Exploring Python Bytecode

@AnjanaVakil

EuroPython 2016

Hi! I’m Anjana, and I’m a Pythoholic

OUTREACHY

mozilla

The **Recurse** Center

a Python puzzle...

```
# outside_fn.py
for i in range(10**8):
    i
```

```
# inside_fn.py
def run_loop():
    for i in range(10**8):
        i

run_loop()
```

```
$ time python3 outside_fn.py
real    0m9.185s
user    0m9.104s
sys     0m0.048s
```

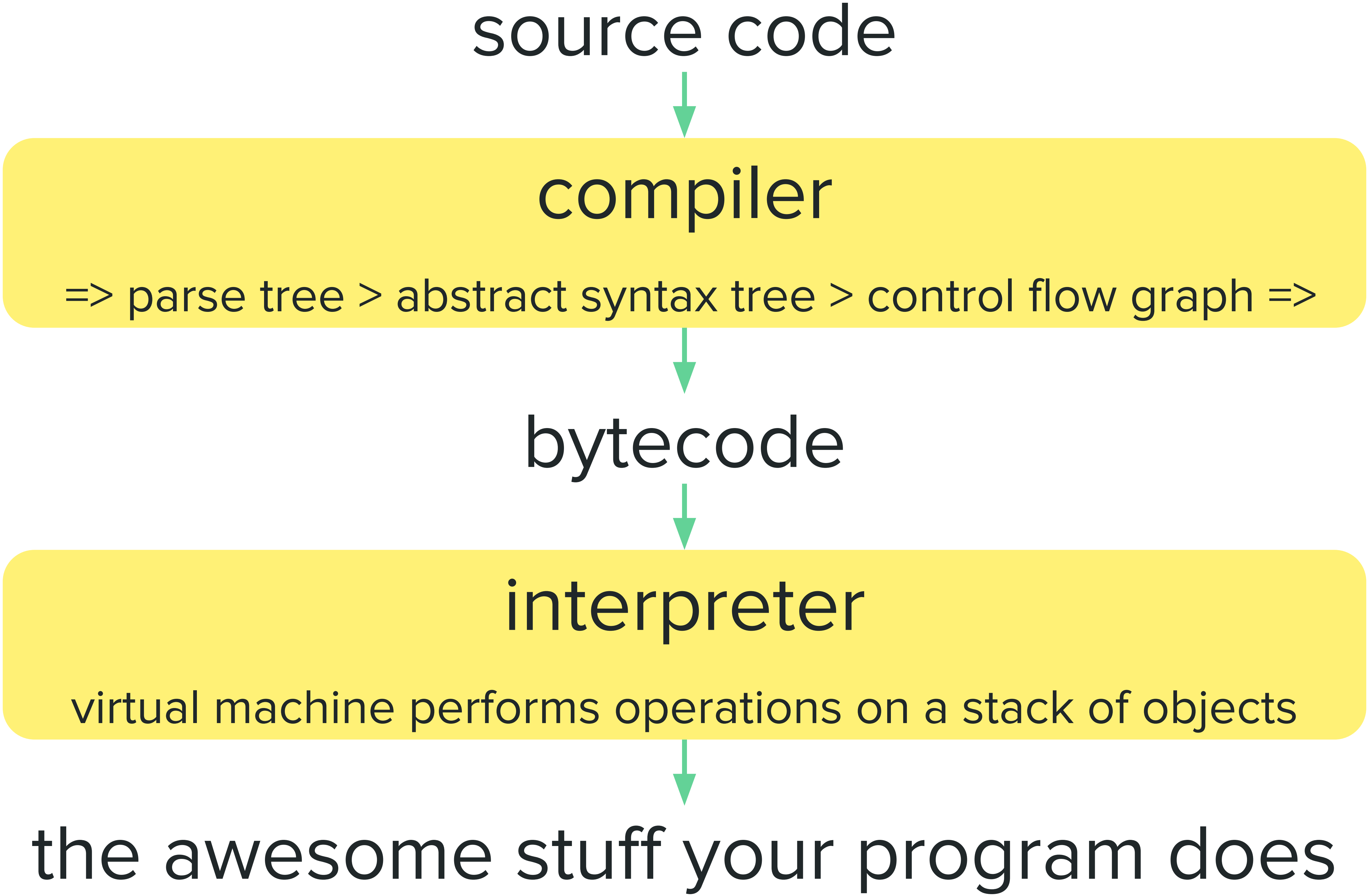
```
$ time python3 inside_fn.py
real    0m5.738s
user    0m5.634s
sys     0m0.055s
```

http://stackoverflow.com/questions/11241523/why-does-python-code-run-faster-in-a-function

What happens when you run Python code?

What happens when you run Python code?

**with CPython*



source code

compiler

=> parse tree > abstract syntax tree > control flow graph =>

bytecode

interpreter

virtual machine performs operations on a stack of objects

the awesome stuff your program does

What is bytecode?

an **intermediate representation** of your program

what the interpreter “sees”

when it runs your program

machine code for a **virtual machine**

(the interpreter)

a series of **instructions**
for **stack operations**

cached as **.pyc** files

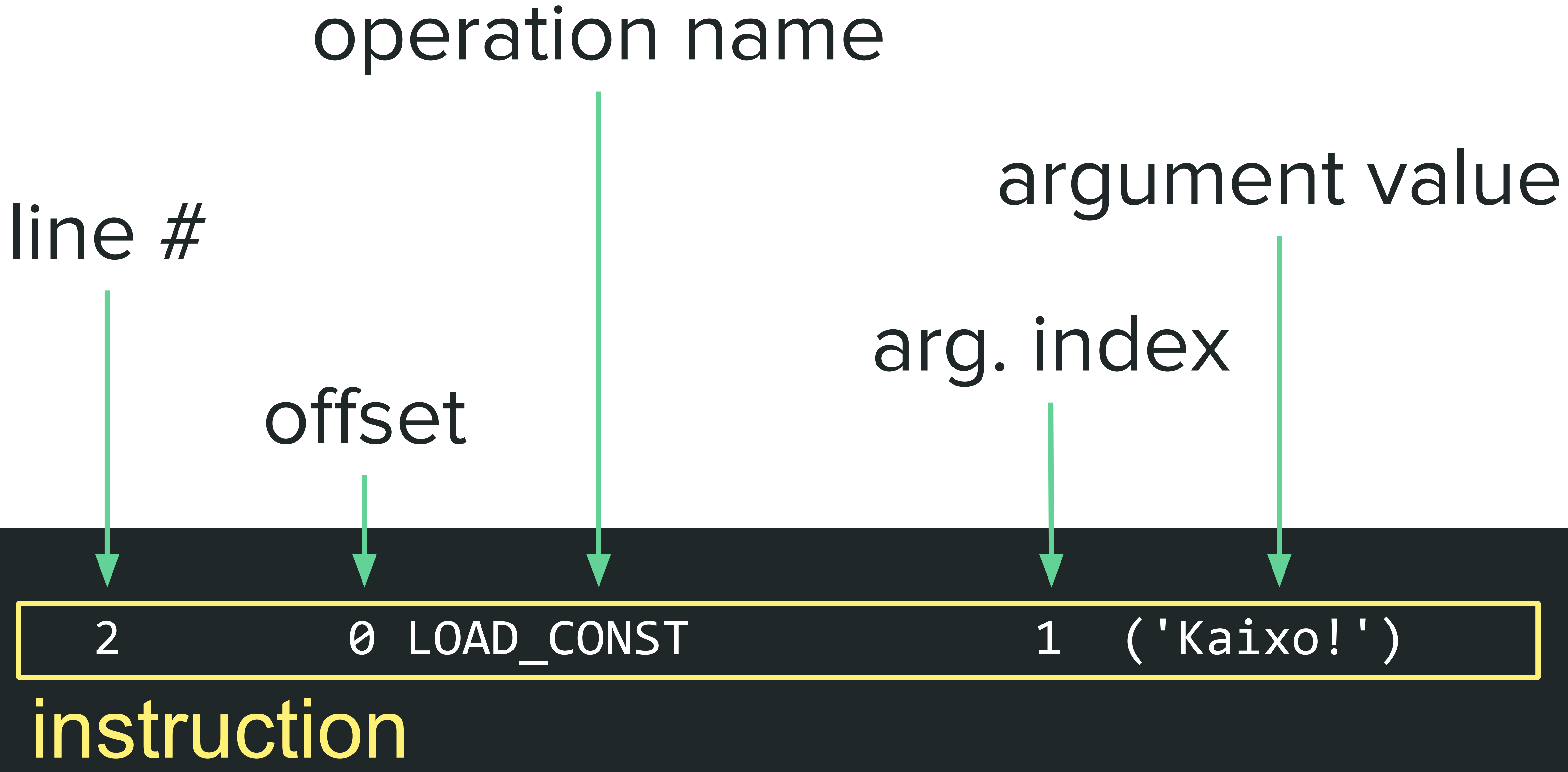
How can we read it?

dis: bytecode disassembler

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

```
>>> def hello():
...     return "Kaixo!"
...
>>> import dis
>>> dis.dis(hello)
  2           0 LOAD_CONST               1 ('Kaixo!')
              3 RETURN_VALUE
```

What does it all mean?



operation name

argument value

line #

arg. index

offset

```
  2           0 LOAD_CONST               1 ('Kaixo!')
```

instruction

sample operations

https://docs.python.org/library/dis.html#python-bytecode-instructions

`LOAD_CONST(`*c*`)` pushes *c* onto top of stack (TOS)

`BINARY_ADD` pops & adds top 2 items, result becomes TOS

`CALL_FUNCTION(`*a*`)` calls function with arguments from stack
a indicates # of positional & keyword args

```
>>> dis.opmap['BINARY_ADD']     # => 23
>>> dis.opname[23]              # => 'BINARY_ADD'
```

What can we `dis`?

functions

```
>>> def add(spam, eggs):
...     return spam + eggs
...
>>> dis.dis(add)
  2        0 LOAD_FAST             0 (spam)
           3 LOAD_FAST             1 (eggs)
           6 BINARY_ADD
           7 RETURN_VALUE
```

classes

```
>>> class Parrot:
...     def __init__(self):
...         self.kind = "Norwegian Blue"
...     def is_dead(self):
...         return True
...
>>>
```

classes

```
>>> dis.dis(Parrot)
Disassembly of __init__:
  3        0 LOAD_CONST              1 ('Norwegian Blue')
           3 LOAD_FAST               0 (self)
           6 STORE_ATTR              0 (kind)
           9 LOAD_CONST              0 (None)
           12 RETURN_VALUE
Disassembly of is_dead:
  5        0 LOAD_GLOBAL             0 (True)
           3 RETURN_VALUE
```

code strings (3.2+)

```
>>> dis.dis("spam, eggs = 'spam', 'eggs'")
  1           0 LOAD_CONST               3 (('spam', 'eggs'))
              3 UNPACK_SEQUENCE          2
              6 STORE_NAME               0 (spam)
              9 STORE_NAME               1 (eggs)
              12 LOAD_CONST              2 (None)
              15 RETURN_VALUE
```

modules

```
$ echo $'print("Ni!")' > knights.py
$ python3 -m dis knights.py
  1       0 LOAD_NAME            0 (print)
          3 LOAD_CONST           0 ('Ni!')
          6 CALL_FUNCTION        1 (1 positional, 0 keyword pair)
          9 POP_TOP
          10 LOAD_CONST          1 (None)
          13 RETURN_VALUE
```

modules (3.2+)

```
# knights.py
print("Ni!")
```

```
>>> dis.dis(open('knights.py').read())
    1     0 LOAD_NAME            0 (print)
          3 LOAD_CONST           0 ('Ni!')
          6 CALL_FUNCTION        1 (1 positional, 0 keyword pair)
          9 RETURN_VALUE
```

modules

```
# knights.py
print("Ni!")
def is_flesh_wound():
    return True
```

```
>>> import knights
Ni!
>>> dis.dis(knights)
Disassembly of is_flesh_wound:
  3       0 LOAD_CONST            1 (True)
          3 RETURN_VALUE
```

nothing! (last traceback)

```
>>> print(spam)
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'spam' is not defined
>>> dis.dis()
  1       0 LOAD_NAME            0 (print)
     -->  3 LOAD_NAME            1 (spam)
          6 CALL_FUNCTION        1 (1 positional, 0 keyword pair)
          9 PRINT_EXPR
         10 LOAD_CONST           0 (None)
         13 RETURN_VALUE
```

Why do we care?

debugging

```
>>> ham/eggs + ham/spam # => ZeroDivisionError: eggs or spam?
>>> dis.dis()
  1           0 LOAD_NAME                0 (ham)
              3 LOAD_NAME                1 (eggs)
              6 BINARY_TRUE_DIVIDE                  # OK here...
              7 LOAD_NAME                0 (ham)
             10 LOAD_NAME                2 (spam)
    -->      13 BINARY_TRUE_DIVIDE                  # error here!
             14 BINARY_ADD
             15 PRINT_EXPR
             16 LOAD_CONST               0 (None)
             19 RETURN_VALUE
```

solving puzzles!

```python
# outside_fn.py
for i in range(10**8):
    i
```

```
$ time python3 outside_fn.py
real    0m9.185s
user    0m9.104s
sys     0m0.048s
```

```python
# inside_fn.py
def run_loop():
    for i in range(10**8):
        i

run_loop()
```

```
$ time python3 inside_fn.py
real    0m5.738s
user    0m5.634s
sys     0m0.055s
```

http://stackoverflow.com/questions/11241523/why-does-python-code-run-faster-in-a-function

```
>>> outside = open('outside_fn.py').read()
>>> dis.dis(outside)
  2       0 SETUP_LOOP          24 (to 27)
          3 LOAD_NAME            0 (range)
          6 LOAD_CONST           3 (100000000)
          9 CALL_FUNCTION        1 (1 positional, 0 keyword pair)
         12 GET_ITER
     >>  13 FOR_ITER            10 (to 26)
         16 STORE_NAME           1 (i)

  3      19 LOAD_NAME            1 (i)
         22 POP_TOP
         23 JUMP_ABSOLUTE       13
     >>  26 POP_BLOCK
     >>  27 LOAD_CONST           2 (None)
         30 RETURN_VALUE
```

```
>>> from inside_fn import run_loop as inside
>>> dis.dis(inside)
  3       0 SETUP_LOOP          24 (to 27)
          3 LOAD_GLOBAL          0 (range)
          6 LOAD_CONST           3 (100000000)
          9 CALL_FUNCTION        1 (1 positional, 0 keyword pair)
         12 GET_ITER
     >>  13 FOR_ITER            10 (to 26)
         16 STORE_FAST           0 (i)

  4      19 LOAD_FAST            0 (i)
         22 POP_TOP
         23 JUMP_ABSOLUTE       13
     >>  26 POP_BLOCK
     >>  27 LOAD_CONST           0 (None)
         30 RETURN_VALUE
```

let’s investigate...

https://docs.python.org/3/library/dis.html#python-bytecode-instructions

`STORE_NAME`(*namei*)

Implements `name = TOS`. *namei* is the index of *name* in the attribute `co_names` of the code object.

`LOAD_NAME`(*namei*)

Pushes the value associated with `co_names[namei]` onto the stack.

`STORE_FAST`(*var_num*)

Stores TOS into the local `co_varnames[var_num]`.

`LOAD_FAST`(*var_num*)

Pushes a reference to the local `co_varnames[var_num]` onto the stack.

Want to dig deeper?

`ceval.c`: the heart of the beast

https://hg.python.org/cpython/file/tip/Python/ceval.c#l1358

A. Kaptur: "A 1500 (!!) line `switch` statement powers your Python"

http://akaptur.com/talks/

- `LOAD_FAST` (#l1368) is ~10 lines, involves fast locals lookup
- `LOAD_NAME` (#l2353) is ~50 lines, involves slow dict lookup
- prediction (#l1000) makes `FOR_ITER` + `STORE_FAST` even faster

More on SO: Why does Python code run faster in a function?

http://stackoverflow.com/questions/11241523/why-does-python-code-run-faster-in-a-function

Resources:

Python Module Of The Week: `dis`

https://pymotw.com/2/dis/

Allison Kaptur: Fun with `dis`

http://akaptur.com/blog/2013/08/14/python-bytecode-fun-with-dis/

Yaniv Aknin: Python Innards

https://tech.blog.aknin.name/category/my-projects/pythons-innards/

Python data model: code objects

https://docs.python.org/3/reference/datamodel.html#index-54

Eli Bendersky: Python ASTs

http://eli.thegreenplace.net/2009/11/28/python-internals-working-with-python-asts/

Thanks to:

Alice Duarte Scarpa, Andy Liang, Allison Kaptur, John J. Workman, Darius Bacon, Andrew Desharnais, John Hergenroeder, John Xia, Sher Minn Chong

...and the rest of the Recursers!

EuroPython

Outreachy

Thank you!

@AnjanaVakil

vakila.github.io