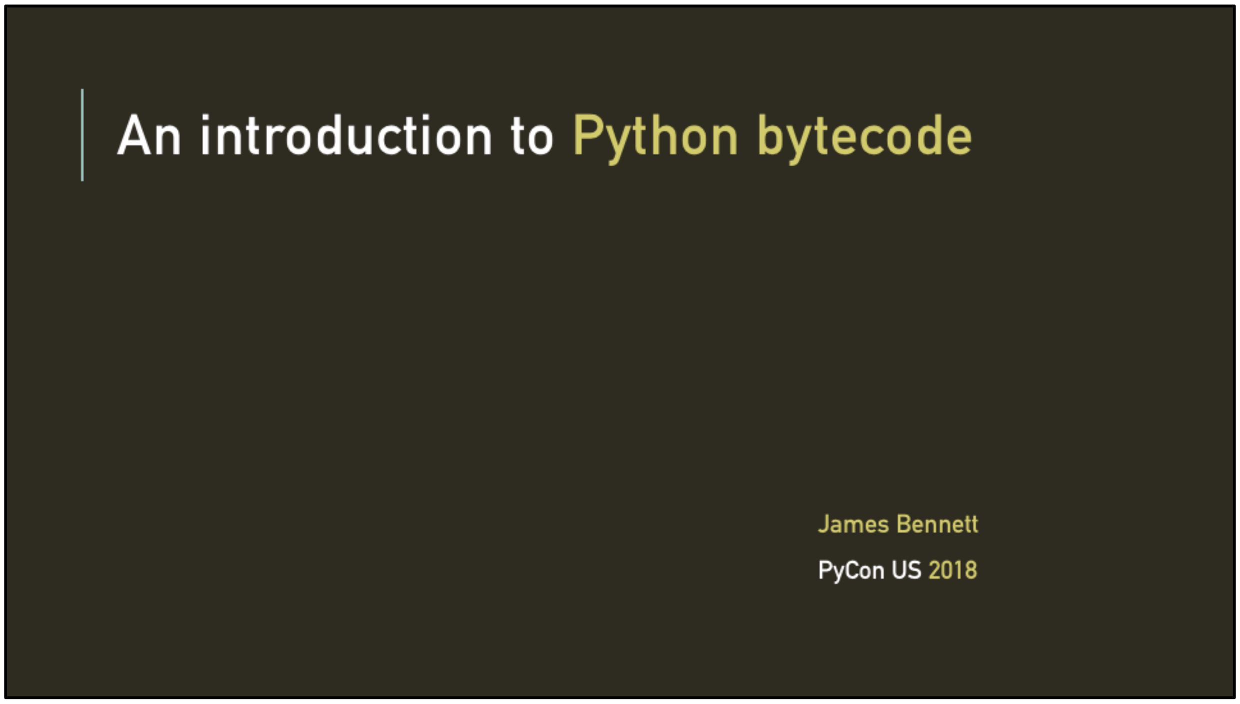
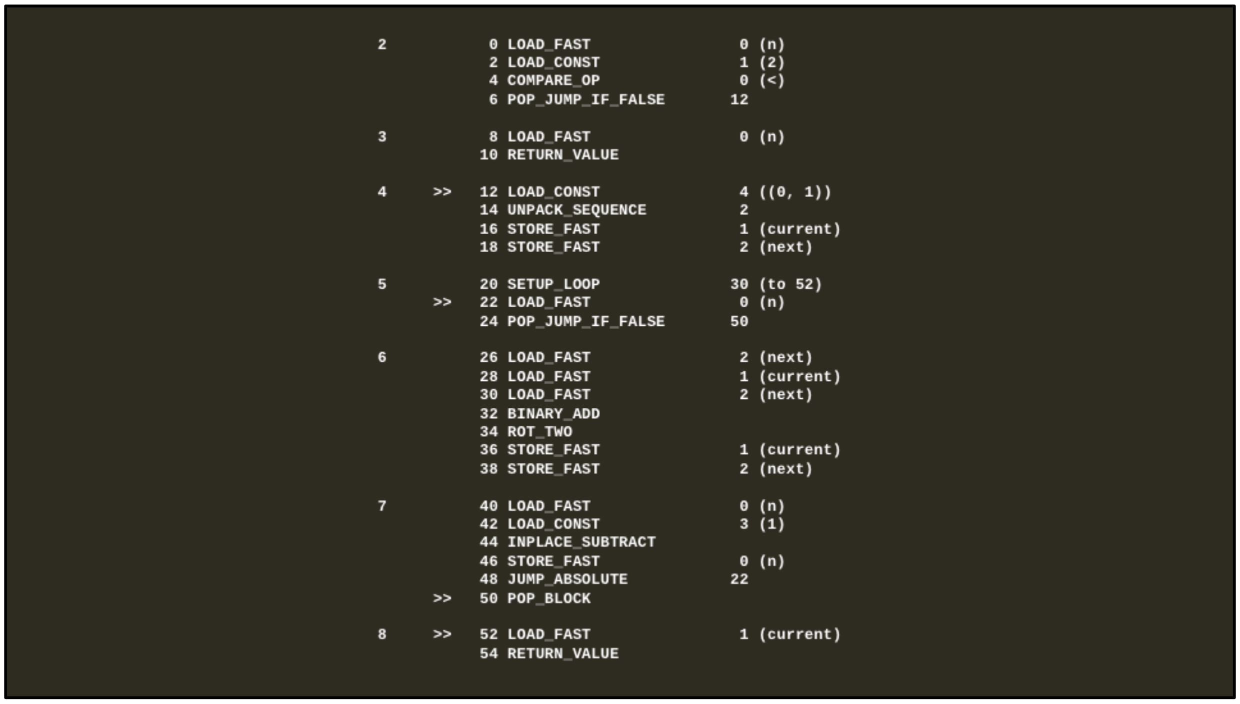


An introduction to Python bytecode

James Bennett

PyCon US 2018

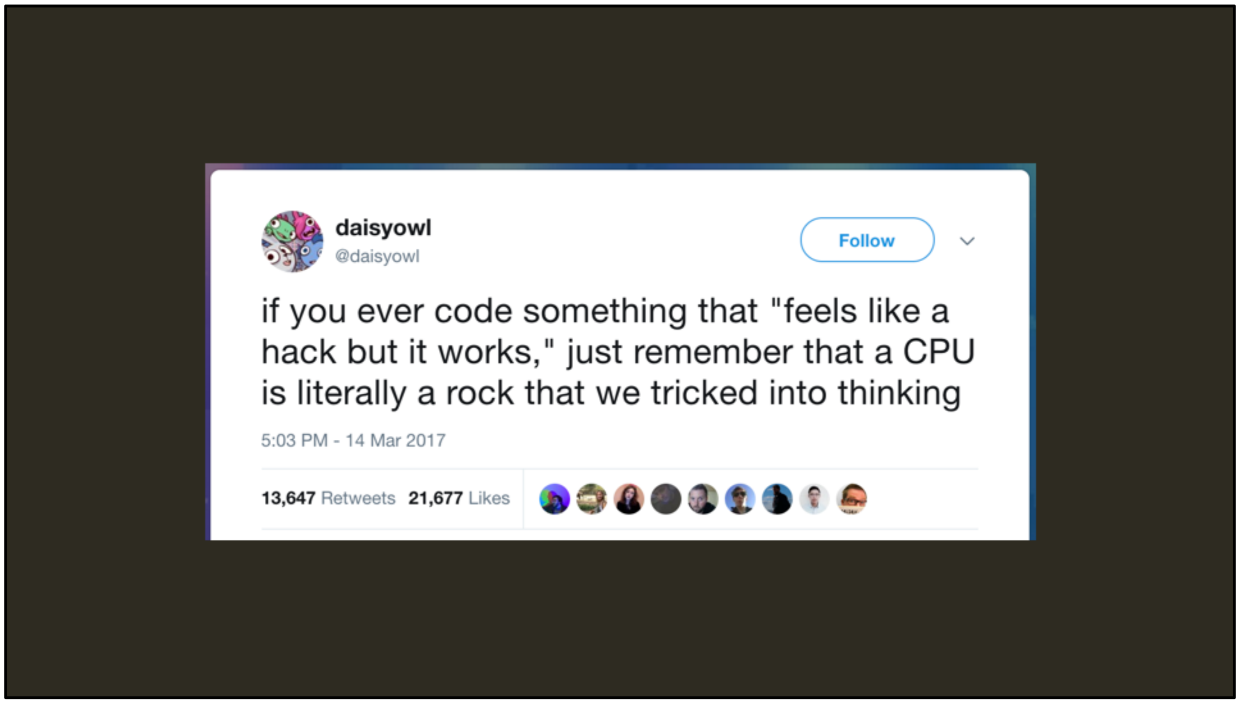
We're all here because we love Python. And we all know what's great about Python: code is read more often than it's written, so Guido built a language optimized to be easy to read.



```
2           0 LOAD_FAST                0 (n)
            2 LOAD_CONST               1 (2)
            4 COMPARE_OP               0 (<)
            6 POP_JUMP_IF_FALSE       12

3           8 LOAD_FAST                0 (n)
           10 RETURN_VALUE

4     >>   12 LOAD_CONST               4 ((0, 1))
           14 UNPACK_SEQUENCE          2
           16 STORE_FAST               1 (current)
           18 STORE_FAST               2 (next)

5          20 SETUP_LOOP              30 (to 52)
      >>   22 LOAD_FAST                0 (n)
           24 POP_JUMP_IF_FALSE       50

6          26 LOAD_FAST                2 (next)
           28 LOAD_FAST                1 (current)
           30 LOAD_FAST                2 (next)
           32 BINARY_ADD
           34 ROT_TWO
           36 STORE_FAST               1 (current)
           38 STORE_FAST               2 (next)

7          40 LOAD_FAST                0 (n)
           42 LOAD_CONST               3 (1)
           44 INPLACE_SUBTRACT
           46 STORE_FAST               0 (n)
           48 JUMP_ABSOLUTE           22
      >>   50 POP_BLOCK

8     >>   52 LOAD_FAST                1 (current)
           54 RETURN_VALUE
```

Like this. Perfectly clear, readable, understandable Python!

OK, so this is cheating a bit, but this really *is* Python. At least, it’s the way Python works. To understand that, though, we need to understand a bit about how computers work.



daisyowl
@daisyowl

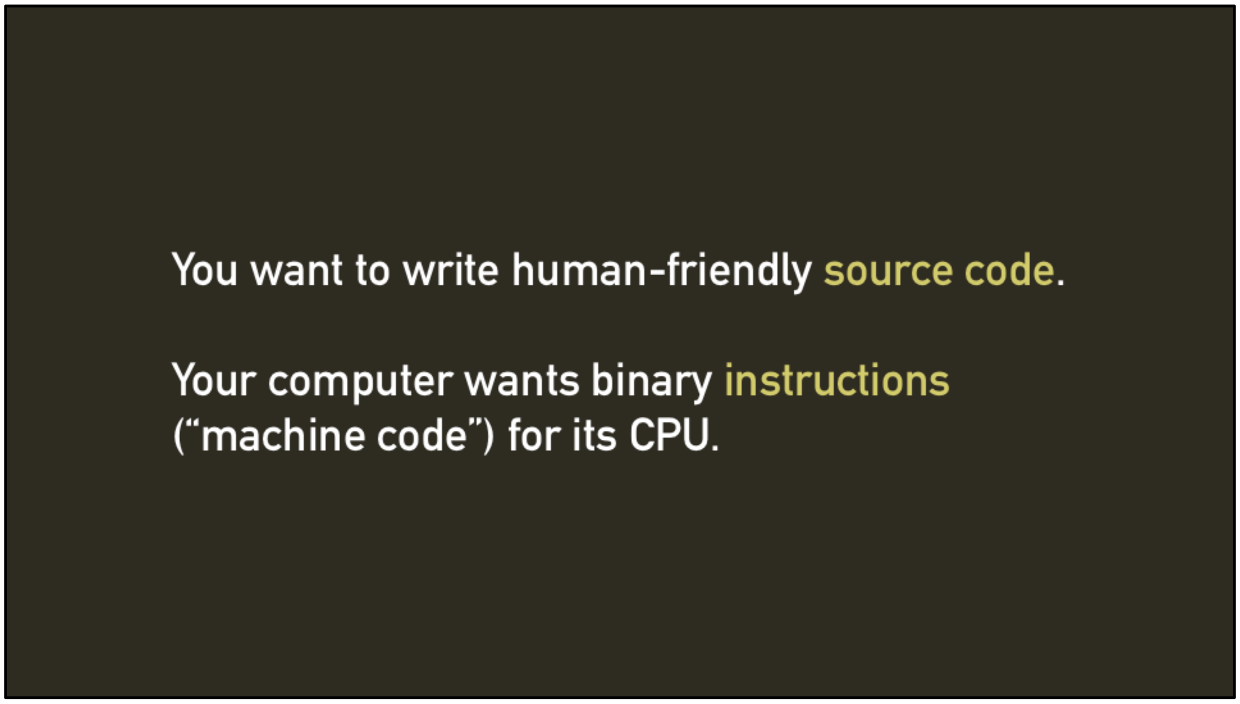
Follow

if you ever code something that "feels like a hack but it works," just remember that a CPU is literally a rock that we tricked into thinking

5:03 PM - 14 Mar 2017

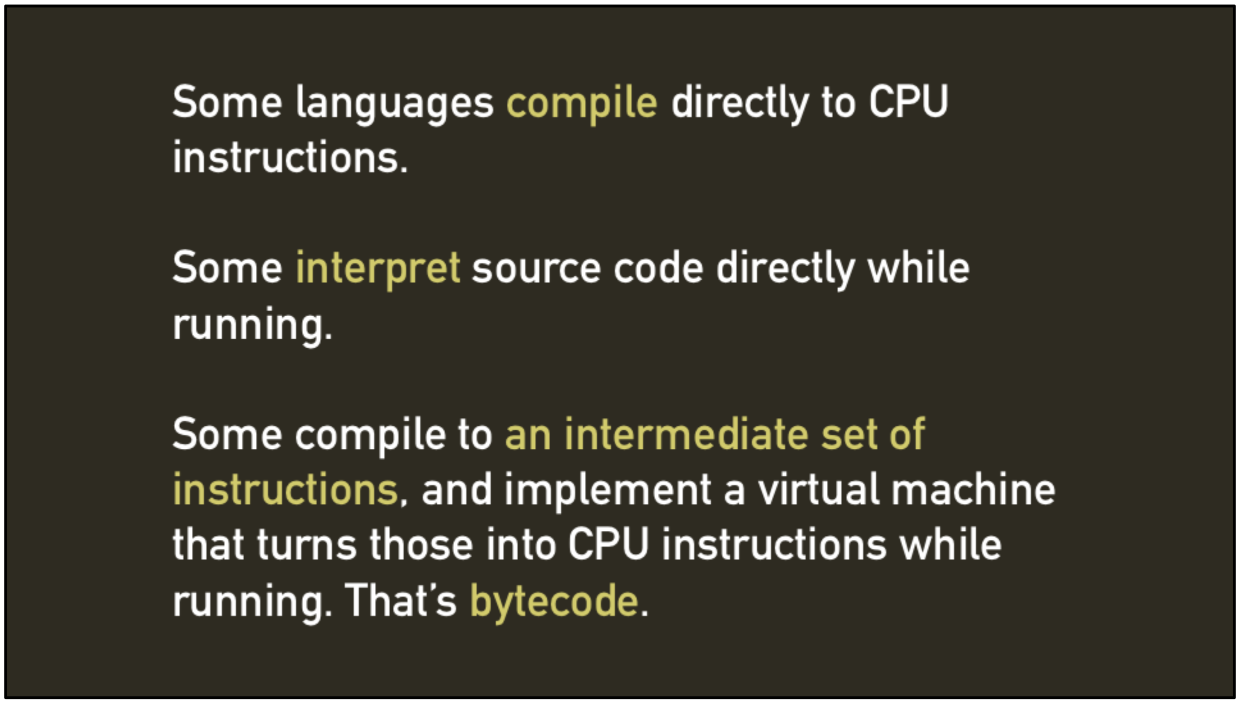
13,647 Retweets 21,677 Likes

I love this tweet, because it is true and therefore beautiful.


You want to write human-friendly source code.

Your computer wants binary instructions ("machine code") for its CPU.

Computers work because a CPU is a bunch of circuits, inscribed on a wafer of silicon, set up so that if you send certain patterns of electricity in you get certain other patterns of electricity out. We call those "instructions" for the CPU, and assign meanings to the patterns, like "add these two numbers". Programming languages need a way to turn the source code you write into instructions for the CPU.

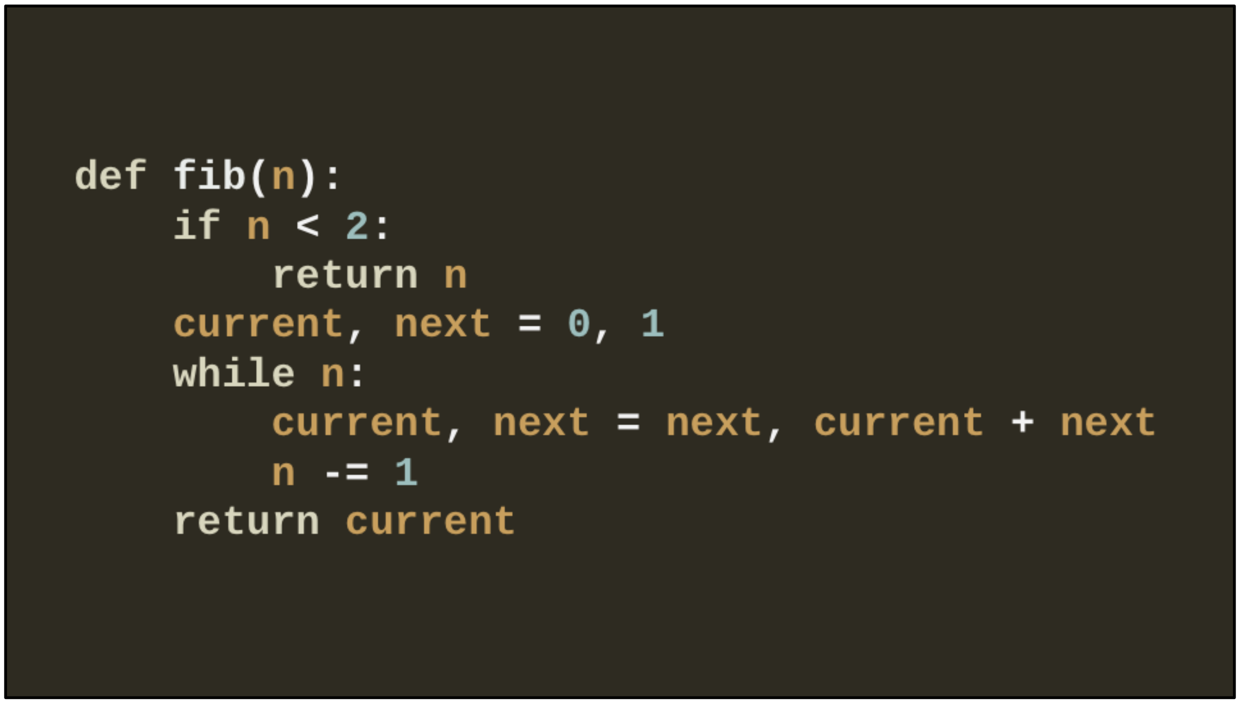

Some languages compile directly to CPU instructions.

Some interpret source code directly while running.

Some compile to an intermediate set of instructions, and implement a virtual machine that turns those into CPU instructions while running. That's bytecode.

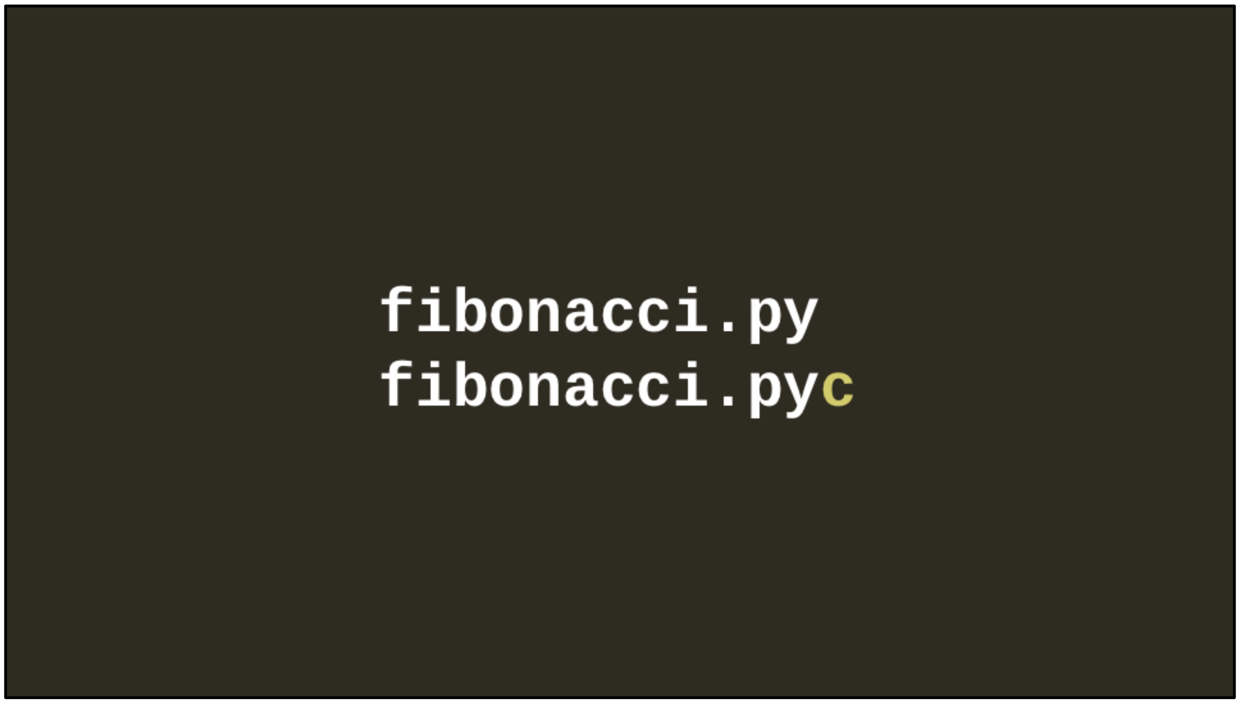
Some do this as a separate step from running the program; we call those compiled languages. Others do it while the program is running, and we call those interpreted languages.

But some languages actually do something in between; they compile to instructions for a CPU that doesn't physically exist, and then implement that CPU in software, translating between the instructions for the virtual machine and instructions for the physical wafer of silicon sitting inside the computer. We call that intermediate set of instructions… bytecode! Java and all the JVM languages do this. All the .NET languages, like C#, do this. And Python does this.



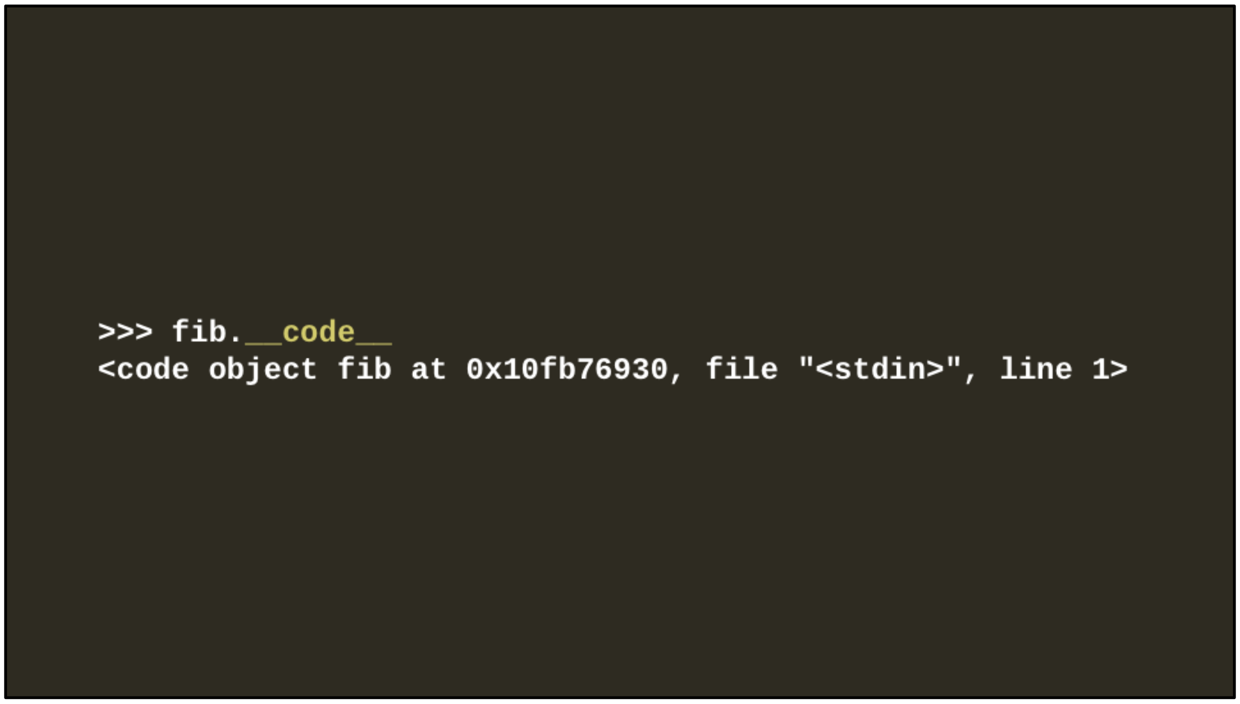
```python
def fib(n):
    if n < 2:
        return n
    current, next = 0, 1
    while n:
        current, next = next, current + next
        n -= 1
    return current
```

Here’s a function that calculates Fibonacci numbers. How does Python execute this?



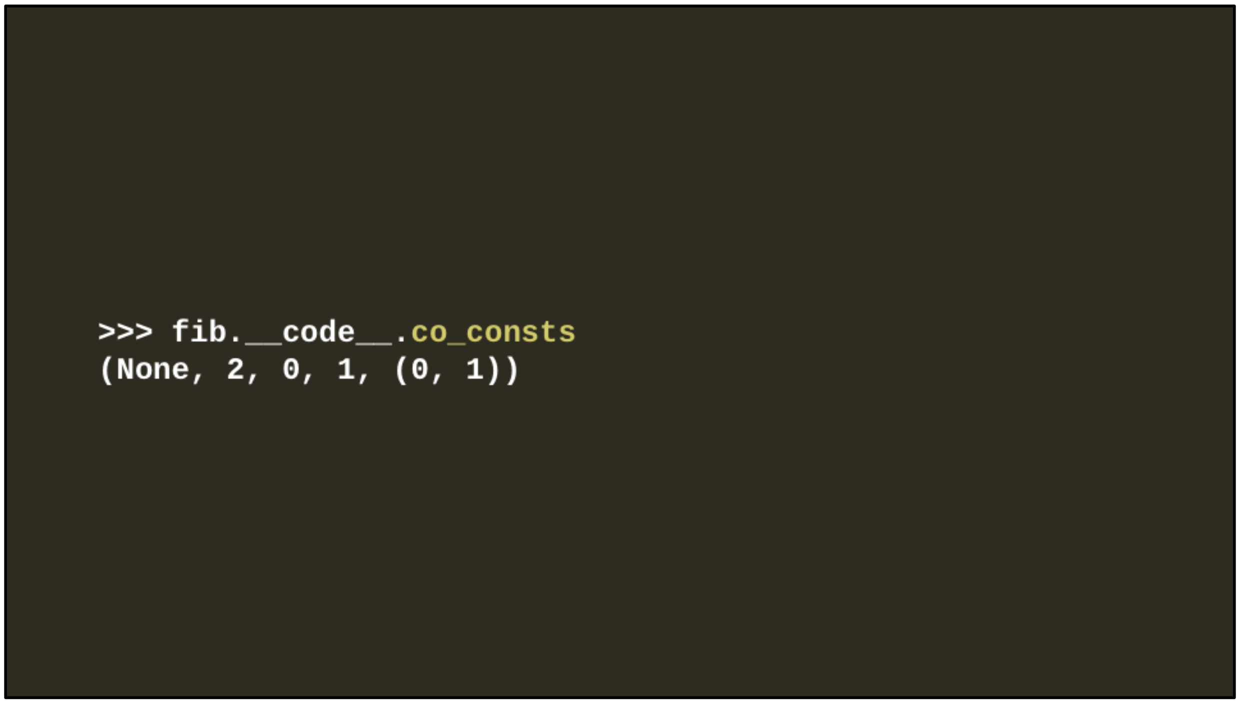
```
fibonacci.py
fibonacci.pyc
```

When you run a Python program, it gets compiled to bytecode. When you import a Python module, it also gets compiled to bytecode, and Python stores the result in a file with a '.pyc' extension. In Python 2 you used to see them alongside the source files, but in Python 3 they live in a __pycache__ directory.



```
>>> fib.__code__
<code object fib at 0x10fb76930, file "<stdin>", line 1>
```

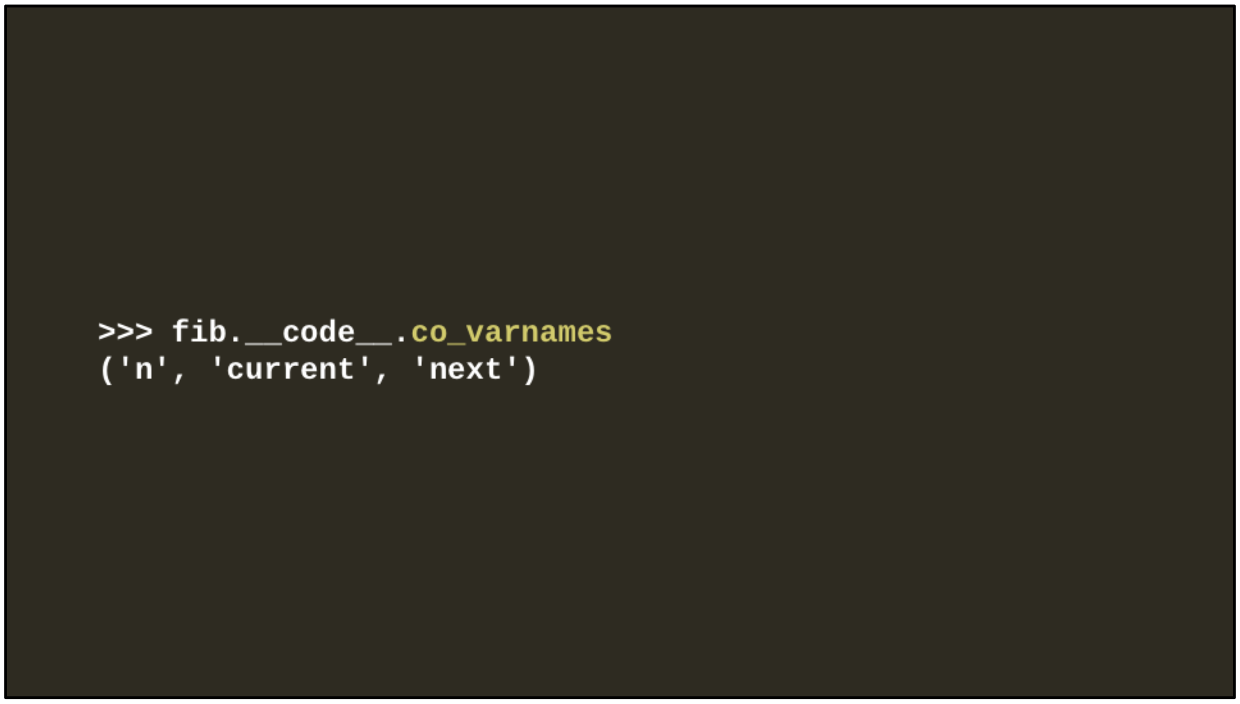
Check out this attribute on the Fibonacci function: it’s a Python code object. It contains everything Python needs to be able to execute the function. And by poking at its attributes, we can see how Python executes the function.



```
>>> fib.__code__.co_consts
(None, 2, 0, 1, (0, 1))
```

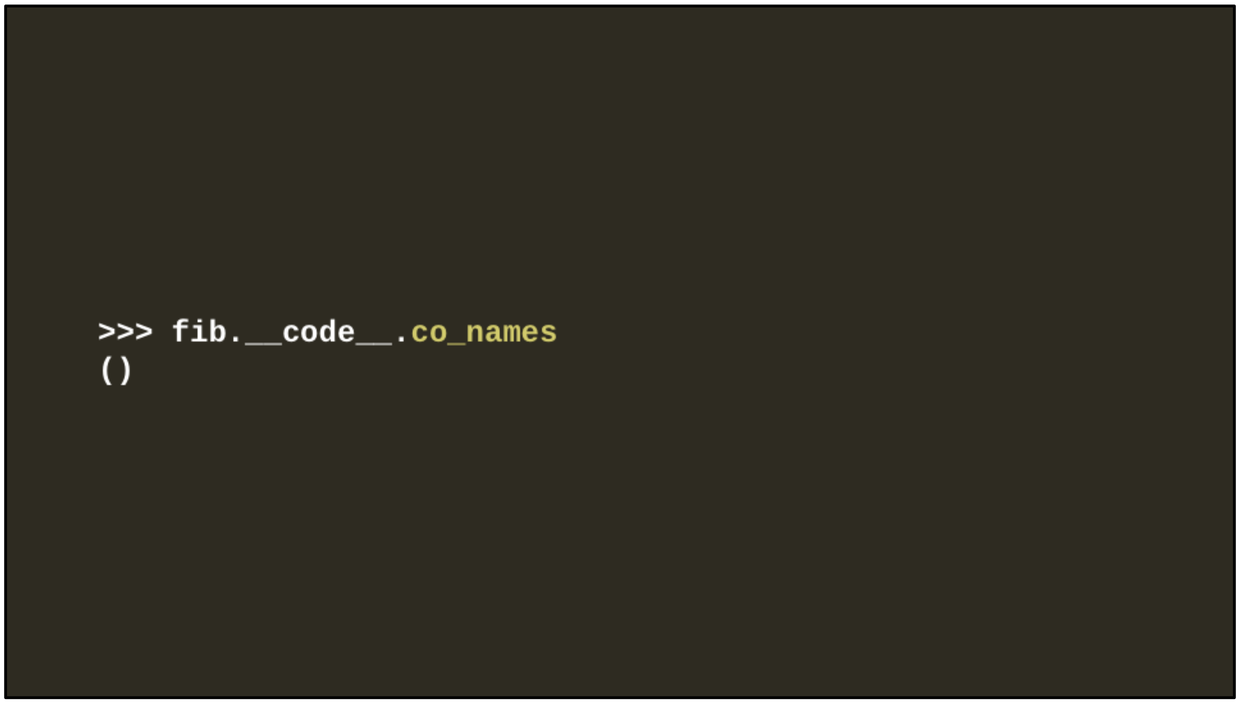
co_consts is a tuple of all the constants (literals) used inside the function body.

Notice that None is in here despite not occurring in the function body. That’s because, if no explicit return statement is ever reached (and Python can’t figure out in advance if it will!), a Python function returns None. So it’s present just in case it’s needed.



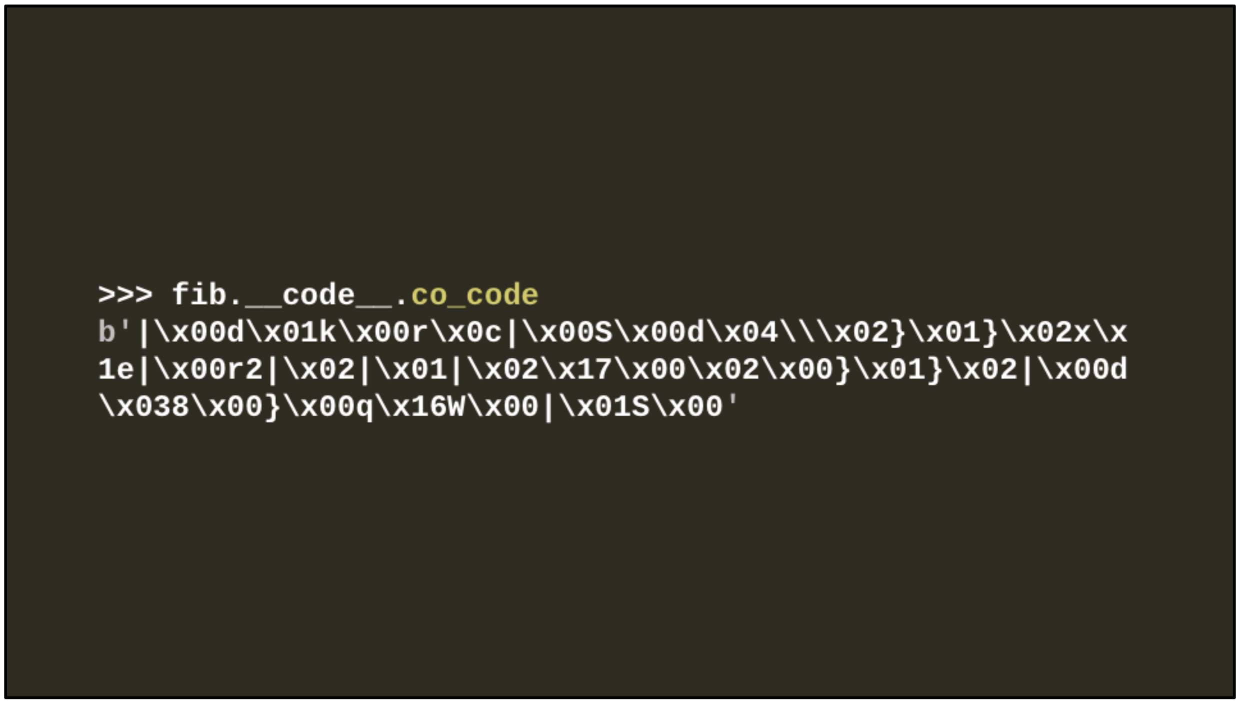
```
>>> fib.__code__.co_varnames
('n', 'current', 'next')
```

co_varnames is another tuple, containing the names of all the local variables of the function.



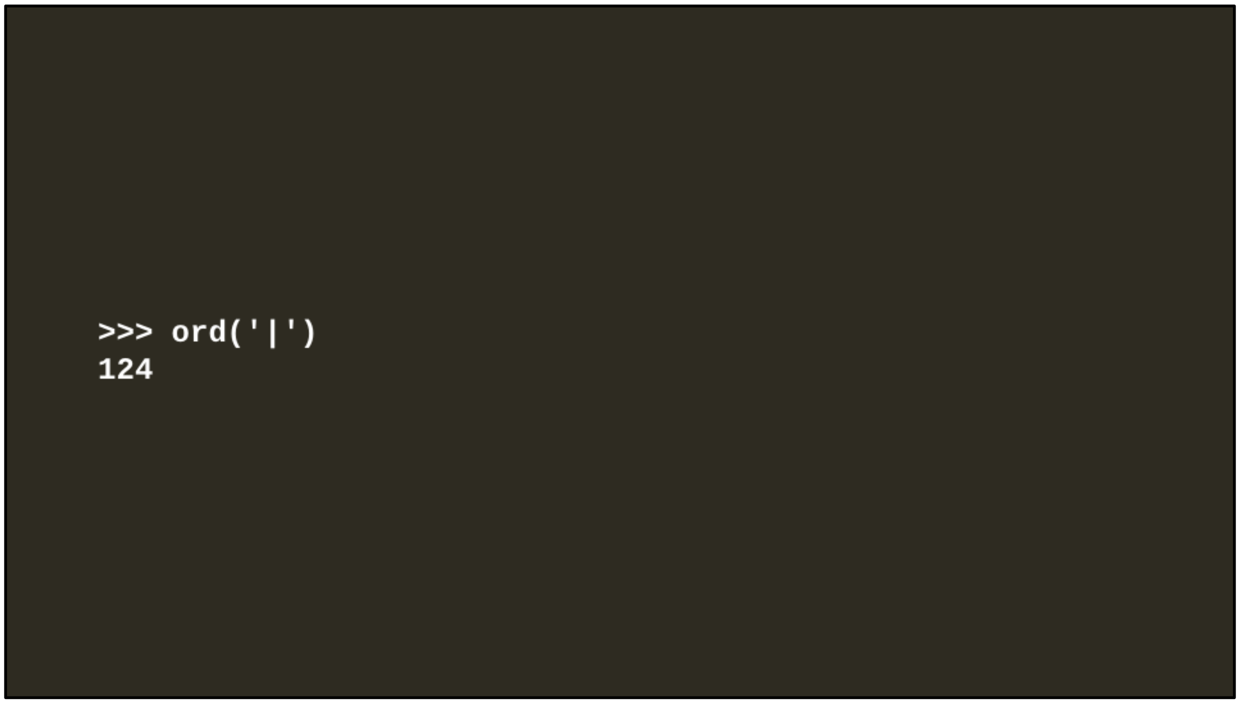
```
>>> fib.__code__.co_names
()
```

Finally, co_names is a tuple containing any nonlocal names used in the function body. The Fibonacci function didn’t use any of those.



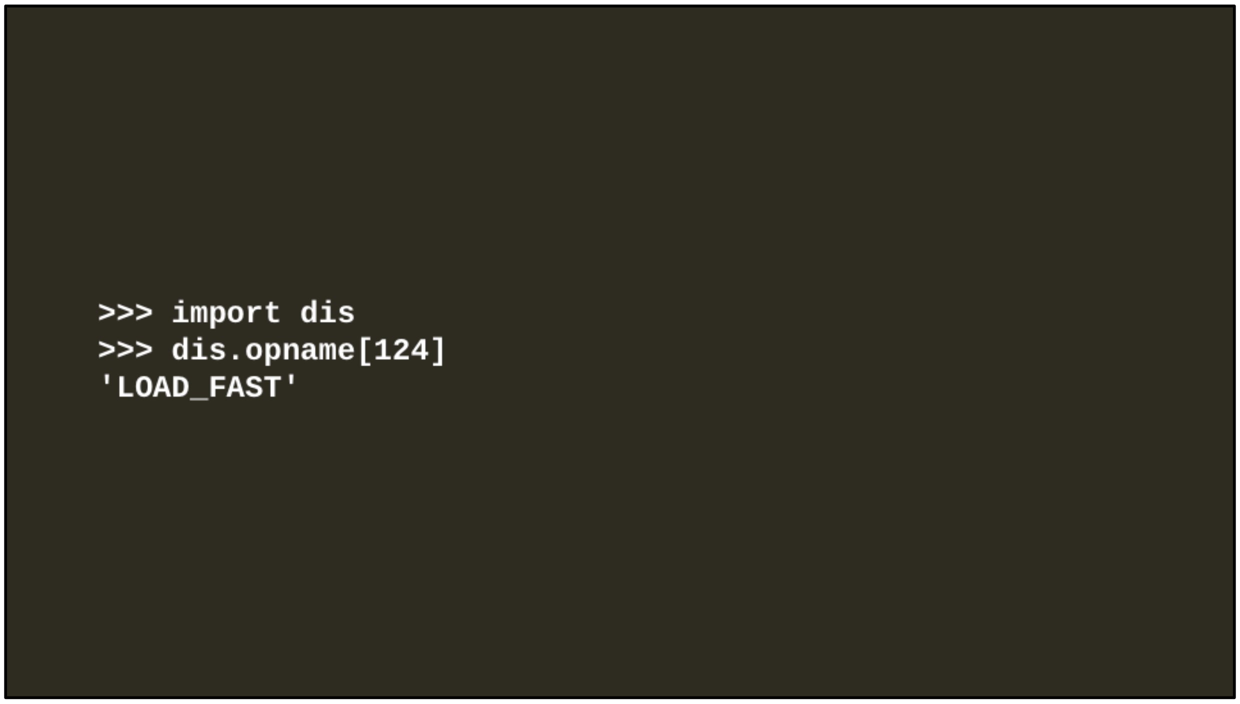
```
>>> fib.__code__.co_code
b'|\x00d\x01k\x00r\x0c|\x00S\x00d\x04\\\x02}\x01}\x02x\x
1e|\x00r2|\x02|\x01|\x02\x17\x00\x02\x00}\x01}\x02|\x00d
\x038\x00}\x00q\x16W\x00|\x01S\x00'
```

And this is the compiled bytecode of the Fibonacci function, as a Python bytes object. Ignore the fact that some of the bytes are ASCII printable. This is not a string!



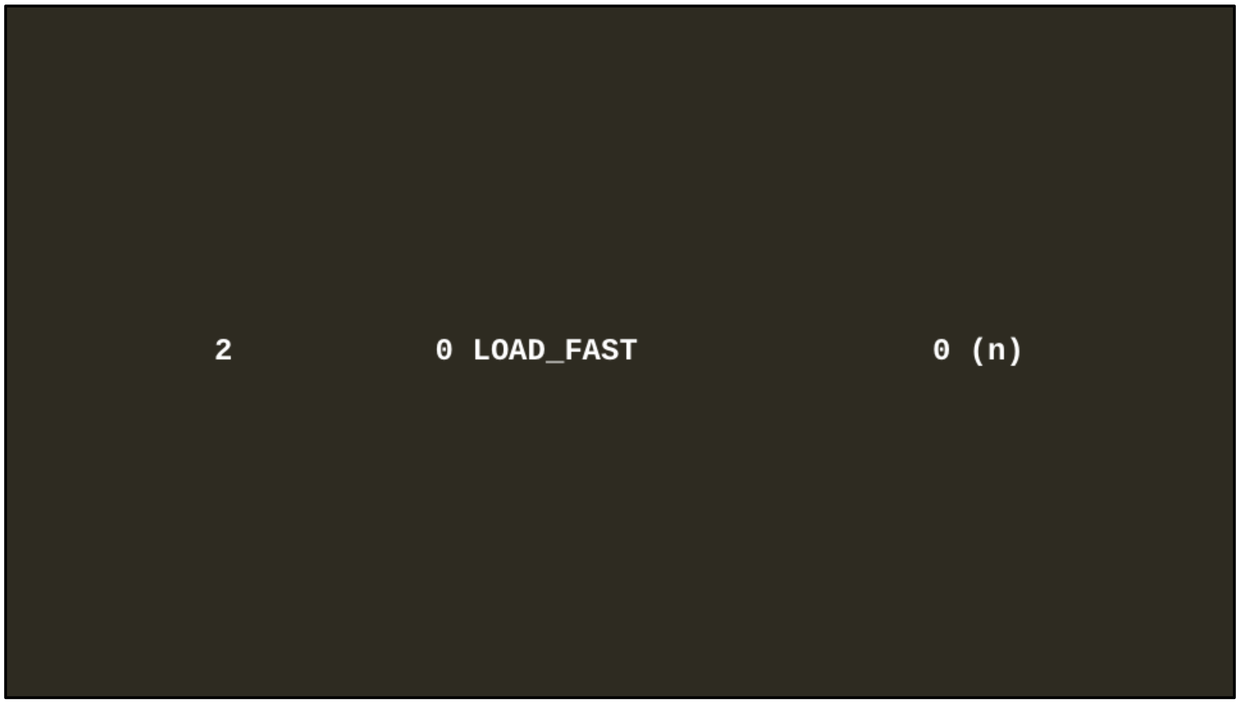
```
>>> ord('|')
124
```

The first byte printed as a pipe character. We can ask Python for its decimal byte value.



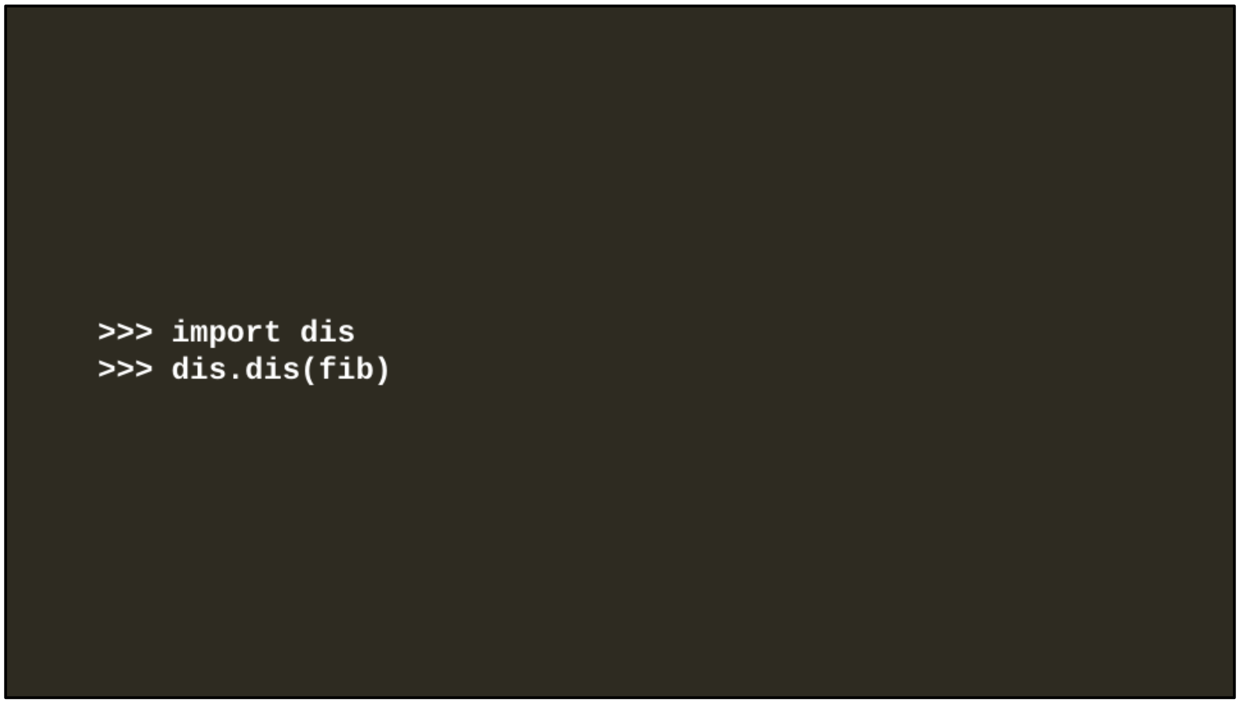
```
>>> import dis
>>> dis.opname[124]
'LOAD_FAST'
```

And then we can go to the dis module in the standard library, and ask Python what bytecode operation has decimal value 124.



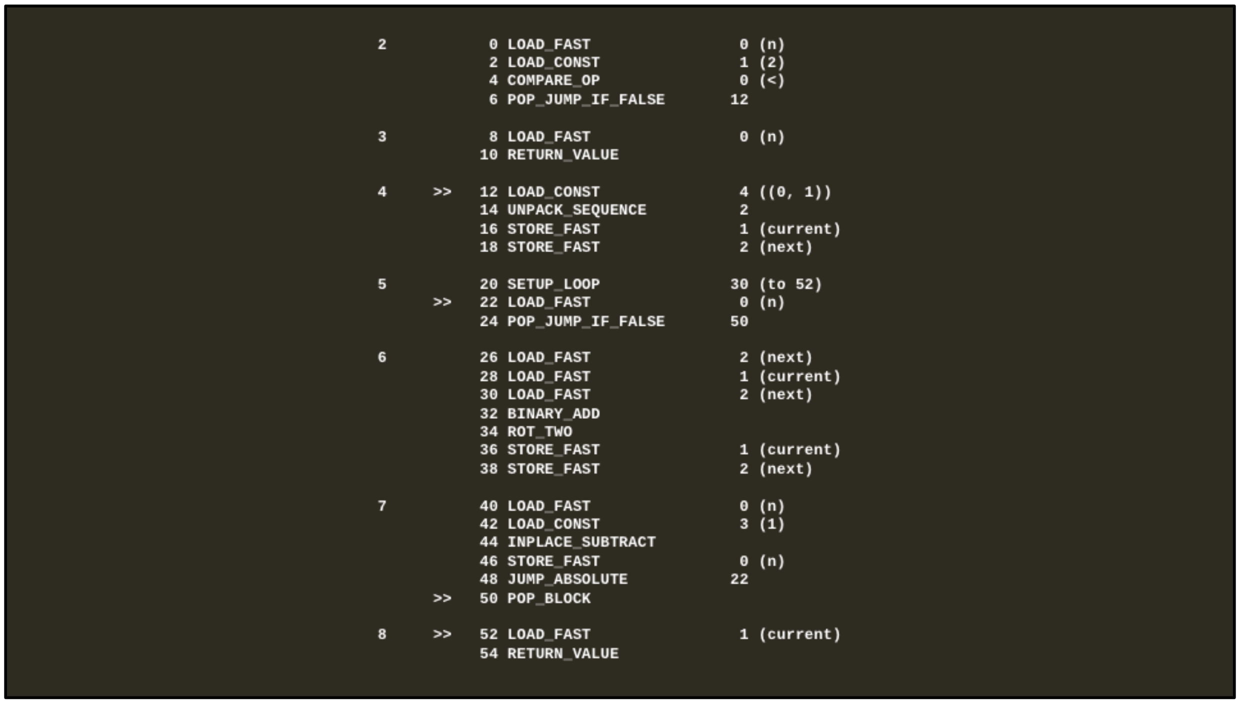
```
  2           0 LOAD_FAST                0 (n)
```

Remember that first slide? This was the first line in it. The second byte of this function’s bytecode is a 0. The first (zeroth) item in the co_varnames tuple was the local variable ‘n’. LOAD_FAST tells Python’s virtual machine to load the value associated with the first name in co_varnames, and push it on the top of the evaluation stack.



```
>>> import dis
>>> dis.dis(fib)
```

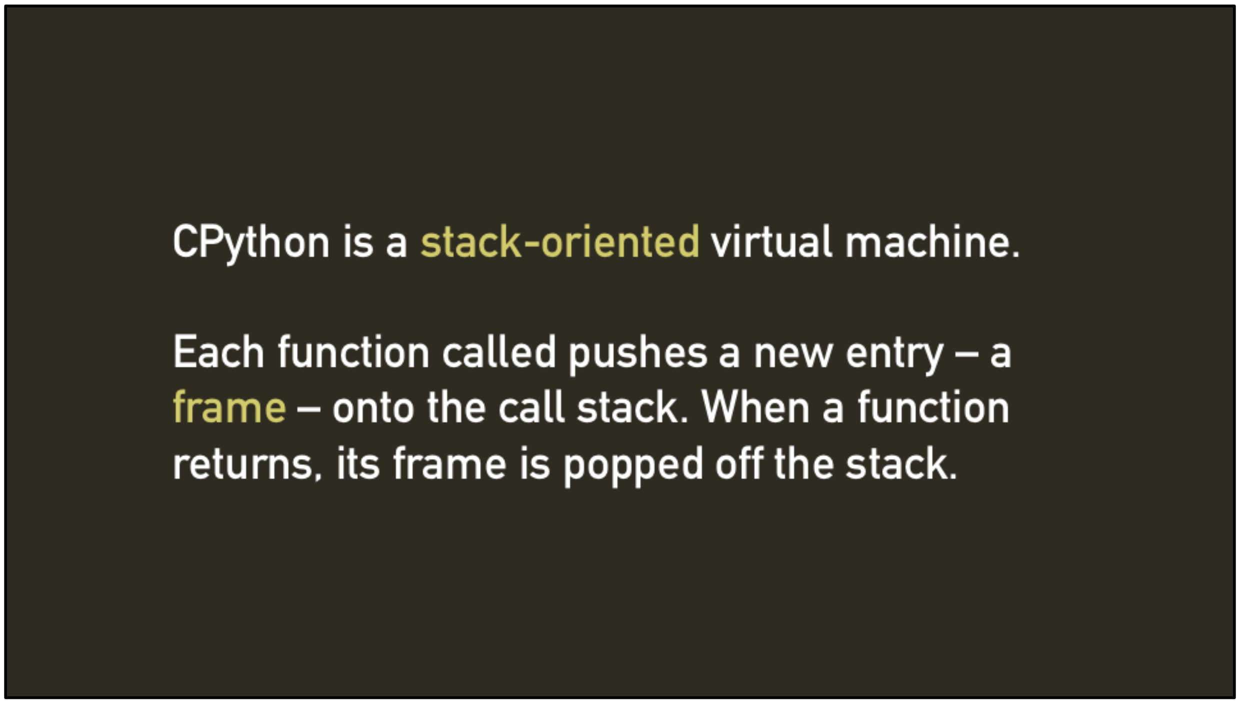
Now that we've seen see how to access the bytecode and make sense of it manually, here's the shortcut to seeing human-readable bytecode in a Python interpreter. It prints out the contents of the first slide.



```
  2           0 LOAD_FAST                0 (n)
              2 LOAD_CONST               1 (2)
              4 COMPARE_OP               0 (<)
              6 POP_JUMP_IF_FALSE       12

  3           8 LOAD_FAST                0 (n)
             10 RETURN_VALUE

  4     >>   12 LOAD_CONST               4 ((0, 1))
             14 UNPACK_SEQUENCE          2
             16 STORE_FAST               1 (current)
             18 STORE_FAST               2 (next)

  5          20 SETUP_LOOP              30 (to 52)
        >>   22 LOAD_FAST                0 (n)
             24 POP_JUMP_IF_FALSE       50

  6          26 LOAD_FAST                2 (next)
             28 LOAD_FAST                1 (current)
             30 LOAD_FAST                2 (next)
             32 BINARY_ADD
             34 ROT_TWO
             36 STORE_FAST               1 (current)
             38 STORE_FAST               2 (next)

  7          40 LOAD_FAST                0 (n)
             42 LOAD_CONST               3 (1)
             44 INPLACE_SUBTRACT
             46 STORE_FAST               0 (n)
             48 JUMP_ABSOLUTE           22
        >>   50 POP_BLOCK

  8     >>   52 LOAD_FAST                1 (current)
             54 RETURN_VALUE
```

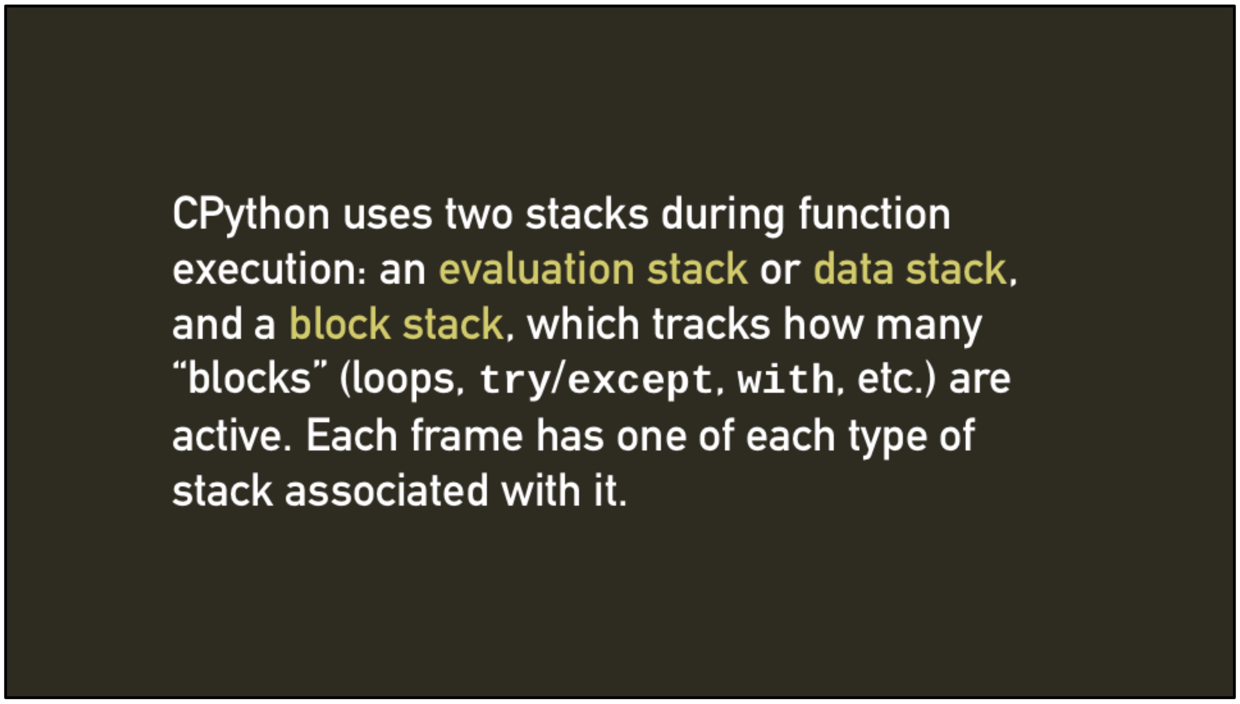
The output contains the line number each instruction came from. There's also another number in front of each instruction; that's the offset, in bytes, from the start of the function's bytecode to that instruction. In Python 3.6, every instruction takes two bytes (instruction code plus argument, and the argument is ignored for instructions that don't take arguments), so the offset is an even number. Previously, the offset could be odd if instructions that didn't take arguments were involved.

Right-pointing angle brackets indicate instructions which are jump targets (some other instruction, like an if statement or the continuation of a loop, might say to go here next).


CPython is a stack-oriented virtual machine.

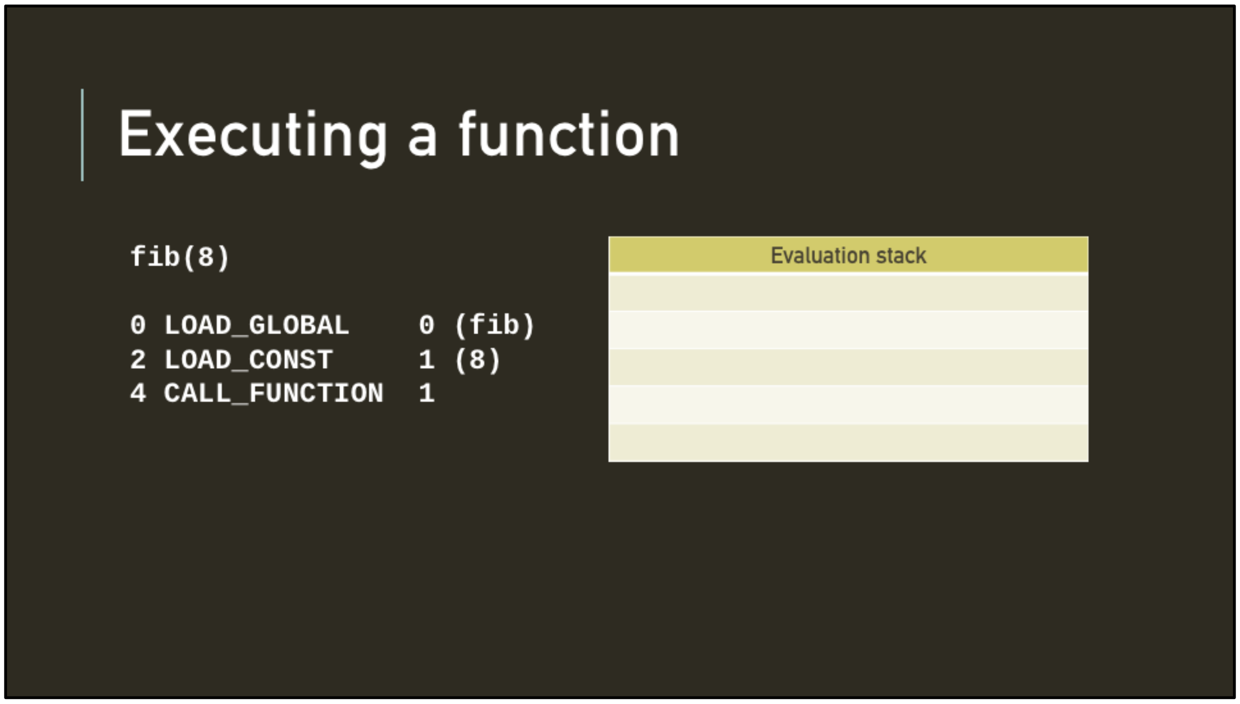
Each function called pushes a new entry – a frame – onto the call stack. When a function returns, its frame is popped off the stack.

The Python VM's fundamental data structure is a stack, which supports two operations: push (put an item on "top") and pop (remove whatever item was on "top" and return its value). The interpreter maintains a call stack with one frame (item) for each function call. Whenever a function returns, its frame is popped off the top of the call stack, and the return value is pushed on the evaluation stack of the calling frame.


CPython uses two stacks during function execution: an evaluation stack or data stack, and a block stack, which tracks how many "blocks" (loops, `try`/`except`, `with`, etc.) are active. Each frame has one of each type of stack associated with it.

Within each call-stack frame, Python sets up a frame-local evaluation stack and a frame-local block stack for execution of the function.

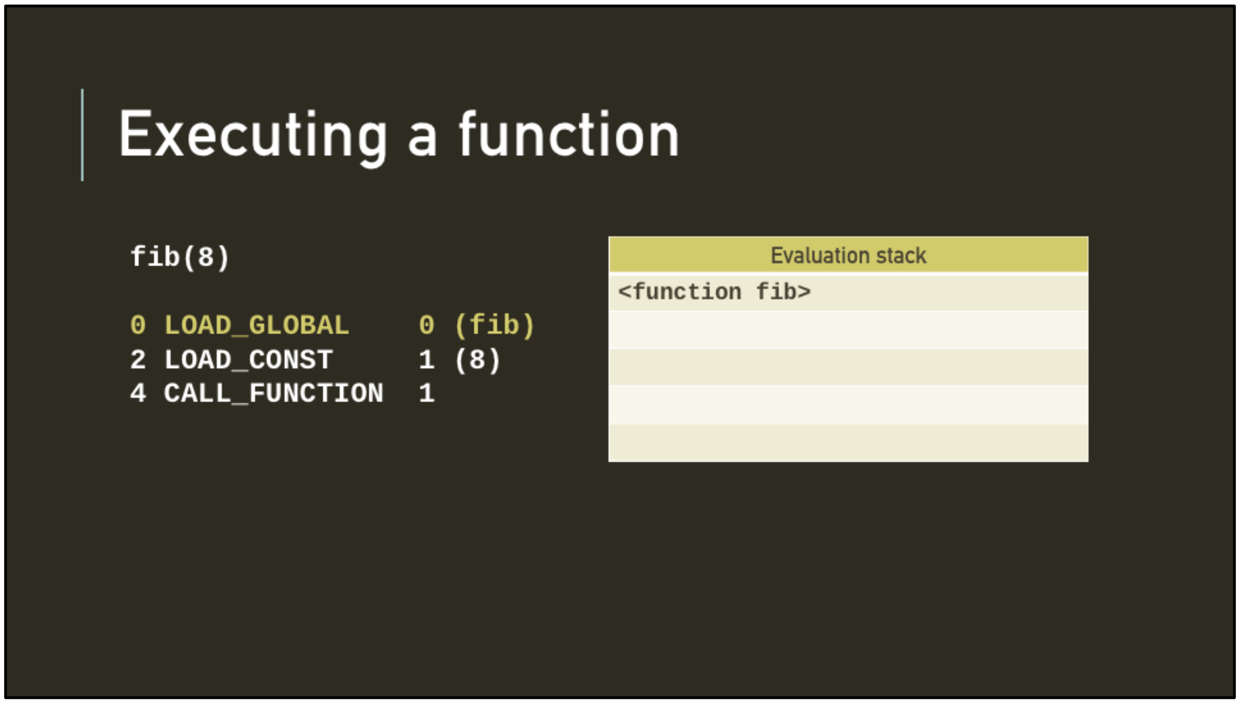
The block stack is necessary since loops and other blocks can be nested, and control flow keywords can break out of them; Python needs to know *which* block is being broken out of.



Executing a function

```
fib(8)

0 LOAD_GLOBAL    0 (fib)
2 LOAD_CONST     1 (8)
4 CALL_FUNCTION  1
```

Here's an example of a function call, calculating the 8th Fibonacci number. It turns into three bytecode operations

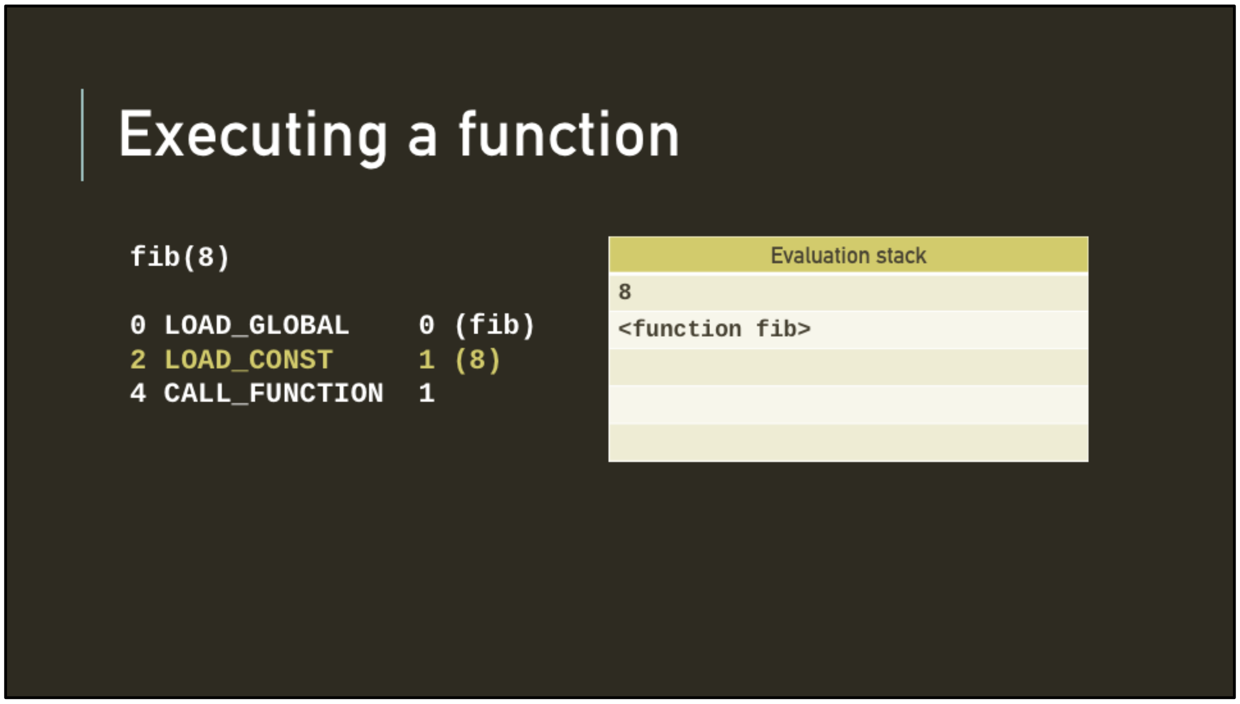


Executing a function

```
fib(8)

0 LOAD_GLOBAL     0 (fib)
2 LOAD_CONST      1 (8)
4 CALL_FUNCTION   1
```

Evaluation stack
`<function fib>`

The first, LOAD_GLOBAL, looks up a name from the global (or nonlocal, if we're in a function body) namespace, and pushes it on top of the stack. In this case, it's 'fib' function.

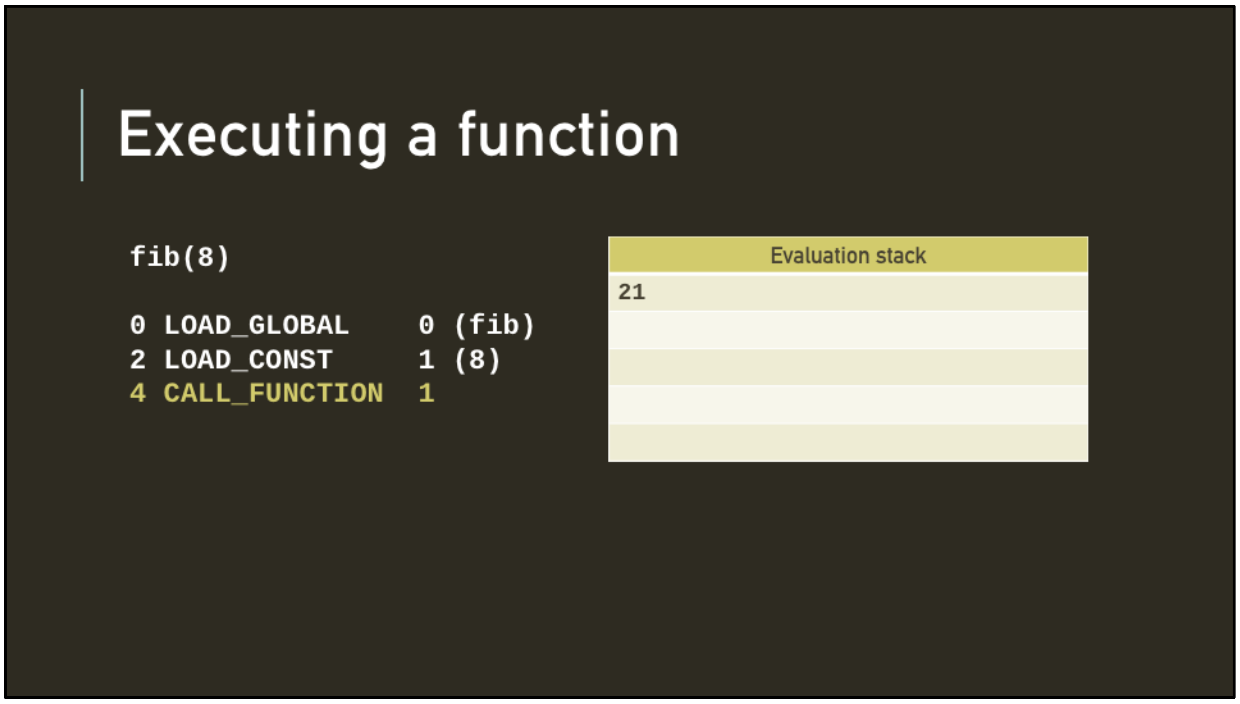


Executing a function

```
fib(8)

0 LOAD_GLOBAL     0 (fib)
2 LOAD_CONST      1 (8)
4 CALL_FUNCTION   1
```

Evaluation stack
8
<function fib>

Next, LOAD_CONST loads the literal integer 8 and pushes it on top of the stack.



Executing a function

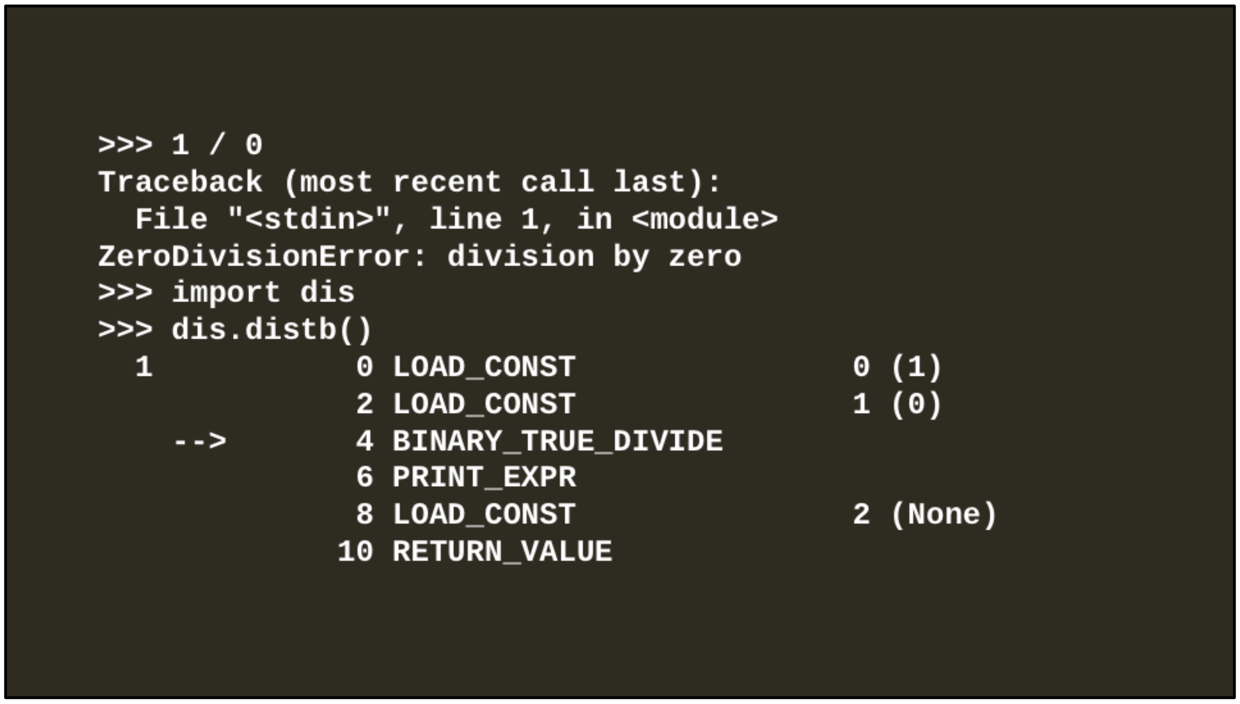
```
fib(8)

0 LOAD_GLOBAL     0 (fib)
2 LOAD_CONST      1 (8)
4 CALL_FUNCTION   1
```

Evaluation stack
21

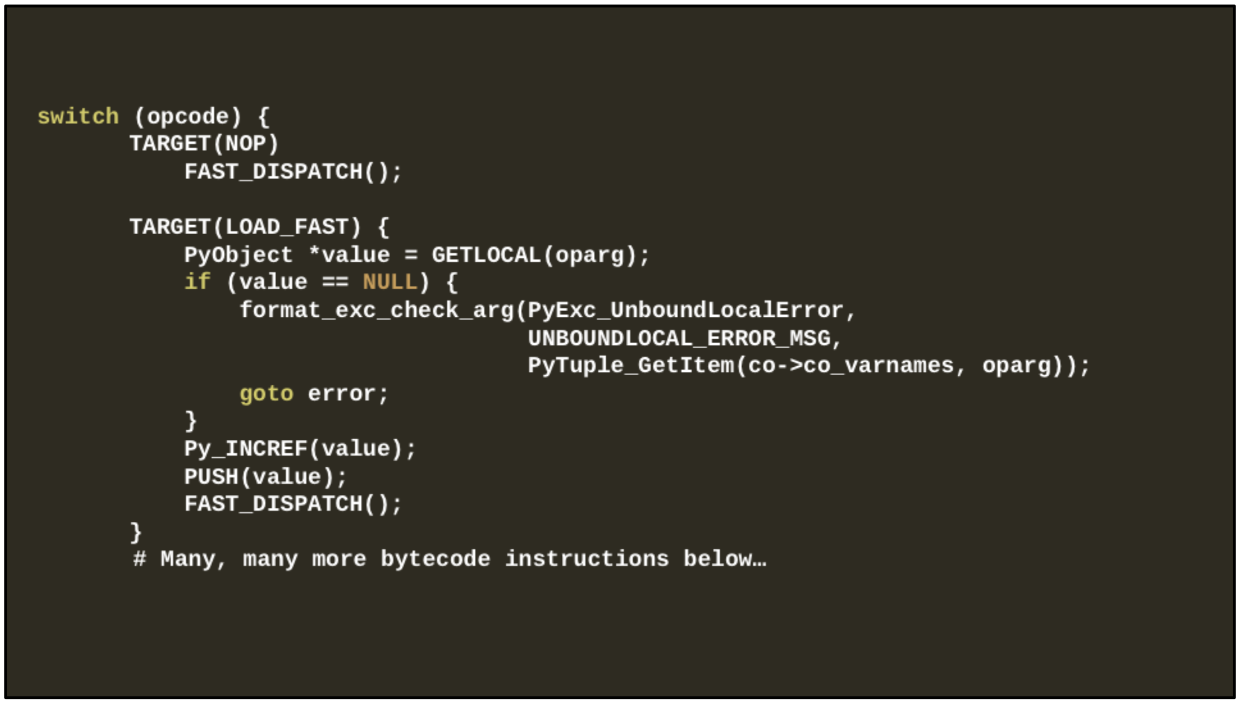
Finally, CALL_FUNCTION carries out the function call. The argument 1 indicates that there's 1 positional argument involved. Python will count that many items down on the stack to find the function to call, popping each of them. Then it pops the function, executes the body of the function in a new call-stack frame, and pushes the return value to the top of the stack. There are other bytecode instructions for calling a function with keyword arguments, and calling a function by using iterable- and mapping-unpacking constructs for arguments.

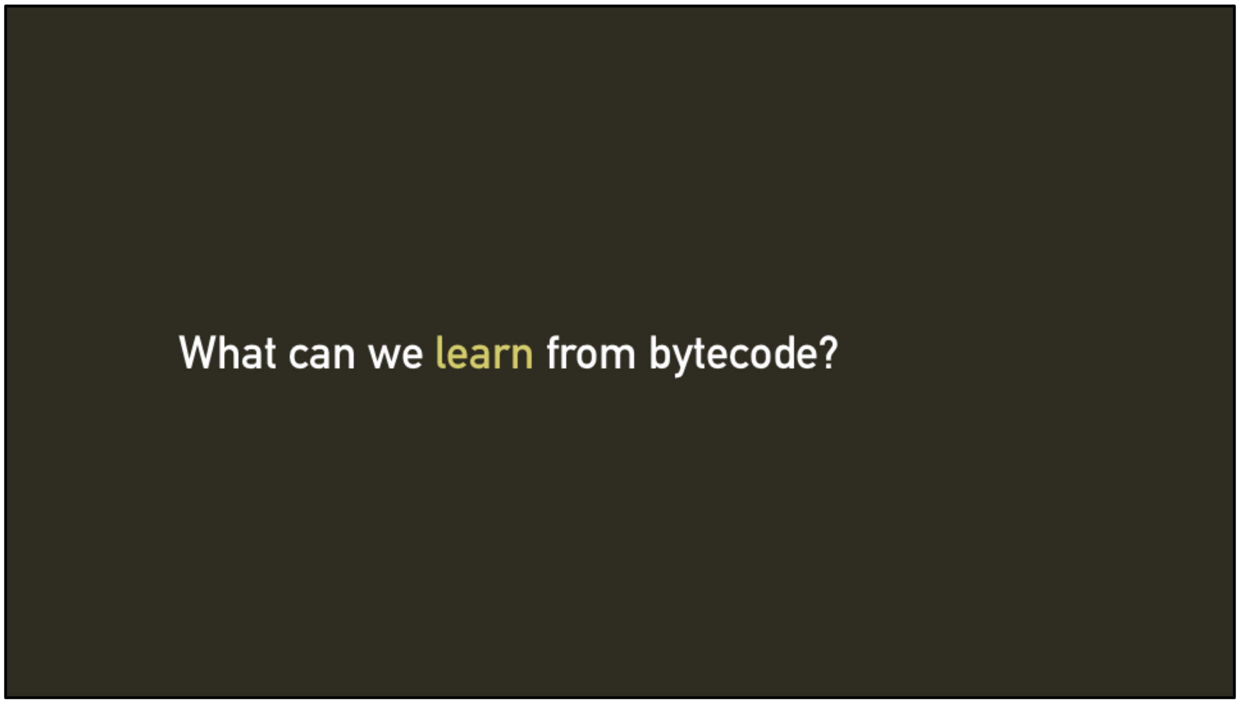
https://docs.python.org/3/library/dis.html

The documentation for the dis module includes a complete list of CPython’s bytecode instructions, along with what each one does and what argument it takes.



```
>>> 1 / 0
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
ZeroDivisionError: division by zero
>>> import dis
>>> dis.distb()
  1           0 LOAD_CONST               0 (1)
              2 LOAD_CONST               1 (0)
    -->       4 BINARY_TRUE_DIVIDE
              6 PRINT_EXPR
              8 LOAD_CONST               2 (None)
             10 RETURN_VALUE
```


```
switch (opcode) {
       TARGET(NOP)
           FAST_DISPATCH();

       TARGET(LOAD_FAST) {
           PyObject *value = GETLOCAL(oparg);
           if (value == NULL) {
               format_exc_check_arg(PyExc_UnboundLocalError,
                                    UNBOUNDLOCAL_ERROR_MSG,
                                    PyTuple_GetItem(co->co_varnames, oparg));
               goto error;
           }
           Py_INCREF(value);
           PUSH(value);
           FAST_DISPATCH();
       }
       # Many, many more bytecode instructions below…
```

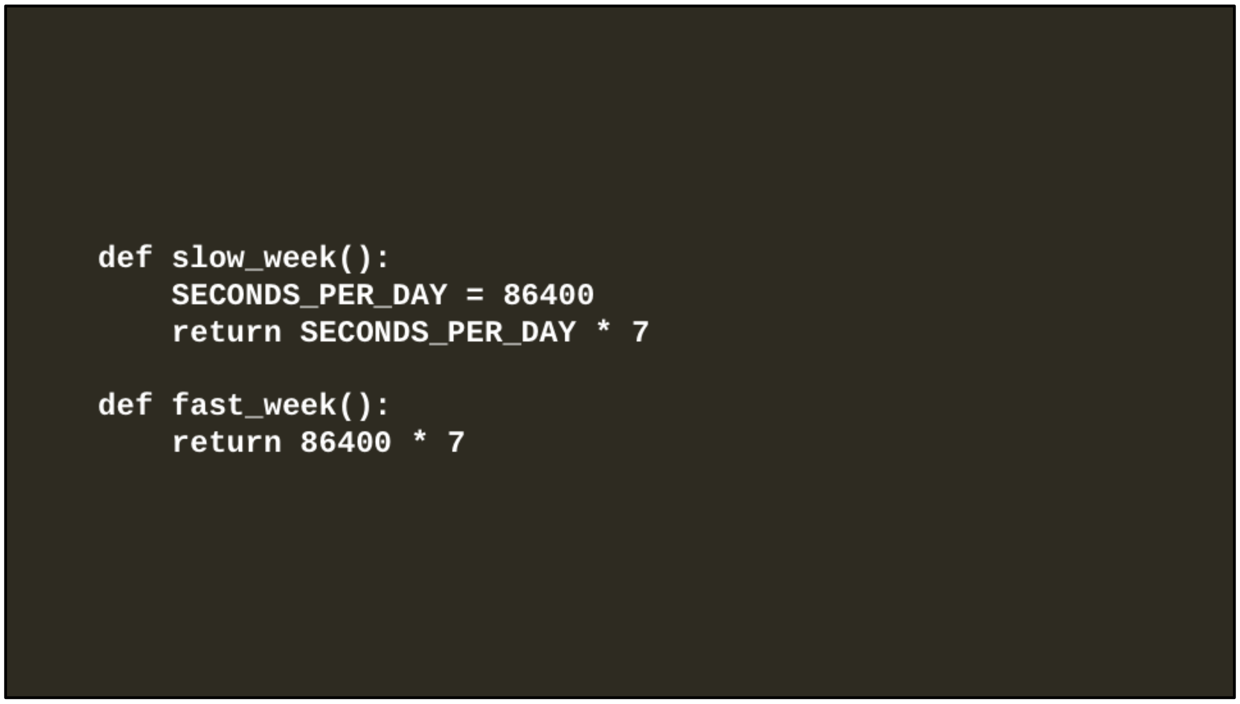


What can we learn from bytecode?

If you've ever used FORTH or Factor or another stack-oriented language, the bytecode instructions and their effects may be a bit easier for you to understand. If not, this is a cool and somewhat unusual way to think about programming, and is worth learning just for that.

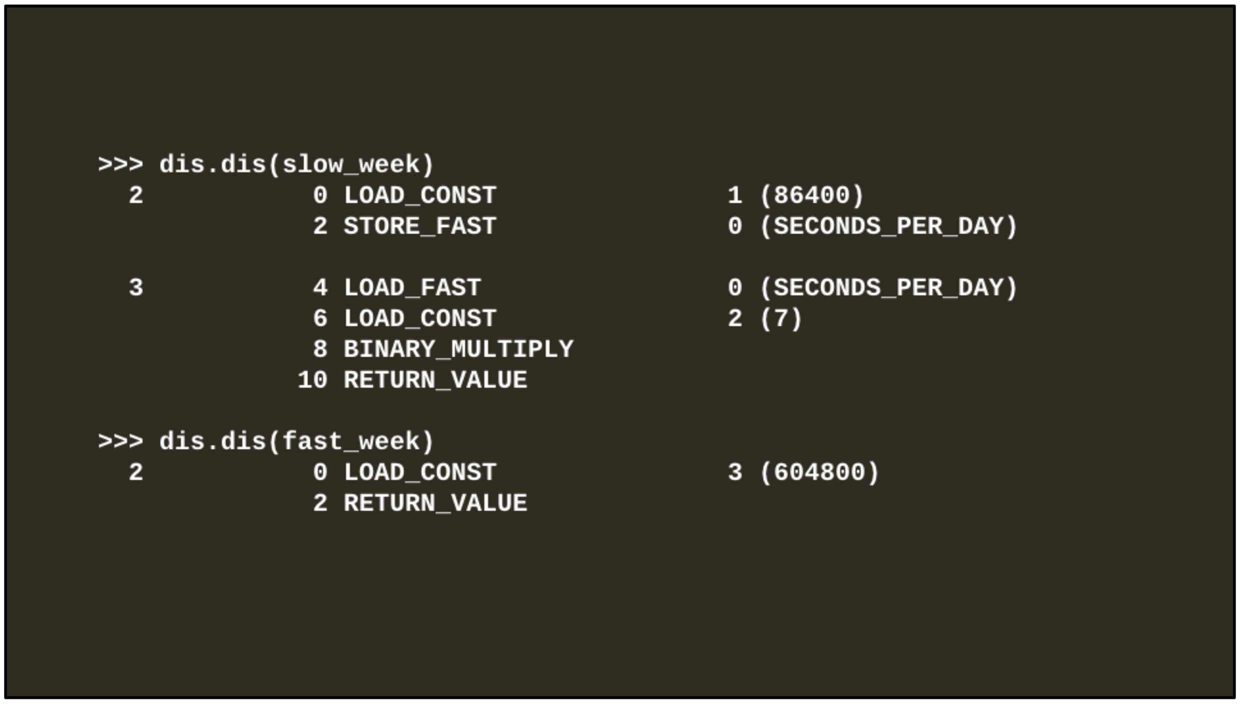
It also has more practical purposes. People often talk about C as "portable assembly", and how easy it is to reason about what a given piece of C code will do – bytecode is the "assembly" of Python, and studying how Python transforms your source code into bytecode will give you insights into what any given piece of Python code will do.

And, of course, we can learn things about how Python itself works, and reason about performance.



```python
def slow_week():
    SECONDS_PER_DAY = 86400
    return SECONDS_PER_DAY * 7

def fast_week():
    return 86400 * 7
```

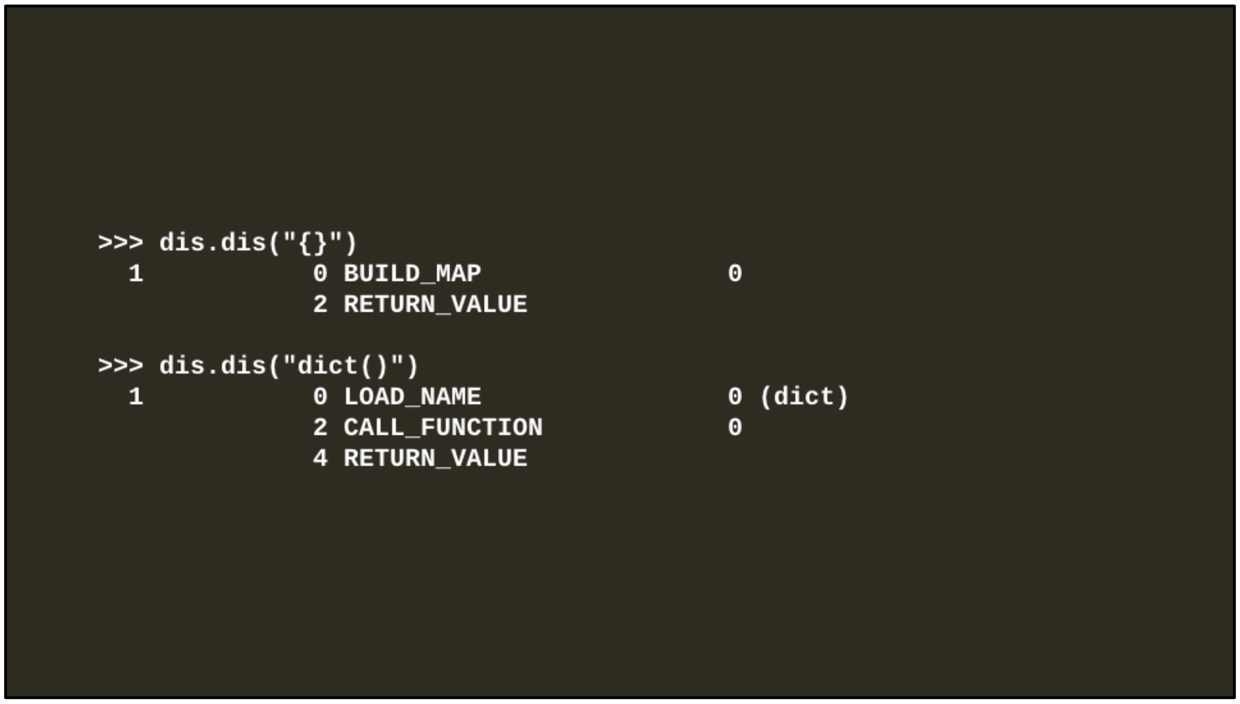
Here are a couple functions which each calculate the number of seconds in a week.
Can you guess which one is faster?



```
>>> dis.dis(slow_week)
  2           0 LOAD_CONST               1 (86400)
              2 STORE_FAST               0 (SECONDS_PER_DAY)

  3           4 LOAD_FAST                0 (SECONDS_PER_DAY)
              6 LOAD_CONST               2 (7)
              8 BINARY_MULTIPLY
             10 RETURN_VALUE

>>> dis.dis(fast_week)
  2           0 LOAD_CONST               3 (604800)
              2 RETURN_VALUE
```

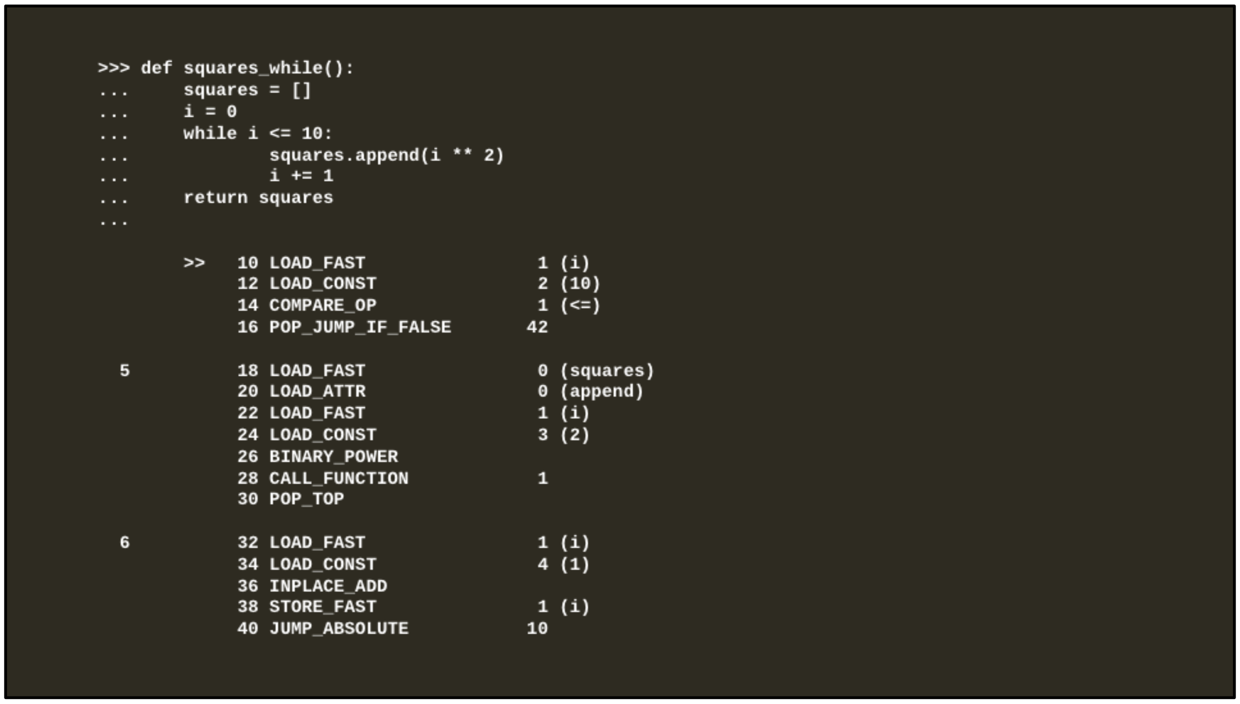
Reading bytecode can show you interesting optimizations. In this case, fast_week is only two bytecode instructions because it only used arithmetic on constant values, so Python did the arithmetic at compile time. But slow_week() is four instructions because it involves a variable, and Python can't statically optimize that away.

This isn't the only sneaky optimization in Python. For example, depending on how you compile your Python, there are some tricks that try to speed up execution by knowing that certain bytecode instructions tend to follow each other; combined with the processor's branch prediction, this allows drastically reduced overhead when dispatching certain pairs of instructions that often occur together. This would have sped up the Fibonacci function from earlier, by predicting a POP_JUMP_IF_FALSE as a likely followup to a COMPARE_OP.



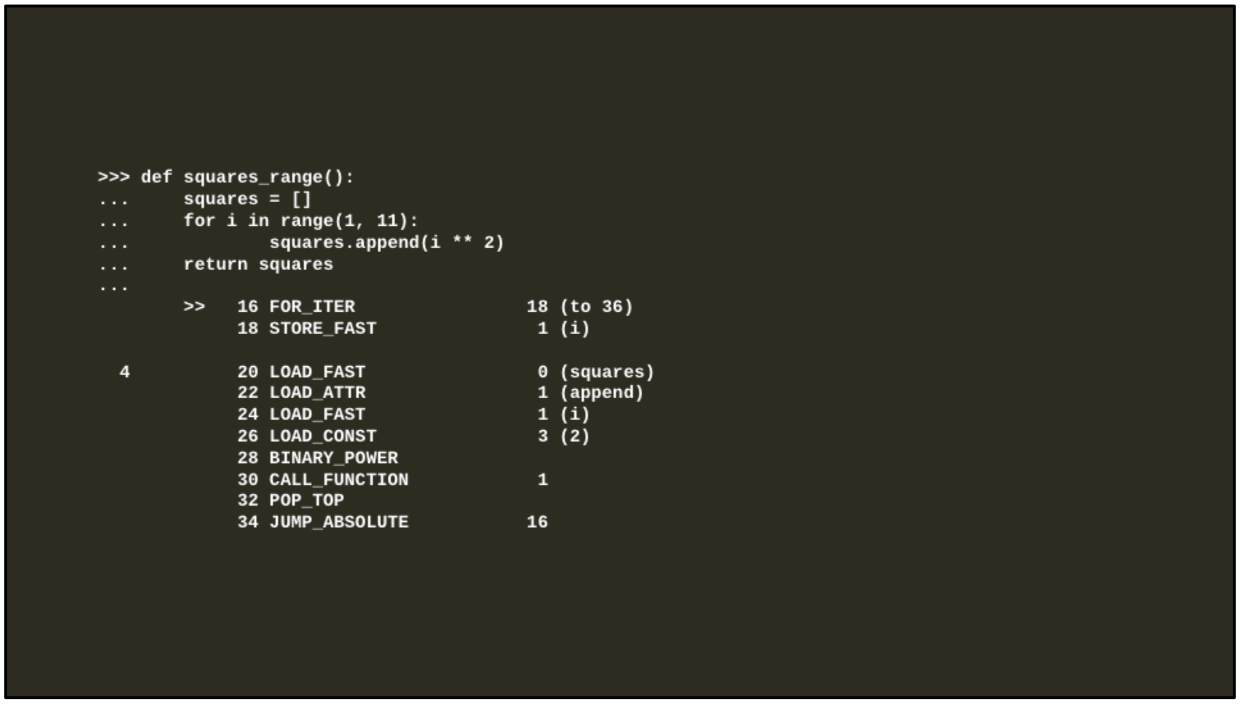
```
>>> dis.dis("{}")
  1           0 BUILD_MAP                0
              2 RETURN_VALUE

>>> dis.dis("dict()")
  1           0 LOAD_NAME                0 (dict)
              2 CALL_FUNCTION            0
              4 RETURN_VALUE
```

Here’s a common type of Python performance question: why is the dictionary literal syntax faster than the dict() function? The answer, once again, is in the bytecode.



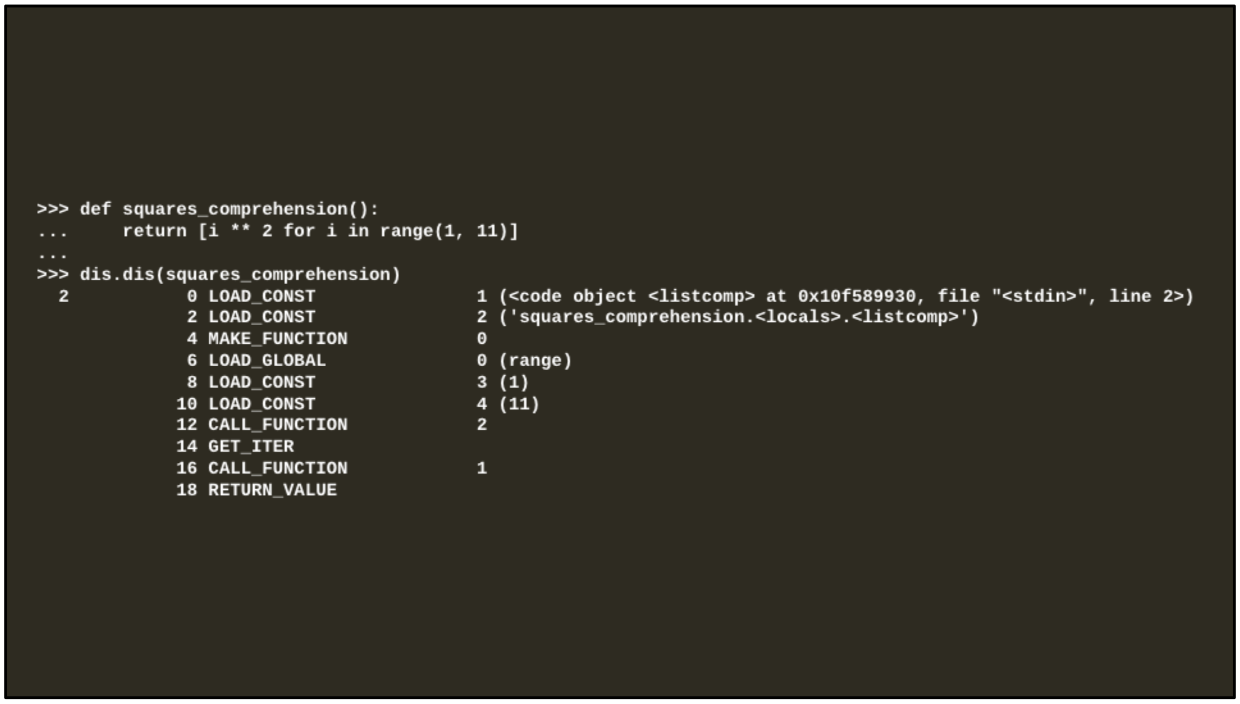
```
>>> def squares_while():
...     squares = []
...     i = 0
...     while i <= 10:
...             squares.append(i ** 2)
...             i += 1
...     return squares
...

        >>   10 LOAD_FAST                1 (i)
             12 LOAD_CONST               2 (10)
             14 COMPARE_OP               1 (<=)
             16 POP_JUMP_IF_FALSE       42

  5          18 LOAD_FAST                0 (squares)
             20 LOAD_ATTR                0 (append)
             22 LOAD_FAST                1 (i)
             24 LOAD_CONST               3 (2)
             26 BINARY_POWER
             28 CALL_FUNCTION            1
             30 POP_TOP

  6          32 LOAD_FAST                1 (i)
             34 LOAD_CONST               4 (1)
             36 INPLACE_ADD
             38 STORE_FAST               1 (i)
             40 JUMP_ABSOLUTE           10
```

Here’s a function that calculates and returns a list of the first ten perfect squares. The body of its loop is 15 bytecode instructions.



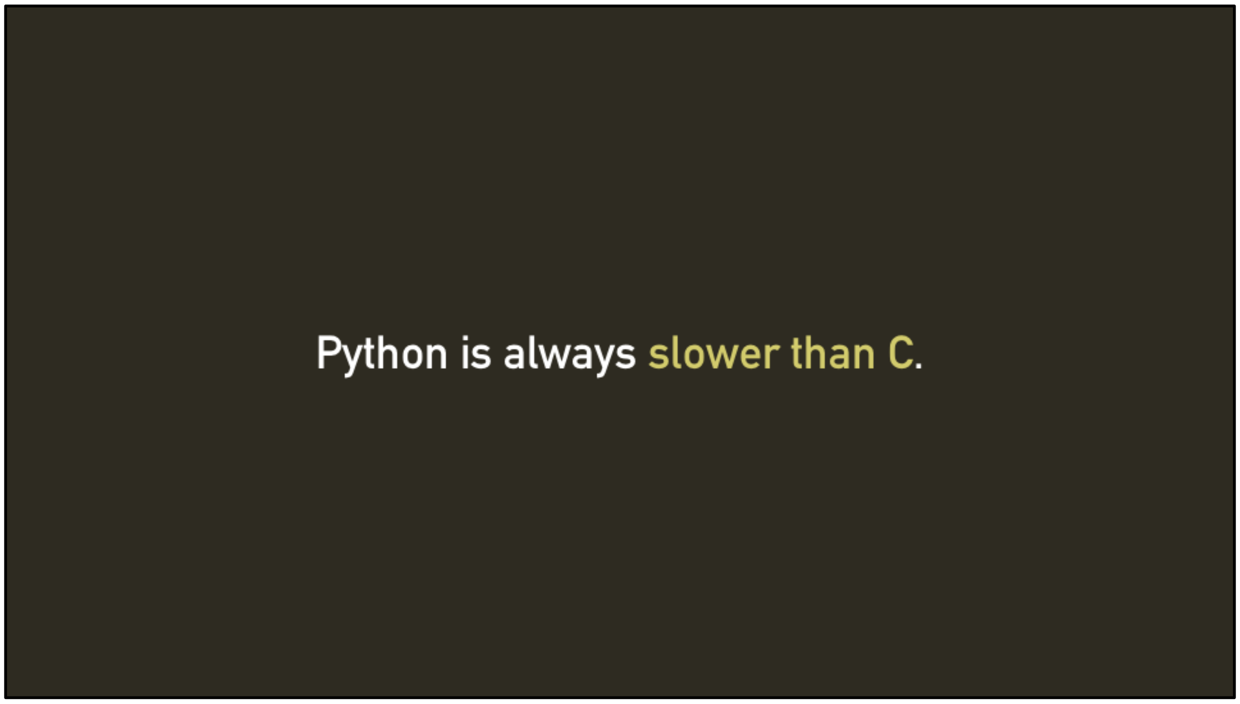
```
>>> def squares_range():
...     squares = []
...     for i in range(1, 11):
...             squares.append(i ** 2)
...     return squares
...
        >>   16 FOR_ITER                18 (to 36)
             18 STORE_FAST               1 (i)

  4          20 LOAD_FAST                0 (squares)
             22 LOAD_ATTR                1 (append)
             24 LOAD_FAST                1 (i)
             26 LOAD_CONST               3 (2)
             28 BINARY_POWER
             30 CALL_FUNCTION            1
             32 POP_TOP
             34 JUMP_ABSOLUTE           16
```

Here's a slightly different version, using range() and a for loop instead of a while loop with explicit counter. Its loop body is 9 instructions!



```
>>> def squares_comprehension():
...     return [i ** 2 for i in range(1, 11)]
...
>>> dis.dis(squares_comprehension)
  2           0 LOAD_CONST               1 (<code object <listcomp> at 0x10f589930, file "<stdin>", line 2>)
              2 LOAD_CONST               2 ('squares_comprehension.<locals>.<listcomp>')
              4 MAKE_FUNCTION            0
              6 LOAD_GLOBAL              0 (range)
              8 LOAD_CONST               3 (1)
             10 LOAD_CONST               4 (11)
             12 CALL_FUNCTION            2
             14 GET_ITER
             16 CALL_FUNCTION            1
             18 RETURN_VALUE
```

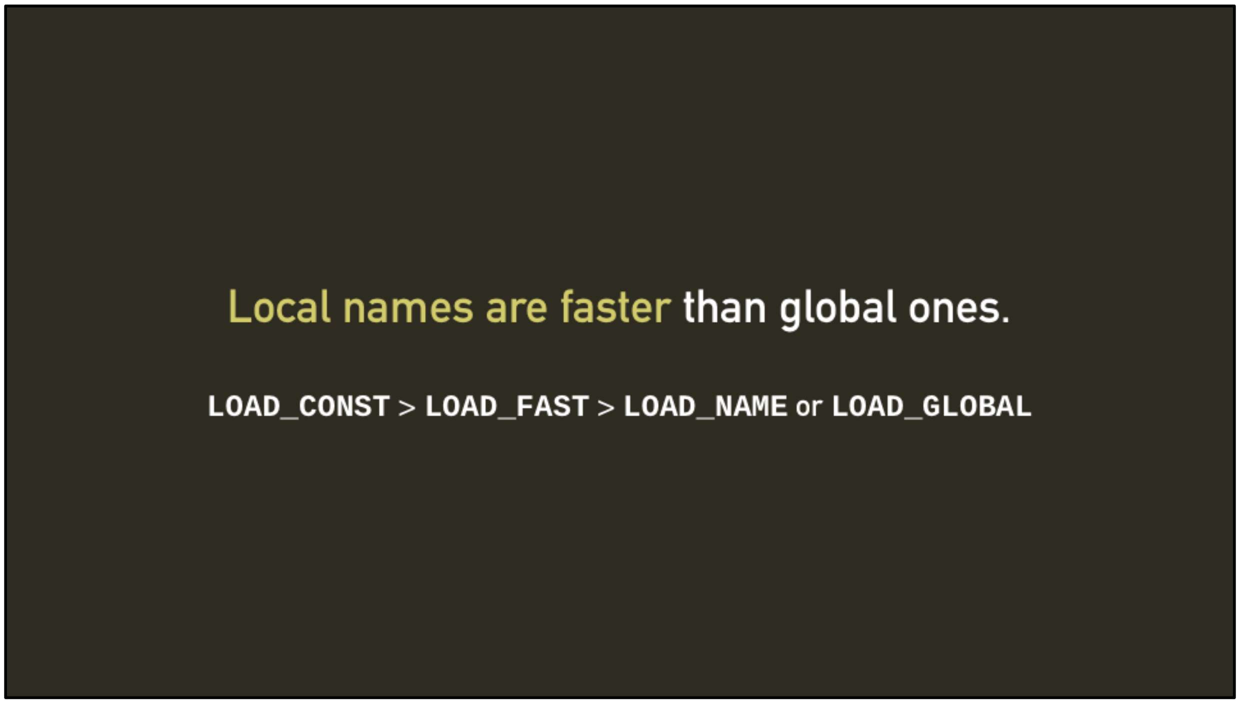
Finally, here's a version using a list comprehension. The entire function body is 9 instructions! The generated code object for the list comprehension is also 9 instructions; this makes a total of 18 instructions (compared to 20 total for the range + loop). We do have to build and execute a function here, though, so it's not a pure gain, and that's an important lesson: fewer instructions doesn't always mean faster, especially if there's a function call involved!



Python is always slower than C.

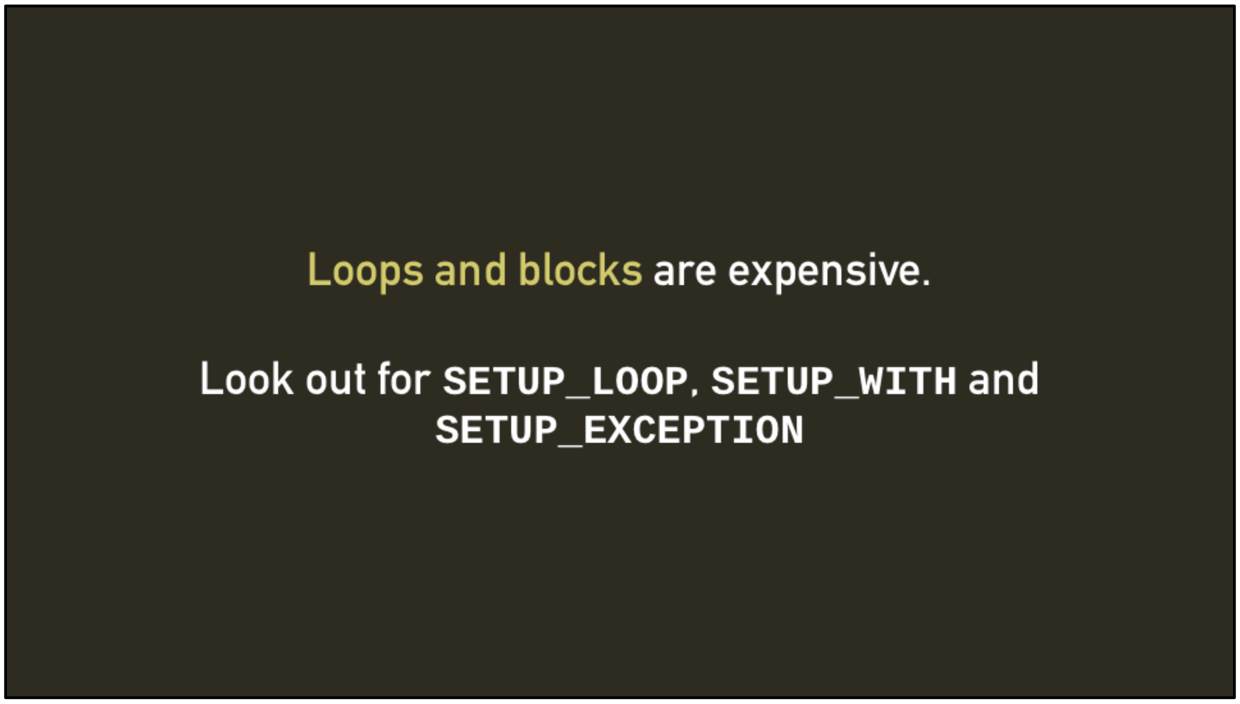
So let’s talk about some guidelines for reading bytecode and reasoning about performance.

This is the single most important rule of performance in Python. Any time you can avoid executing pure-Python code, you should.


Local names are faster than global ones.

LOAD_CONST > LOAD_FAST > LOAD_NAME or LOAD_GLOBAL

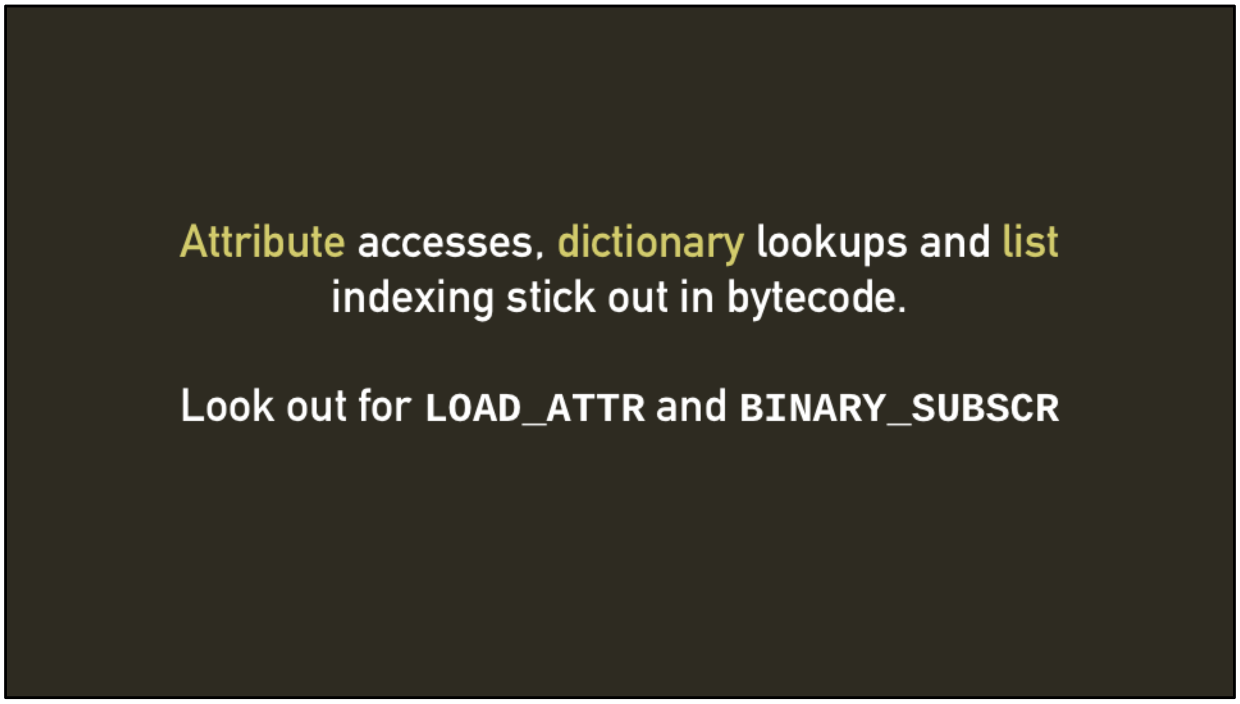
This isn’t necessarily obvious from the bytecode, but the implementations of these instructions have significantly different performance characteristics. A lot of this has to do with the complexity of the lookup; nonlocal names may have to search in multiple namespaces before they find what they’re looking for.



Loops and blocks are expensive.

Look out for **SETUP_LOOP**, **SETUP_WITH** and **SETUP_EXCEPTION**

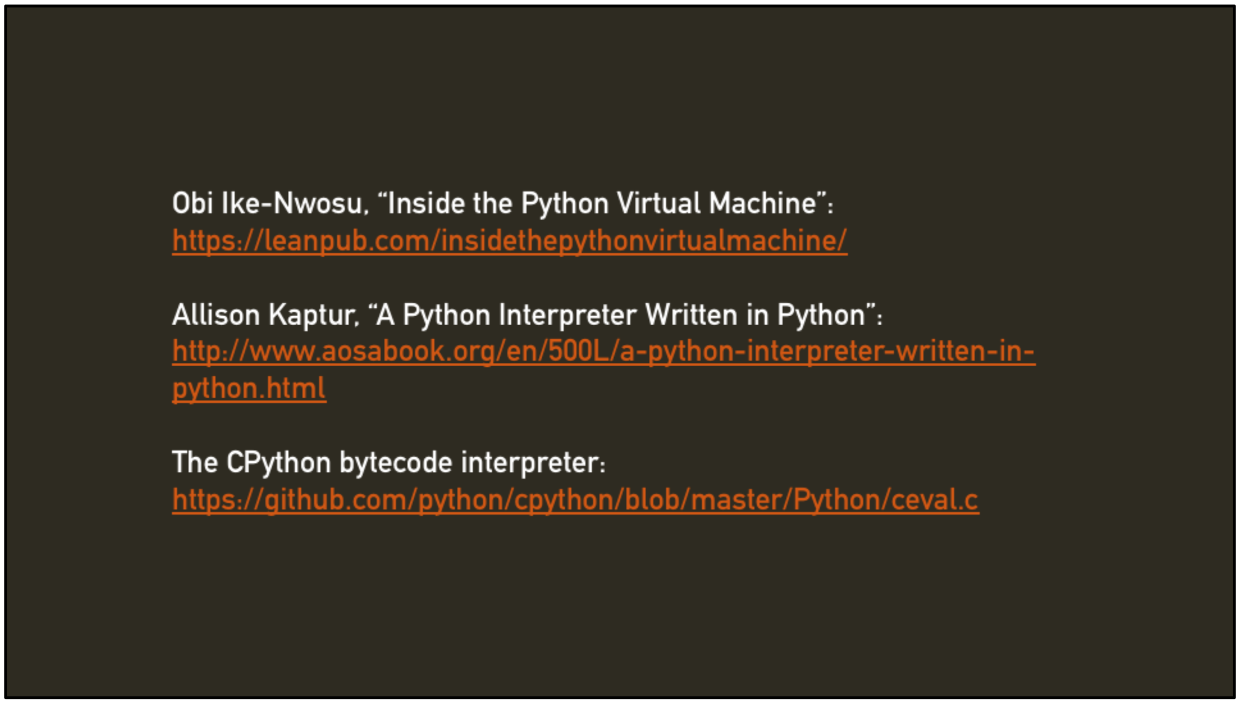
Any loop or block involves multiple instructions to enter and exit, instructions to set up any names local to the block, and pushing/popping the block stack.



Attribute accesses, dictionary lookups and list indexing stick out in bytecode.

Look out for `LOAD_ATTR` and `BINARY_SUBSCR`

This is part of why people often recommend aliasing an attribute of a nonlocal name in the function body. These operations can also be expensive, especially if you're doing them multiple levels deep, or once each time through a loop.



Obi Ike-Nwosu, "Inside the Python Virtual Machine":
https://leanpub.com/insidethepythonvirtualmachine/

Allison Kaptur, "A Python Interpreter Written in Python":
http://www.aosabook.org/en/500L/a-python-interpreter-written-in-python.html

The CPython bytecode interpreter:
https://github.com/python/cpython/blob/master/Python/ceval.c

The actual Python bytecode interpreter lives in the file ceval.c; the execution of bytecode instructions is handled by a large switch() statement starting around the middle of that file.

Questions?

@ubernostrum
https://www.b-list.org/