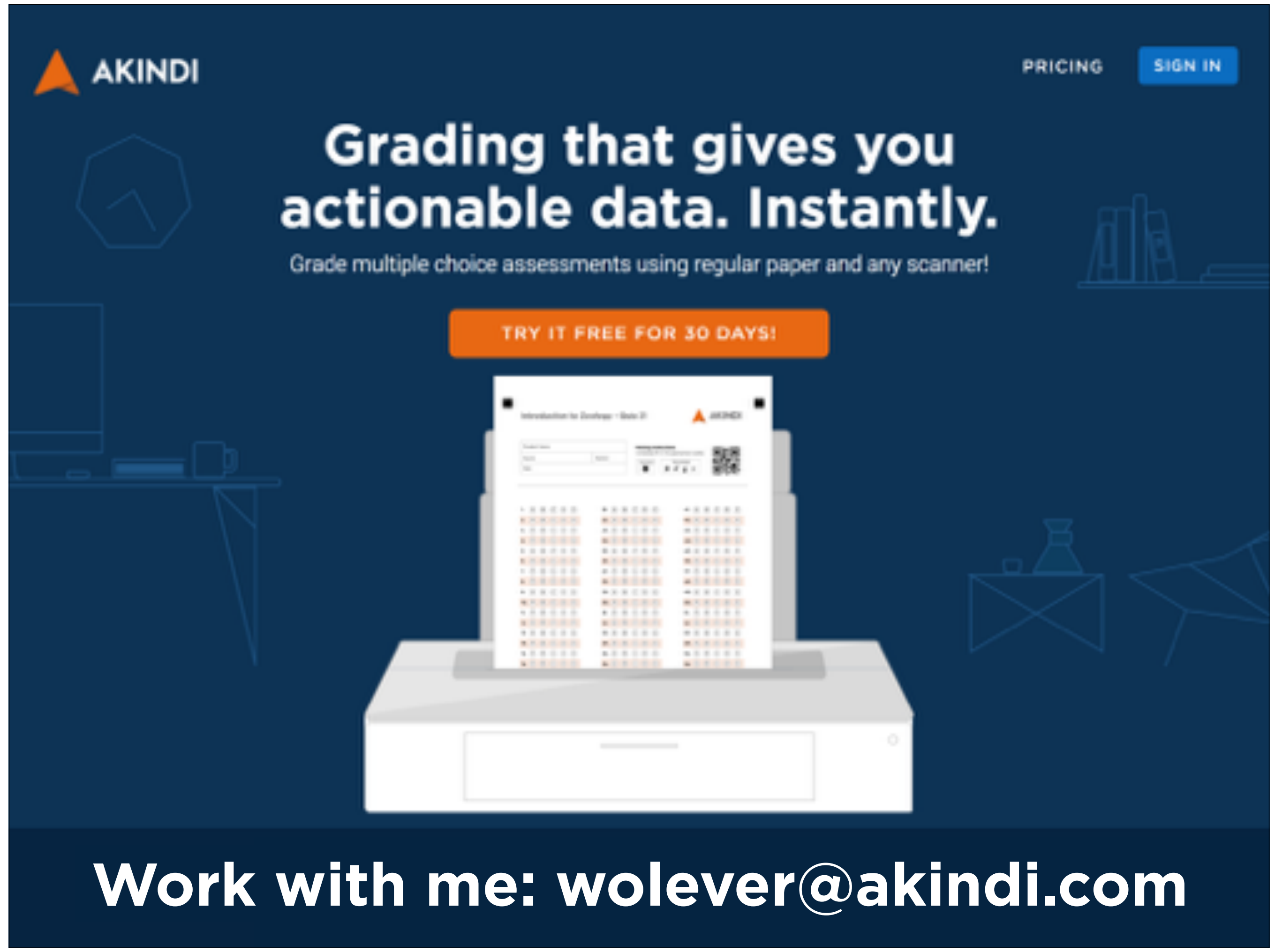
@wolever

… but that doesn't work with __len__.

You can see that, even though o.__len__() has been overriden to return 17, calling len still shows 42.

Without googling the answer, see if you can figure it out for yourself.

Now, two quick things I'd like to mention while I've got the stage!



First, I'm hiring! My company, Akindi, builds a product that lets teachers print Scantron-style bubble sheets from any printer, scan them from any scanner, and get all the results online. We are - and this isn't bragging, it's an objective fact - the best on the market at what we do… and that's mostly because we're the only product in our market built after the year 2000.

So if you're in Toronto (or interested in moving there) and you'd like to write software that makes teacher's lives less terrible, I'd love to chat with you!

thanks! see you in at 2:00 in B112!

David Wolever
@wolever

Second, I love knots. They're great. And I'm going to be holding a knot open space at 2:00 in B112.

I'll be geeking about knots, showing you how to tie some of my favourite, and hopefully learning some new ones.

Even if you've never tied anything but your shoes, c'mon out and I'll show you the perfect shoelace knot that will never, ever come undone… and looks real pretty to boot.

I've got a bunch of rope to give away, and it should be a good time.

Thank you all so much! If you've got any questions or comments, I'd love to chat or tweet with you afterwards!

Links

- https://twitter.com/wolever
- The StackOverflow question: http://stackoverflow.com/questions/28885132/why-is-x-in-x-faster-than-x-x
- Alison Kaptur's post on the peephole optimizer: http://akaptur.com/blog/2014/08/02/the-cpython-peephole-optimizer-and-you/
- ctags: http://ctags.sourceforge.net
- Computed gotos:
  https://bugs.python.org/issue4753
  http://eli.thegreenplace.net/2012/07/12/computed-goto-for-efficient-dispatch-tables
- String interning: http://guilload.com/python-string-interning/

@wolever

Links will be in the slides

(bonus content: pdb)

Learning an interactive debugger will have a ***profound*** impact on your ability to understand new code

(bonus content: pdb)

Instead of just reading through code and guessing what's happening, a debugger lets you step through and see exactly what's happening.

try using a debugger!

Instead of just reading through code and guessing what's happening, a debugger lets you step through and see exactly what's happening.

try using a debugger!

- Start right now: put this line in your code somewhere:
  `import pdb; pdb.set_trace()`
- Options: `pdb` / `pdb++` / `bpdb` / `ipdb` / `nose.tools.set_trace`
- `%pdb` in IPython
- WinPDB for remote interactive debugging (cross platform)
- `celery.contrib.rdb` for celery tasks
- The IDE you're already using
- A shortcut key in Vim:

```
map <F8>Ofrom nose.tools import set_trace; \
    set_trace() # BREAK<esc>
```

try using a debugger!

Bonus points: debug into library code.

Put a a debug statement into
`$VIRTUAL_ENV/lib/python2.7/site-packages/django/db/models/base.py`

the pdb commands you need

```
(pdb) list   # show source code
(pdb) next   # execute next line
(pdb) step   # enter the next function
(pdb) return # return from function
(pdb) print  # print a value
(pdb) bt     # print stack ("back") trace
(pdb) up     # move up one stack frame
(pdb) down   # move down one stack frame
```