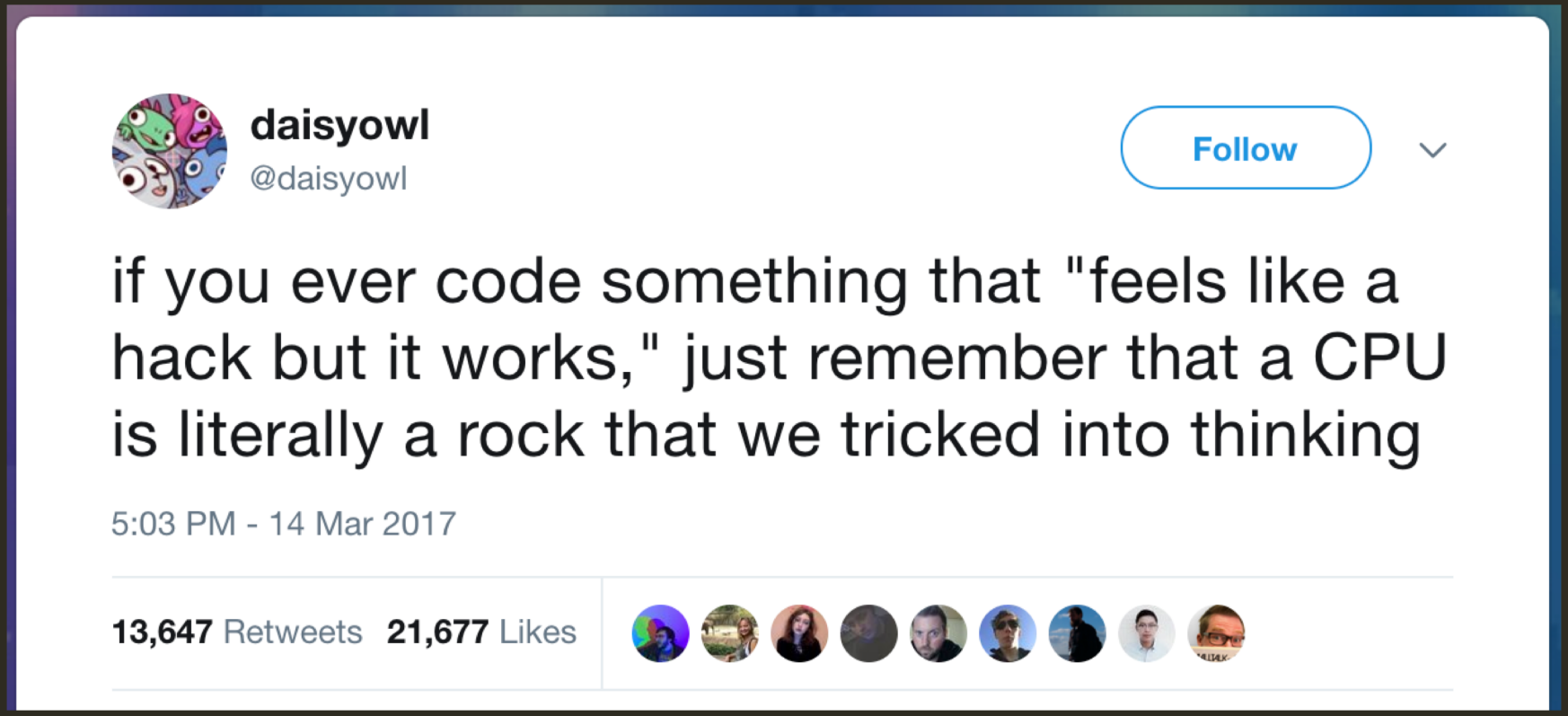
An introduction to Python bytecode

James Bennett

BayPiggies Python Preview

```
  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 LOAD_CONST               2 (0)
             22 STORE_FAST               3 (counter)

  6          24 SETUP_LOOP              34 (to 60)
        >>   26 LOAD_FAST                3 (counter)
             28 LOAD_FAST                0 (n)
             30 COMPARE_OP               0 (<)
             32 POP_JUMP_IF_FALSE       58

  7          34 LOAD_FAST                2 (next)
             36 LOAD_FAST                1 (current)
             38 LOAD_FAST                2 (next)
             40 BINARY_ADD
             42 ROT_TWO
             44 STORE_FAST               1 (current)
             46 STORE_FAST               2 (next)

  8          48 LOAD_FAST                3 (counter)
             50 LOAD_CONST               3 (1)
             52 INPLACE_ADD
             54 STORE_FAST               3 (counter)
             56 JUMP_ABSOLUTE           26
        >>   58 POP_BLOCK

  9     >>   60 LOAD_FAST                1 (current)
             62 RETURN_VALUE
```


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

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

fibonacci.py
fibonacci.pyc

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

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

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

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

```
>>> fib.__code__.co_code
b'|\x00d\x01k\x00r\x0c|\x00S\x00d\x04\\\x02}\x01}\x02d\x
02}\x03x"|\x03|\x00k\x00r:|\x02|\x01|\x02\x17\x00\x02\x0
0}\x01}\x02|\x03d\x037\x00}\x03q\x1aW\x00|\x01S\x00'
```

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

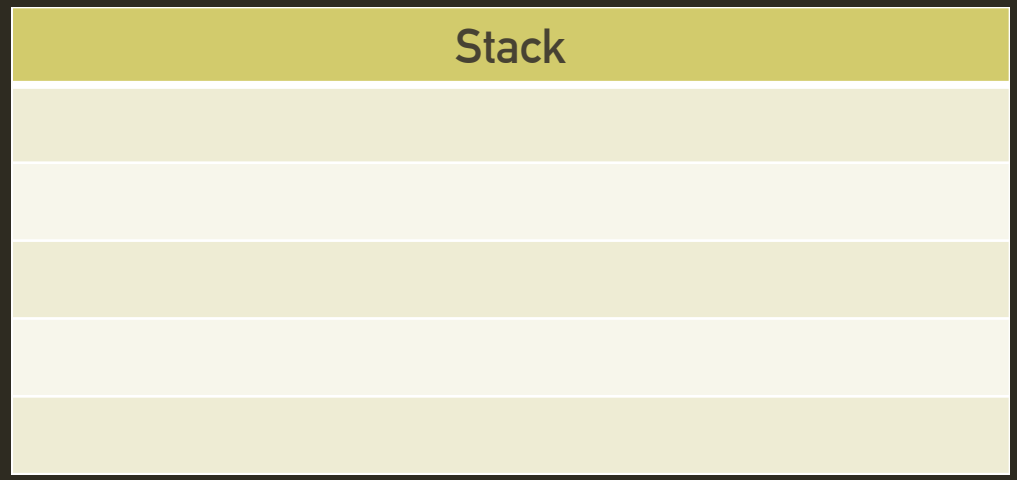
```
>>> import dis
>>> dis.opname[124]
'LOAD_FAST'
```

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

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

Executing a function

```
fib(8)

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



Executing a function

```
fib(8)

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

Stack
<function fib>

Executing a function

```
fib(8)

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

Stack
8
<function fib>

Executing a function

```
fib(8)

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

Stack
21

CPython actually uses two stacks: an evaluation stack or data stack (the one we've been looking at) 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.

```
dis.dis()
dis.distb()
dis.Bytecode
```

```
>>> 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
```

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

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

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

```
>>> 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
```

```
>>> 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
```

```
>>> 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
```

```
>>> 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
```

```
>>> 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
```

Python is always slower than C.

Local names are faster than global ones.

`LOAD_CONST` > `LOAD_FAST` > `LOAD_NAME` or `LOAD_GLOBAL`

Loops and blocks are expensive.

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

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

Look out for **`LOAD_ATTR`** and **`BINARY_SUBSCR`**

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