An introduction to Python bytecode

James Bennett
PyCon US 2018
2 0 LOAD_FAST 0 (n)
2 2 LOAD_CONST 1 (2)
4 4 COMPARE_OP 0 (<)
6 6 POP_JUMP_IF_FALSE 12

3 8 LOAD_FAST 0 (n)
10 10 RETURN_VALUE

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

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

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

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

8 >> 52 LOAD_FAST 1 (current)
54 54 RETURN_VALUE
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
You want to write human-friendly source code.

Your computer wants binary instructions ("machine code") for its 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.
def fib(n):
    if n < 2:
        return n
    current, next = 0, 1
    while n:
        current, next = next, current + next
        n -= 1
    return current
fibonacci.py
fibonacci.py.c
>>> 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')
>>> fib.__code__.co_names
()
>>> fib.__code__.co_code
b'|\x00d|\x01k|\x00r|\x0c|\x00S|\x00d|\x04|\x02|\x01|\x02|x1e|\x00r2|\x02|\x01|\x02|x17|x00|x02|x00|x01|x02|x00d|x038|x00|x00q|x16W|x00|x01S|x00'}
>>> ord('|')
124
>>> import dis
>>> dis.opname[124]
'LOAD_FAST'
2 \theta \text{ LOAD\_FAST } \theta \ (n)
>>> import dis
>>> dis.dis(fib)
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
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.
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.
Executing a function

```python
fib(8)
```

Evaluation stack:

<table>
<thead>
<tr>
<th></th>
<th>LOAD_GLOBAL 0 (fib)</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>LOAD_CONST 1 (8)</td>
</tr>
<tr>
<td>4</td>
<td>CALL_FUNCTION 1</td>
</tr>
</tbody>
</table>
Executing a function

fib(8)

<table>
<thead>
<tr>
<th></th>
<th>Instruction</th>
<th>Stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>LOAD_GLOBAL 0 (fib)</td>
<td>0</td>
</tr>
<tr>
<td>2</td>
<td>LOAD_CONST 1 (8)</td>
<td>1</td>
</tr>
<tr>
<td>4</td>
<td>CALL_FUNCTION 1</td>
<td>1</td>
</tr>
</tbody>
</table>
Executing a function

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

<table>
<thead>
<tr>
<th>Evaluation stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>8</td>
</tr>
<tr>
<td>&lt;function fib&gt;</td>
</tr>
</tbody>
</table>
Executing a function

fib(8)

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

<table>
<thead>
<tr>
<th>Evaluation stack</th>
</tr>
</thead>
<tbody>
<tr>
<td>21</td>
</tr>
</tbody>
</table>
https://docs.python.org/3/library/dis.html
>>> 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          2 LOAD_CONST            1 (0)
  -->
  4          4 BINARY_TRUE_DIVIDE
  6          6 PRINT_EXPR
  8          8 LOAD_CONST            2 (None)
 10         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);
        PUSH(value);
        FAST_DISPATCH();
    }
    # Many, many more bytecode instructions below...
What can we learn from bytecode?
def slow_week():
    SECONDS_PER_DAY = 86400
    return SECONDS_PER_DAY * 7

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


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

@ubernostrum

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