Unspeakably Evil Hacks in Service of Marginally-Improved Syntax

Compile-Time Metaprogramming in Python

https://github.com/ssanderson/pybay2016

Twitter: @scottbsanderson

About Me:

- API Design Lead at Quantopian
- Background in Mathematics and Philosophy
- **Twitter:** @scottbsanderson
- **GitHub:** ssanderson

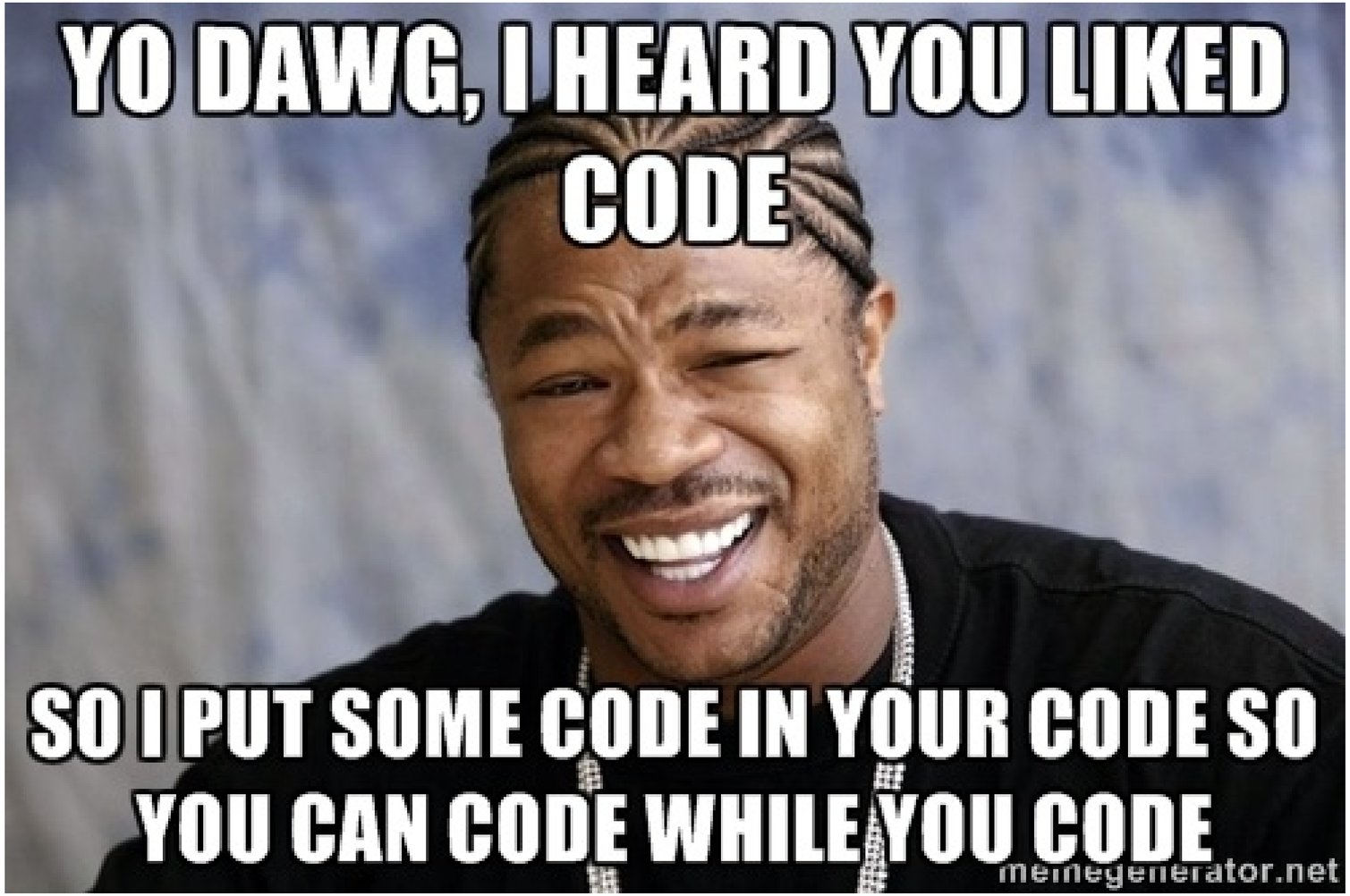
Outline

- "Standard" Metaprogramming Techniques
- Python Program Representations
- Custom File Encodings
- Import Hooks
- Bytecode Rewriting
- Conclusions

- Spicy Memes!

Metaprogramming

Metaprogramming is the writing of computer programs with the ability to treat programs as their data. It means that a program could be designed to read, generate, analyse or transform other programs, and even modify itself while running. - Wikipedia


YO DAWG, I HEARD YOU LIKED CODE
SO I PUT SOME CODE IN YOUR CODE SO YOU CAN CODE WHILE YOU CODE
memegenerator.net

Decorators

In [2]:

```
def noisy_add(a, b):
    print("add called with args: {args}".format(args=(a, b)))

    return a + b

...

def noisy_save(s):
    print("save called with args: {args}".format(args=(s,)))

    # /dev/null is web scale
    with open('/dev/null', 'w') as f:
        f.write(s)

noisy_add(1, 2)
noisy_save('Important Data')
```

```
add called with args: (1, 2)
save called with args: ('Important Data',)
```

In [3]:

```
from functools import wraps

def noisy(f):
    "A decorator that prints arguments to a function before calling it."
    name = f.__name__

    @wraps(f)
    def print_then_call_f(*args):
        print("{f} called with args: {args}".format(f=name, args=args))
        return f(*args)

    return print_then_call_f
```

In [4]:

```
@noisy
def add(a, b):
    return a + b

@noisy
def save(s):
    # Still web scale
    with open('/dev/null', 'w') as f:
        f.write(s)

add(1, 2)
save("Important Data")
```

```
add called with args: (1, 2)
save called with args: ('Important Data',)
```

Metaclasses

In [5]:

```
class Vector:
    "A 2-Dimensional vector."

    def __init__(self, x, y):
        self.x = x
        self.y = y

    def magnitude(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    def doubled(self):
        return type(self)(self.x * 2, self.y * 2)

v0 = Vector(1, 2)
print("Magnitude: %f" % v0.magnitude())
print("Doubled Magnitude: %f" % v0.doubled().magnitude())
```

```
Magnitude: 2.236068
Doubled Magnitude: 4.472136
```

In [6]:

```
class PropertyVector:
    "A 2-Dimensional vector, now with 100% fewer parentheses!"
    def __init__(self, x, y):
        self.x = x
        self.y = y

    @property
    def magnitude(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    @property
    def doubled(self):
        return type(self)(self.x * 2, self.y * 2)

v1 = PropertyVector(1, 2)
print("Magnitude: %f" % v1.magnitude)
print("Doubled Magnitude: %f" % v1.doubled.magnitude)
```

```
Magnitude: 2.236068
Doubled Magnitude: 4.472136
```

In [7]:

```
# Our metaclass will automatically convert anything with this signature
# into a property.
property_signature = inspect.FullArgSpec(
    args=['self'], varargs=None, varkw=None, defaults=None,
    kwonlyargs=[], kwonlydefaults=None, annotations={},
)

class AutoPropertyMeta(type):
    """Metaclass that wraps no-argument methods in properties."""

    def __new__(mcls, name, bases, clsdict):
        for name, class_attr in clsdict.items():
            try:
                signature = inspect.getfullargspec(class_attr)
            except TypeError:
                continue

            if signature == property_signature:
                print("Wrapping %s in a property." % name)
                clsdict[name] = property(class_attr)

        return super().__new__(mcls, name, bases, clsdict)
```

In [8]:

```
class AutoPropertyVector(metaclass=AutoPropertyMeta):
    "A 2-Dimensional vector, now with 100% less @property calls!"
    def __init__(self, x, y):
        self.x = x
        self.y = y

    def magnitude(self):
        return math.sqrt(self.x ** 2 + self.y ** 2)

    def doubled(self):
        return type(self)(self.x * 2, self.y * 2)

v2 = AutoPropertyVector(1, 2)
print("")
print("Magnitude: %f" % v2.magnitude)
print("Doubled Magnitude: %f" % v2.doubled.magnitude)
```

```
Wrapping magnitude in a property.
Wrapping doubled in a property.

Magnitude: 2.236068
Doubled Magnitude: 4.472136
```

exec

"The Swiss Army Knife of Metaprogramming"

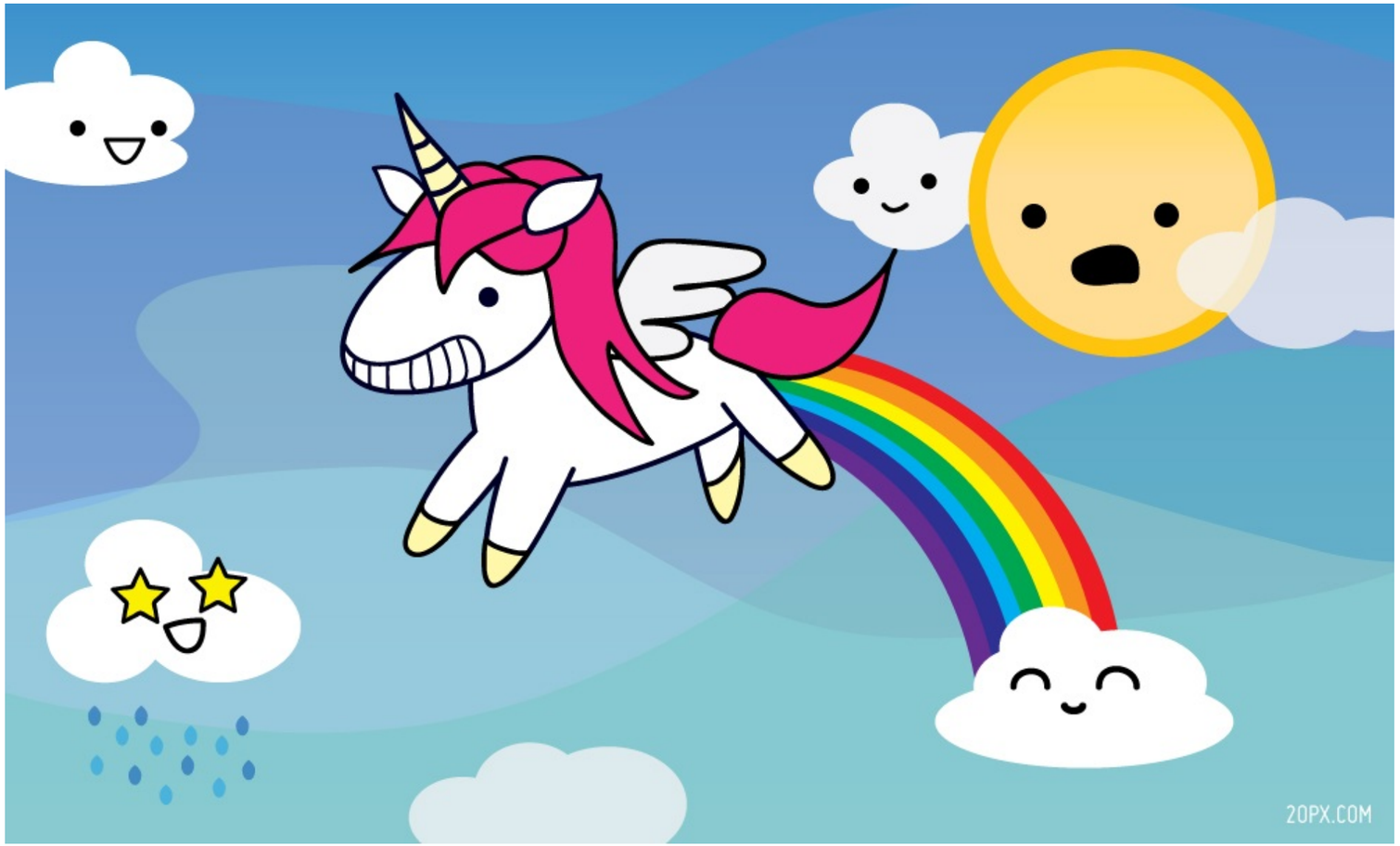
In [9]:

```
from pybay2016.simple_namedtuple import simple_namedtuple

Point = simple_namedtuple('Point', ['x', 'y', 'z'])
p = Point(x=1, y=2, z=3)

print("p.x is {p.x}".format(p=p))
print("p[1] is {p[1]}".format(p=p))
```

```
p.x is 1
p[1] is 2
```

Review

- Decorators allow us to naturally express modifications to existing classes and functions.
- Metaclasses allow us to customize class creation.
- `exec` allows us to use string-manipulation tools for program manipulation.


20PX.COM

That's all great but...

- Abstractions often incur runtime overhead.
- Certain operations can't be overloaded (e.g. `is` and `not`, or catching exceptions).
- No support for syntactic extensions.
  - Can't add new syntax.
  - Often can't repurpose existing syntax.

- Doesn't quite feel *evil* enough...

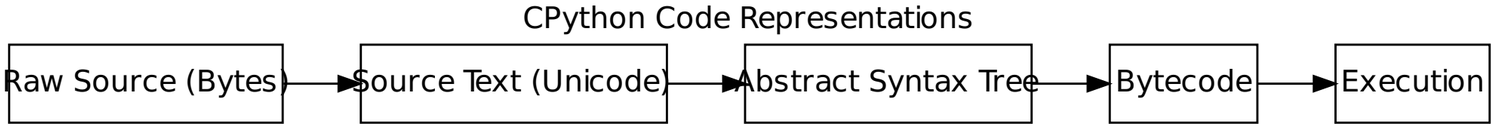

WE NEED TO GO
DEEPER

Let’s look at a complicated function...

In [10]:
```
def addtwo(a):
    return a + 2

addtwo(1)
```

Out[10]:
```
3
```

What Happened When I Hit Enter?


CPython Code Representations
Raw Source (Bytes)
Source Text (Unicode)
Abstract Syntax Tree
Bytecode
Execution

In [11]:
```
raw_source = b"""\
def addtwo(a):
    return a + 2

addtwo(1)
"""

raw_source
list(raw_source)[:10]
```

Out[11]:
```
[100, 101, 102, 32, 97, 100, 100, 116, 119, 111]
```

In [12]:

```
# Bytes to Text
import codecs

decoded_source = codecs.getdecoder('utf-8')(raw_source)[0]

print(decoded_source)
```

```
def addtwo(a):
    return a + 2

addtwo(1)
```

In [13]:

```
# Text to AST
import ast
syntax_tree = ast.parse(decoded_source)

body = syntax_tree.body
show_ast(body[1])
```

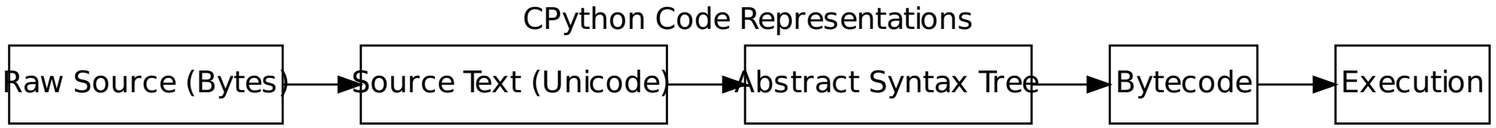
```
Expr(
  value=Call(
    func=Name(id='addtwo', ctx=Load()),
    args=[
      Num(1),
    ],
    keywords=[],
    starargs=None,
    kwargs=None,
  ),
)
```

In [14]:

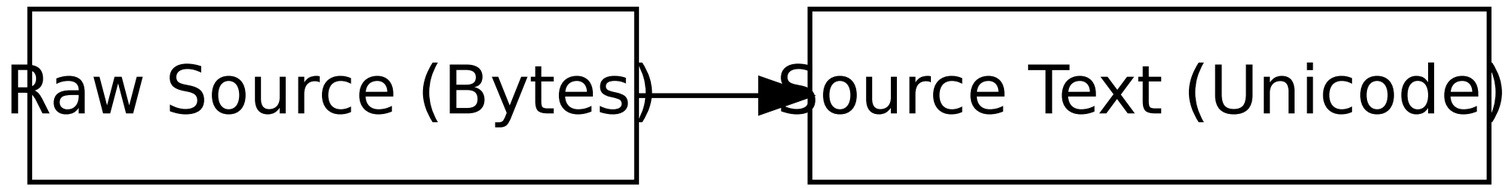
```
# AST -> Bytecode
code = compile(syntax_tree, 'pybay2016', 'exec')
show_disassembly(code)
```

```
<module>
--------
  1           0 LOAD_CONST               0 (<code object addtwo at 0x7fe06009230
0, file "pybay2016", line 1>)
              3 LOAD_CONST               1 ('addtwo')
              6 MAKE_FUNCTION            0
              9 STORE_NAME               0 (addtwo)

  4          12 LOAD_NAME                0 (addtwo)
             15 LOAD_CONST               2 (1)
             18 CALL_FUNCTION            1 (1 positional, 0 keyword pair)
             21 POP_TOP
             22 LOAD_CONST               3 (None)
             25 RETURN_VALUE

<module>.addtwo
---------------
  2           0 LOAD_FAST                0 (a)
              3 LOAD_CONST               1 (2)
              6 BINARY_ADD
              7 RETURN_VALUE
```

What can I muck with?



Custom Source Encodings



In [15]:
```
!cat ../pybay2016/rot13.py
```

```
# encoding: pybay2016-rot13

qrs uryyb():
    cevag("Uryyb Ebgngrq Jbeyq!")
```

Python doesn't know about our encoding...

In [16]:
```
from pybay2016.rot13 import hello

hello()
```

```
  File "<string>", line unknown
SyntaxError: unknown encoding for '/home/ssanderson/projects/pybay2016/pybay2016
/rot13.py': pybay2016-rot13
```

...until we register with the `codecs` module.

In [17]:
```
from codecs import register
from pybay2016.encoding import search_function

register(search_function)
```

In [18]:
```
from pybay2016.rot13 import hello

hello()
```

```
Hello Rotated World!
```

What is this actually useful for?

In [19]:
```
!cat ../pybay2016/pyxl.py
```

```
# encoding: pyxl

import pyxl.html as html


def hello_html():
    x = "Hello World!"
    return <html>
             <body>{x}</body>
           </html>
```

In [20]:
```
import pyxl.codec.register # Activates the pyxl encoding

from pybay2016.pyxl import hello_html

hello_html()
```

Out[20]:
```
<pyxl.html.x_html at 0x7fe068ec6da0>
```

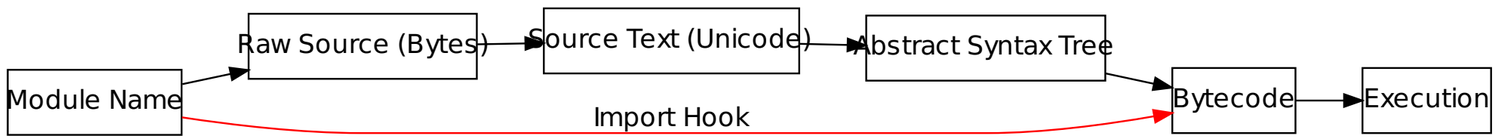
In [21]:
```
str(hello_html())
```

Out[21]:
```
'<html><body>Hello World!</body></html>'
```

Custom Encoding Summary

- Encodings registered globally with the `codecs` module.
- Opt-in on a **per-file** basis.
- Only one encoding per file.
- Files must end in `.py`.
- Operates semantically on bytes <-> text layer.

Import Hooks



Hy - Lisp Embedded in Python

In [22]:
```
! cat ../pybay2016/hy_example.hy
```

```
(defn hyfact [n]
  "Lisp in Python!"
  (defn fact-impl [n acc]
    (if (<= n 1)
        acc
      (fact-impl (- n 1) (* acc n))))
  (fact-impl n 1))
```

In [23]:

```
# Python doesn't know about .hy files by default.
from pybay2016.hy_example import hyfact

hyfact(5)
```

```
---------------------------------------------------------------------------
ImportError                               Traceback (most recent call last)
<ipython-input-23-948adc87f421> in <module>()
      1 # Python doesn't know about .hy files by default.
----> 2 from pybay2016.hy_example import hyfact
      3
      4 hyfact(5)

ImportError: No module named 'pybay2016.hy_example'
```

In [24]:

```
# But importing hy registers a MetaImporter that knows about .hy files.
print("Before:")
pprint.pprint(sys.meta_path[0])

import hy

print("After:")
pprint.pprint(sys.meta_path[0])
```

```
Before:
<class '_frozen_importlib.BuiltinImporter'>
After:
<hy.importer.MetaImporter object at 0x7fe06011ccf8>
```

In [25]:
```
from pybay2016.hy_example import hyfact

hyfact(5)
```

Out[25]:
```
120
```

Cython - Pseudo-Python Compiled to C

In [26]:
```
!cat ../pybay2016/cython_example.pyx
```

```
cpdef cyfact(int n):
    cdef int acc = 1
    cdef int i
    for i in range(1, n + 1):
        acc *= i
    return acc
```

In [27]:
```
import pyximport
pyximport.install()  # Installs a Cython meta-importer.

from pybay2016.cython_example import cyfact

print("cyfact is a %s" % type(cyfact))
cyfact(5)
```

```
cyfact is a <class 'builtin_function_or_method'>
```

Out[27]:
```
120
```

In [28]:

```
print("Python Factorial:")
%timeit hyfact(25)

print("\nCython Factorial:")
%timeit cyfact(25)
```

```
Python Factorial:
100000 loops, best of 3: 3.23 µs per loop

Cython Factorial:
10000000 loops, best of 3: 52.1 ns per loop
```

Import Hook Summary

- Registered Globally on `sys.meta_path`.
- Loader gets to choose how a file is imported.
- No restrictions on file structure.
    - Doesn't even have correspond to a filesystem entry...

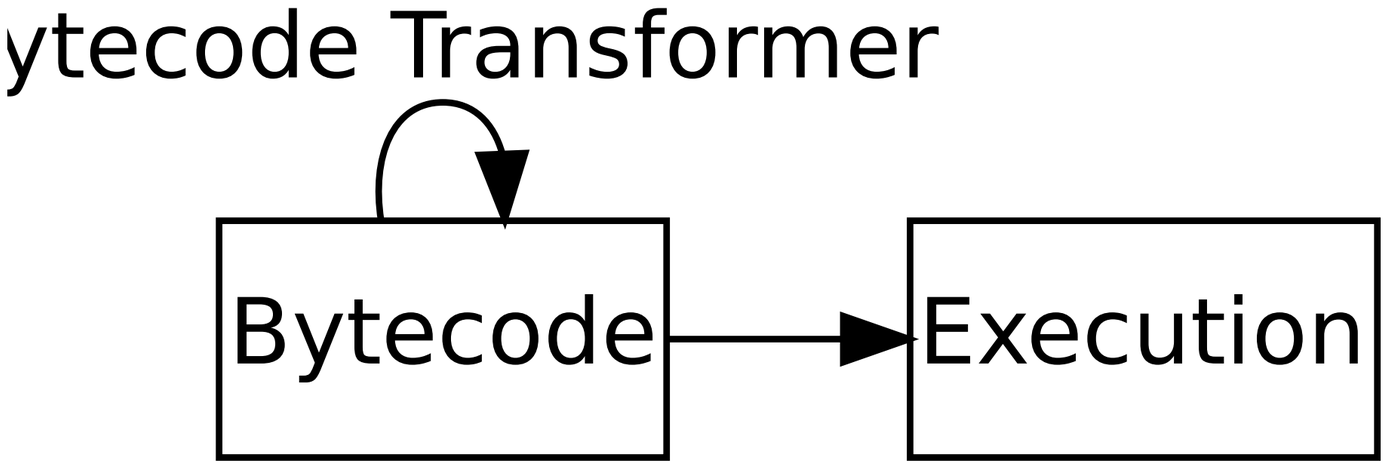
Problem:

Both import hooks and custom encodings rely on some **external** piece of code having been run before a hooked module can be executed.

This makes it hard to ensure that transformations are used correctly/reliably by users.

What if we just rewrite the code we already have...?

Bytecode Manipulation



In [29]:
```
addcode = addtwo.__code__
list(addcode.co_code)
```

Out[29]:
```
[124, 0, 0, 100, 1, 0, 23, 83]
```

In [30]:
```
from pybay2016.bytecode import code_attrs

code_attrs(addcode)
```

Out[30]:
```
{'co_argcount': 1,
 'co_cellvars': (),
 'co_code': b'|\x00\x00d\x01\x00\x17S',
 'co_consts': (None, 2),
 'co_filename': '<ipython-input-10-ba723be474f5>',
 'co_firstlineno': 1,
 'co_flags': 67,
 'co_freevars': (),
 'co_kwonlyargcount': 0,
 'co_lnotab': b'\x00\x01',
 'co_name': 'addtwo',
 'co_names': (),
 'co_nlocals': 1,
 'co_stacksize': 2,
 'co_varnames': ('a',)}
```

In [31]:
```
import dis

print("Raw Bytes: %s" % list(addcode.co_code))
print("\nDisassembly:\n")
dis.dis(addcode)
```

```
Raw Bytes: [124, 0, 0, 100, 1, 0, 23, 83]

Disassembly:

  2           0 LOAD_FAST                0 (a)
              3 LOAD_CONST               1 (2)
              6 BINARY_ADD
              7 RETURN_VALUE
```

Addition is so 2015...

In [32]:

```
def replace_all(l, old, new):
    "Replace all instances of `old` in `l` with `new`"
    out = []
    for elem in l:
        if elem == old: out.append(new)
        else: out.append(elem)
    return out

addbytes = addcode.co_code
mulbytes = bytes(replace_all(list(addbytes), 23, 20))

print("Old Disassembly:"); dis.dis(addbytes)
print("\nNew Disassembly:"); dis.dis(mulbytes)
```

```
Old Disassembly:
          0 LOAD_FAST                0 (0)
          3 LOAD_CONST               1 (1)
          6 BINARY_ADD
          7 RETURN_VALUE

New Disassembly:
          0 LOAD_FAST                0 (0)
          3 LOAD_CONST               1 (1)
          6 BINARY_MULTIPLY
          7 RETURN_VALUE
```

Just overwriting the code won't work...

In [33]:
```
add.__code__.co_code = mulbytes
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
<ipython-input-33-41941fd77925> in <module>()
----> 1 add.__code__.co_code = mulbytes

AttributeError: readonly attribute
```

...but copying everything else will!

In [34]:
```
from types import CodeType

mulcode = CodeType(
    addcode.co_argcount,
    addcode.co_kwonlyargcount,
    addcode.co_nlocals,
    addcode.co_stacksize,
    addcode.co_flags,
    mulbytes,   # Use our new bytecode.
    addcode.co_consts,
    addcode.co_names,
    addcode.co_varnames,
    addcode.co_filename,
    'multwo',  # Use a new name.
    addcode.co_firstlineno,
    addcode.co_lnotab,
    addcode.co_freevars,
    addcode.co_cellvars,
)

mulcode
```

Out[34]:
```
<code object multwo at 0x7fe060092db0, file "<ipython-input-10-ba723be474f5>", l
ine 1>
```

We can rebuild a modified function the same way.

In [35]:
```
from types import FunctionType

multwo = FunctionType(
    mulcode,   # Use new bytecode.
    addtwo.__globals__,
    'multwo',  # Use new __name__.
    addtwo.__defaults__,
    addtwo.__closure__,
)
multwo
```

Out[35]:
```
<function __main__.multwo>
```

In [36]:
```
multwo(5)
```

Out[36]:
```
10
```

Pattern for Bytecode Transformers

- Start with an existing function.
- Extract its `__code__`.
- Apply some transformation.
- Contruct new `code` by copying everything not changed.
- Construct new function from new code object.

In [37]:

```
from codetransformer import CodeTransformer, pattern
from codetransformer.instructions import *

class ruby_strings(CodeTransformer):

    @pattern(LOAD_CONST)
    def _format_bytes(self, instr):
        yield instr
        if not isinstance(instr.arg, bytes):
            return

        # Equivalent to:
        # s.decode('utf-8').format(**locals())
        yield LOAD_ATTR('decode')
        yield LOAD_CONST('utf-8')
        yield CALL_FUNCTION(1)
        yield LOAD_ATTR('format')
        yield LOAD_CONST(locals)
        yield CALL_FUNCTION(0)
        yield CALL_FUNCTION_KW()
```

In [38]:
```
@ruby_strings()
def example(a, b, c):
    return b"a is {a}, b is {b}, c is {c!r}"

example(1, 2, 'foo')
```

Out[38]:
```
"a is 1, b is 2, c is 'foo'"
```

More Examples: Overloaded Exceptions

In [39]:
```
from codetransformer.transformers.exc_patterns import \
    pattern_matched_exceptions


@pattern_matched_exceptions()
def foo():
    try:
        raise ValueError('bar')
    except ValueError('buzz'):
        return 'buzz'
    except ValueError('bar'):
        return 'bar'

foo()
```

Out[39]:
```
'bar'
```

More Examples : Overloaded Literals

In [40]:
```
from codetransformer.transformers.literals import ordereddict_literals

@ordereddict_literals
def make_dictionary(a, b):
    return {a: 1, b: 2}

make_dictionary('a', 'b')
```

Out[40]:
```
OrderedDict([('a', 1), ('b', 2)])
```

Bytecode Transformer Summary

- Doesn't require external setup.
- Can be applied on a per-function basis like any other decorator.
- Relies on CPython implementation details:
  - Only works in CPython (no PyPy).
  - Significant changes between minor versions.
- Bugs can cause segfaults.

Conclusions

- Python's standard metaprogramming tools are pretty awesome.
- Import Hooks and Custom Encodings allow us to extend syntax if we're willing to depend on global setup.
- Bytecode transformers give is isolated, composable transformations, at the cost of cross-platform compatibility.

Thanks!

- **Talk Content:** https://github.com/ssanderson/pybay2016
- **Twitter:** @scottbsanderson
- **GitHub:** ssanderson