Hacking Python Byte Codes

24 September 2013

patrick.miller@gmail.com

https://github.com/patmiller

Background

- 17+ years of Python experience (v1.2+)
- Mostly high performance work
  - Livermore Labs (nuclear weapons design)
  - DE Shaw Research (computational chemistry)
  - JP Morgan IB (risk analysis)
  - Jump Trading (backend and simulation work)
- I have a lot of interest in the computational boundary between C/C++ and Python
  - Wrapping tools
  - Numeric expression compiler
  - Byte code translators
  - Byte code -> C++ converters

The need for speed…

- Python is used because it is "fast enough"
  - People efficient is not machine efficient
- Faster is better though
  - Better throughput…
  - Less power…
  - Fewer servers…
- How do we get there?
  - A C++ rewrite is expensive and rigid!
  - Efficient Python code can be hard to maintain!
- How far can we push the Python/C++ boundary?

So, we'll look at byte codes for optimization opportunities

- Looking for automatic or semi-automatic ones
- Mostly peephole fixes
- Optimization opportunities that the language cannot find without programmer guidance
  - i.e. ones that change the "letter" of the semantics, but retain their "spirit"
- We're looking for 5%, 10%, 15% improvements
- Want to stay in the "pure" Python world

Tradition!

- This idea of building specialized optimizers is not new to Python
- There is a long tradition of ad hoc optimizers based on local knowledge
- On Wall St, for example, there was a huge industry that specialized APL interpreters around known patterns
- Something similar is available for C/C++
  - http://rosecompiler.org/

A simple function… how does it really work?

```
>>> def f(x):
...         y = x+2
...         return abs(y)
```

Anatomy of a function

```
>>> def f(x):
...         y = x+2
...         return abs(y)
```

func_name	"f"
func_defaults	()
func_code	

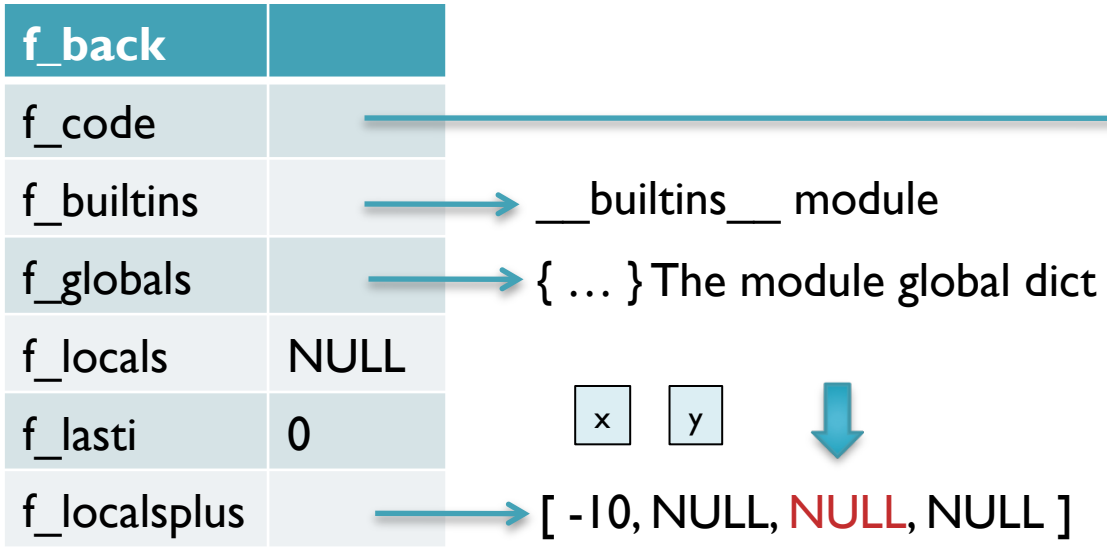
"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"'|\0…"	co_code
	co_cellvars
	co_freevars

```
 0 LOAD_FAST                0 (x)
 3 LOAD_CONST               1 (2)
 6 BINARY_ADD
 7 STORE_FAST               1 (y)

10 LOAD_GLOBAL              0 (abs)
13 LOAD_FAST                1 (y)
16 CALL_FUNCTION            1
19 RETURN_VALUE
```

Building a frame – f(-10)

```
>>> def f(x):
...         y = x+2
...         return abs(y)
```



"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"|\0…"	co_code
	co_cellvars
	co_freevars

```
 0 LOAD_FAST          0 (x)
 3 LOAD_CONST         1 (2)
 6 BINARY_ADD
 7 STORE_FAST         1 (y)

10 LOAD_GLOBAL        0 (abs)
13 LOAD_FAST          1 (y)
16 CALL_FUNCTION      1
19 RETURN_VALUE
```

Executing a frame – f(-10)

```
>>> def f(x):
...     y = x+2
...     return abs(y)
```

f_back	
f_code	
f_builtins	
f_globals	
f_locals	NULL
f_lasti	0
f_localsplus	

__builtins__ module

{ … } The module global dict

x y

[-10, NULL, NULL, NULL]

[-10, NULL, -10, NULL]

"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"'|\0…"	co_code
	co_cellvars
	co_freevars

```
 0 LOAD_FAST          0 (x)
 3 LOAD_CONST         1 (2)
 6 BINARY_ADD
 7 STORE_FAST         1 (y)

10 LOAD_GLOBAL        0 (abs)
13 LOAD_FAST          1 (y)
16 CALL_FUNCTION      1
19 RETURN_VALUE
```

Executing a frame – f(-10)

```
>>> def f(x):
...     y = x+2
...     return abs(y)
```

f_back	
f_code	
f_builtins	
f_globals	
f_locals	NULL
f_lasti	3
f_localsplus	

f_builtins → __builtins__ module

f_globals → { … } The module global dict

x y

f_localsplus → [-10, NULL, NULL, NULL]
[-10, NULL, -10, NULL]
[-10, NULL, -10, 2]

"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"'|\0…"	co_code
	co_cellvars
	co_freevars

```
0 LOAD_FAST          0 (x)
3 LOAD_CONST         1 (2)
6 BINARY_ADD
7 STORE_FAST         1 (y)

10 LOAD_GLOBAL       0 (abs)
13 LOAD_FAST         1 (y)
16 CALL_FUNCTION     1
19 RETURN_VALUE
```

Executing a frame – f(-10)

```
>>> def f(x):
...     y = x+2
...     return abs(y)
```

f_back	
f_code	
f_builtins	
f_globals	
f_locals	NULL
f_lasti	6
f_localsplus	

f_builtins → __builtins__ module

f_globals → { … } The module global dict

x y

f_localsplus →
[-10, NULL, NULL, NULL]
[-10, NULL, -10, NULL]
[-10, NULL, -10, 2]
[10, NULL, -8, NULL]

"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"|\0…"	co_code
	co_cellvars
	co_freevars

```
0 LOAD_FAST          0 (x)
3 LOAD_CONST         1 (2)
6 BINARY_ADD
7 STORE_FAST         1 (y)

10 LOAD_GLOBAL       0 (abs)
13 LOAD_FAST         1 (y)
16 CALL_FUNCTION     1
19 RETURN_VALUE
```

Executing a frame – f(-10)

```
>>> def f(x):
...     y = x+2
...     return abs(y)
...
```

f_back	
f_code	
f_builtins	
f_globals	
f_locals	NULL
f_lasti	7
f_localsplus	

f_builtins → __builtins__ module

f_globals → { … } The module global dict

x y

f_localsplus → [-10, NULL, NULL, NULL]
[-10, NULL, -10, NULL]
[-10, NULL, -10, 2]
[10, NULL, -8, NULL]
[10, -8, NULL, NULL]

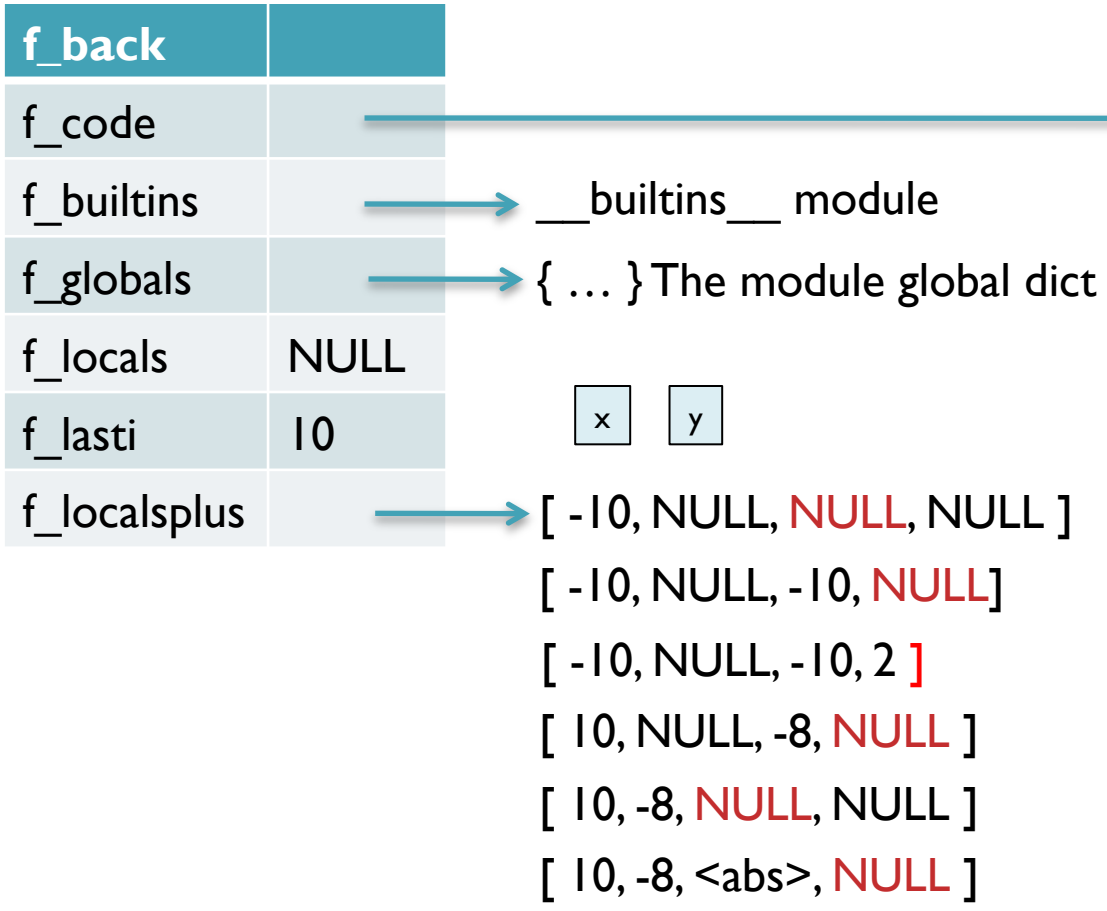
"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"|\0…"	co_code
	co_cellvars
	co_freevars

```
0 LOAD_FAST          0 (x)
3 LOAD_CONST         1 (2)
6 BINARY_ADD
7 STORE_FAST         1 (y)

10 LOAD_GLOBAL       0 (abs)
13 LOAD_FAST         1 (y)
16 CALL_FUNCTION     1
19 RETURN_VALUE
```

Executing a frame – f(-10)

```
>>> def f(x):
...     y = x+2
...     return abs(y)
```



"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"|\0…"	co_code
	co_cellvars
	co_freevars

```
0 LOAD_FAST       0 (x)
3 LOAD_CONST      1 (2)
6 BINARY_ADD
7 STORE_FAST      1 (y)

10 LOAD_GLOBAL    0 (abs)
13 LOAD_FAST      1 (y)
16 CALL_FUNCTION  1
19 RETURN_VALUE
```

Executing a frame – f(-10)

```
>>> def f(x):
...     y = x+2
...     return abs(y)
```

f_back	
f_code	
f_builtins	
f_globals	
f_locals	NULL
f_lasti	13
f_localsplus	

f_builtins → __builtins__ module

f_globals → { … } The module global dict

x y

f_localsplus →
[-10, NULL, NULL, NULL]
[-10, NULL, -10, NULL]
[-10, NULL, -10, 2]
[10, NULL, -8, NULL]
[10, -8, NULL, NULL]
[10, -8, <abs>, NULL]
[10, -8, <abs>, -8]

"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"|\0…"	co_code
	co_cellvars
	co_freevars

```
0 LOAD_FAST          0 (x)
3 LOAD_CONST         1 (2)
6 BINARY_ADD
7 STORE_FAST         1 (y)

10 LOAD_GLOBAL       0 (abs)
13 LOAD_FAST         1 (y)
16 CALL_FUNCTION     1
19 RETURN_VALUE
```

Executing a frame – f(-10)

```
>>> def f(x):
...     y = x+2
...     return abs(y)
```

f_back	
f_code	
f_builtins	
f_globals	
f_locals	NULL
f_lasti	16
f_localsplus	

f_builtins → __builtins__ module

f_globals → { … } The module global dict

x y

f_localsplus →
[-10, NULL, NULL, NULL]
[-10, NULL, -10, NULL]
[-10, NULL, -10, 2]
[10, NULL, -8, NULL]
[10, -8, NULL, NULL]
[10, -8, <abs>, NULL]
[10, -8, <abs>, -8]
[10, -8, 8, NULL]

"f"	co_name
"foo.py"	co_filename
7	co_firstlineno
0x43	co_flags
"…"	co_lnotab
("abs",)	co_names
("x","y",)	co_varnames
(None,2,)	co_consts
2	co_stacksize
2	co_nlocals
1	co_argcount
"'|\0…"	co_code
	co_cellvars
	co_freevars

```
0 LOAD_FAST          0 (x)
3 LOAD_CONST         1 (2)
6 BINARY_ADD
7 STORE_FAST         1 (y)

10 LOAD_GLOBAL       0 (abs)
13 LOAD_FAST         1 (y)
16 CALL_FUNCTION     1
19 RETURN_VALUE
```

A dilemma

- C++/Java programmers think of modules like namespaces…. ::std::cout
- But in Python, modules are really thinly disguised dictionaries*

```
00002 /* Module object implementation */
00003
00004 #include "Python.h"
00005 #include "structmember.h"
00006
00007 typedef struct {
00008         PyObject_HEAD
00009         PyObject *md_dict;
00010 } PyModuleObject;
```

* from moduleobject.c, © 2010, PSF

This has performance implications which engender maintenance/readability implications

```
import math
def dilemma(x):
    return math.sin(x)*math.sin(x) + math.cos(x)*math.cos(x)
```

```
from math import sin,cos
def dilemma(x):
    return sin(x)*sin(x) + cos(x)*cos(x)
```

```
def dilemma(x):
    from math import sin,cos
    return sin(x)*sin(x) + cos(x)*cos(x)
```

```
import math
def dilemma(x,sin=math.sin,cos=math.cos):
    return sin(x)*sin(x) + cos(x)*cos(x)
```

Why not write what we want and teach the compiler fix it!

```
import math
def dilemma(x):
        return math.sin(x)*math.sin(x) \
        + math.cos(x)*math.cos(x)
```

8 dictionary lookups!

```
>>> import dis
>>> dis.dis(dilemma)
  2           0 LOAD_GLOBAL              0 (math)
              3 LOAD_ATTR                1 (sin)
              6 LOAD_FAST                0 (x)
              9 CALL_FUNCTION            1
             12 LOAD_GLOBAL              0 (math)
             15 LOAD_ATTR                1 (sin)
             18 LOAD_FAST                0 (x)
             21 CALL_FUNCTION            1
             24 BINARY_MULTIPLY
             25 LOAD_GLOBAL              0 (math)
             28 LOAD_ATTR                2 (cos)
             31 LOAD_FAST                0 (x)
             34 CALL_FUNCTION            1
             37 LOAD_GLOBAL              0 (math)
             40 LOAD_ATTR                2 (cos)
             43 LOAD_FAST                0 (x)
             46 CALL_FUNCTION            1
             49 BINARY_MULTIPLY
             50 BINARY_ADD
             51 RETURN_VALUE
```

Our tools: dis & byteplay

- The "dis" module comes with Python
  - dis.dis(f) or dis.dis(code_object)
- Byteplay is a google code project
  - https://code.google.com/p/byteplay/
  - It provides primitives to more easily manipulate code objects

A byteplay code object breaks apart the byte codes into a simple list*

```
>>> from byteplay import Code
>>> code = Code.from_code(dilemma.func_code)
>>> print code.code
  2           1 LOAD_GLOBAL              math
              2 LOAD_ATTR                sin
              3 LOAD_FAST                x
              4 CALL_FUNCTION            1
              5 LOAD_GLOBAL              math
              6 LOAD_ATTR                sin
              7 LOAD_FAST                x
              8 CALL_FUNCTION            1
              9 BINARY_MULTIPLY
             10 LOAD_GLOBAL              math
             11 LOAD_ATTR                cos
             12 LOAD_FAST                x
             13 CALL_FUNCTION            1
             14 LOAD_GLOBAL              math
             15 LOAD_ATTR                cos
             16 LOAD_FAST                x
             17 CALL_FUNCTION            1
             18 BINARY_MULTIPLY
             19 BINARY_ADD
             20 RETURN_VALUE
```

*When printed, you don't see implied SetLineno and <Label> objects

We can just replace the parts we need and rebuild the function

```
>>> from byteplay import Code,LOAD_CONST
>>> code = Code.from_code(dilemma.func_code)
>>> code.code[1:3] = [  (LOAD_CONST,math.sin) ]
>>> code.code[4:6] = [  (LOAD_CONST,math.sin) ]
>>> code.code[8:10] = [  (LOAD_CONST,math.cos) ]
>>> code.code[11:13] = [  (LOAD_CONST,math.cos) ]
>>> print code.code
  2           1 LOAD_CONST           <built-in function sin>
              2 LOAD_FAST            x
              3 CALL_FUNCTION        1
              4 LOAD_CONST           <built-in function sin>
              5 LOAD_FAST            x
              6 CALL_FUNCTION        1
              7 BINARY_MULTIPLY
              8 LOAD_CONST           <built-in function cos>
              9 LOAD_FAST            x
             10 CALL_FUNCTION        1
             11 LOAD_CONST           <built-in function cos>
             12 LOAD_FAST            x
             13 CALL_FUNCTION        1
             14 BINARY_MULTIPLY
             15 BINARY_ADD
             16 RETURN_VALUE
>>> dilemma.func_code = code.to_code() # Mutable!
```

How did we do?

```
>>> def original(x):
...     return math.sin(x)*math.sin(x) + math.cos(x)*math.cos(x)
>>> with LittleTimer(1000) as T_original:
...     original(1.0)
>>> print T_original.rate,'original per second'
1666390.147 original per second

>>> from math import sin,cos
>>> def outer(x):
...     return sin(x)*sin(x) + cos(x)*cos(x)
>>> print T_outer.rate,'outer per second'
2192526.92107 outer per second

>>> from math import sin,cos
>>> def localsave(x):
...     sin_ = sin
...     cos_ = cos
...     return sin_(x)*sin_(x) + cos_(x)*cos_(x)
>>> print T_localsave.rate,'localsave per second'
1621300.34789 localsave per second

>>> def ugly(x,sin=math.sin,cos=math.cos):
...     return sin(x)*sin(x) + cos(x)*cos(x)
>>> print T_ugly.rate,'ugly per second'
2049000.48852 ugly per second

>>> with LittleTimer(1000) as T_patched:
...     dilemma(1.0)
>>> print T_patched.rate,'patched per second'
2427259.25926 patched per second
```

We can generalize this and put it into a @decorator

- Look for LOAD_GLOBAL foo
- If we can find foo in globals(), make a LOAD_CONST

- But what about the LOAD_ATTR?
  - Well, if we have a <const> value followed by one or more LOAD_ATTR, we chain them!

Here it is rolled up in a decorator

```
def cache_globals(f)
    from byteplay import Code, LOAD_CONST, LOAD_GLOBAL, LOAD_ATTR
    code = Code.from_code(co)

    missing = object()
    pc = 0
    while pc < len(code.code):
        op,arg = code.code[pc]
        if op == LOAD_GLOBAL:
            const = f.func_globals.get(arg,missing)
            if const is not missing:
                code.code[pc] = (LOAD_CONST,const)

        elif op == LOAD_ATTR:
            prev_op,prev_arg = code.code[pc-1]
            const = getattr(prev_arg,arg,missing)
            if const is not missing:
                code.code[pc-1:pc+1] = [(LOAD_CONST,const)]
                pc -= 1
        pc += 1

    f.func_code = code.to_code()
    return f
```

When you have a hammer…

- All the world looks like a nail
- What else can we do?

- LittleTimer is a byte code manipulator
- How about something to remove print statements from the function body?
- How about collapsing DEBUG conditionals?

LittleTimer

- It is a byte code manipulator to time very small actions with really low overhead
  - It grabs the "body" of a with statement
  - It replicates those byte codes n times
  - It slaps on some extra stuff to copy local vars
  - It adds a time.time() call begin and end
- This is just standard Python introspection, but on **code** instead of values

LittleTimer in action

```
>>> with LittleTimer(2) as T:
…         a = b+c
>>> dis.dis(T._LittleTimer__timerbody)
  1           0 LOAD_CONST               1 (10)
              3 STORE_FAST               0 (b)
              6 LOAD_CONST               2 (20)
              9 STORE_FAST               1 (c)
             12 LOAD_CONST               0 (None)
             15 STORE_FAST               2 (a)
             18 LOAD_CONST               3 (<built-in function time>)
             21 CALL_FUNCTION            0
             24 LOAD_FAST                0 (b)
             27 LOAD_FAST                1 (c)
             30 BINARY_ADD
             31 STORE_FAST               2 (a)
             34 LOAD_FAST                0 (b)
             37 LOAD_FAST                1 (c)
             40 BINARY_ADD
             41 STORE_FAST               2 (a)
             44 LOAD_CONST               3 (<built-in function time>)
             47 CALL_FUNCTION            0
             50 ROT_TWO
             51 BINARY_SUBTRACT
             52 LOAD_CONST               4 (2)
             55 BINARY_DIVIDE
             56 RETURN_VALUE
```

@unprint

- Do you hate to strip out print statements only to have to put them back in?
- What if we had a decorator @unprint that removed them?
  - s/@unprint/#@unprint/g puts them in or strips them out
  - Alternately, you could control with a global
- You can find the PRINT_ITEM codes and then work backwards to the start of an expression using byteplay.getse

The debug conundrum

- I had a buddy at Scripps Institute that carefully isolated all his debug code into a debug function. Then he profiled his code and realized that 30% of his runtime was spent skipping his calls to DEBUG()
- Function call overhead is non-trivial in Python. This pattern will kill performance:

```
import debugging
from debugging import DEBUG
debugging.DEBUGGING = True

def foo(x):
    DEBUG("in foo with",x)
    blah...
```

```
from debugging import DEBUG
DEBUGGING = False

def DEBUG(*args):
    if DEBUGGING:
        print ' '.join(
            map(str,args))
```

Debugging without the function call?

- OK, what if I just use an if DEBUG: ?
- It's still pretty bad (global reference and jump), but a little better.
- But! I can easily write a byteplay based decorator (@smartdebug) to remove them!

```
DEBUGGING = False

@smartdebug
def foo(x):
    if DEBUGGING:
        x = 10
        print "in foo with",x
    blah()
```

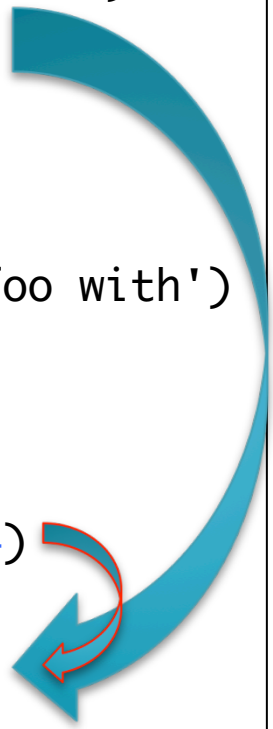
What are we looking for?

```
  2           0 LOAD_GLOBAL              0 (DEBUGGING)
              3 POP_JUMP_IF_FALSE       24

  3           6 LOAD_CONST               1 (10)
              9 STORE_FAST               0 (x)

  4          12 LOAD_CONST               2 ('in foo with')
             15 PRINT_ITEM
             16 LOAD_FAST                0 (x)
             19 PRINT_ITEM
             20 PRINT_NEWLINE
             21 JUMP_FORWARD             0 (to 24)

  5     >>   24 LOAD_GLOBAL              1 (blah)
             27 CALL_FUNCTION            0
             30 POP_TOP
             31 LOAD_CONST               0 (None)
             34 RETURN_VALUE
```



```
def foo(x):
    DEBUG("in foo with",x)
    blah...
```

@smartdebug

- Strategy: Look for the beginning of if DEBUG statement
  - `LOAD_GLOBAL DEBUG,JUMP_IF_TRUE`
  - `LOAD_GLOBAL DEBUG,JUMP_IF_FALSE`
- Find the "else"
  - `JUMP_FORWARD L1`
- Find the "endif"
- Look at value of DEBUG and pick the true part or else part as appropriate

Can we do that for DEBUG()?

- Sure!
- The process is like that of removing print statements, we look for all the expressions that are passed into the function. This "statically fixes" our debugging at function definition time.

Can I get this code?

- Sure! It is a bit rough but you can get it!
- https://github.com/patmiller/bytecode_toys.git
- I'll demo it in just a second.
- This also contains two handy routines that are almost always provide some speedup
  - make_local_functions_constant()
  - make_local_modules_constant()

Some thoughts to takeaway…

- Use the "dis" module to help understand what your code is doing
- Use LittleTimer to get a better idea of tiny efficiency gains
- Use byte code rewriters to get the code to do what you want while maintaining the original syntax
- You can be amazingly aggressive once you realize that byte codes are malliable

Keep in contact!

- I have a finance career blog on Wall Street Oasis
- I'm on LinkedIn too of course!
  - http://www.linkedin.com/pub/pat-miller/3/b19/b31/
- Old fashioned email works best
  - patrick.miller@gmail.com
- I can't really do much consulting work, but I'm always happy to answer interesting Python questions for free!

Demo!

- Cross your fingers and hope for the best here!