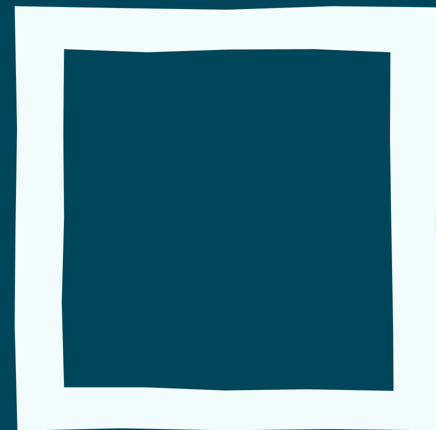
5 Years of Bad Ideas

Armin Ronacher

Introduction



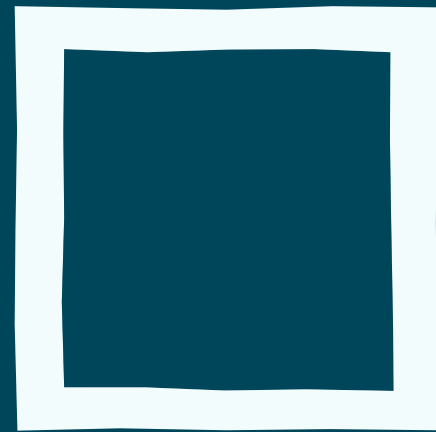
What defines Python?

Core Values

- Beautiful Code
- Low-clutter Syntax
- Everything is a first class object, functions included

But Also

- Ability to eval() and compile()
- Access to interpreter internals
- A community that can monkey patch, but would not do it unless necessary.



Good or Bad Magic

About Magic

"And honestly, I only find this shit in web frameworks. WTF is up with Python people going ballistic when a web app uses magic? Bullshit."

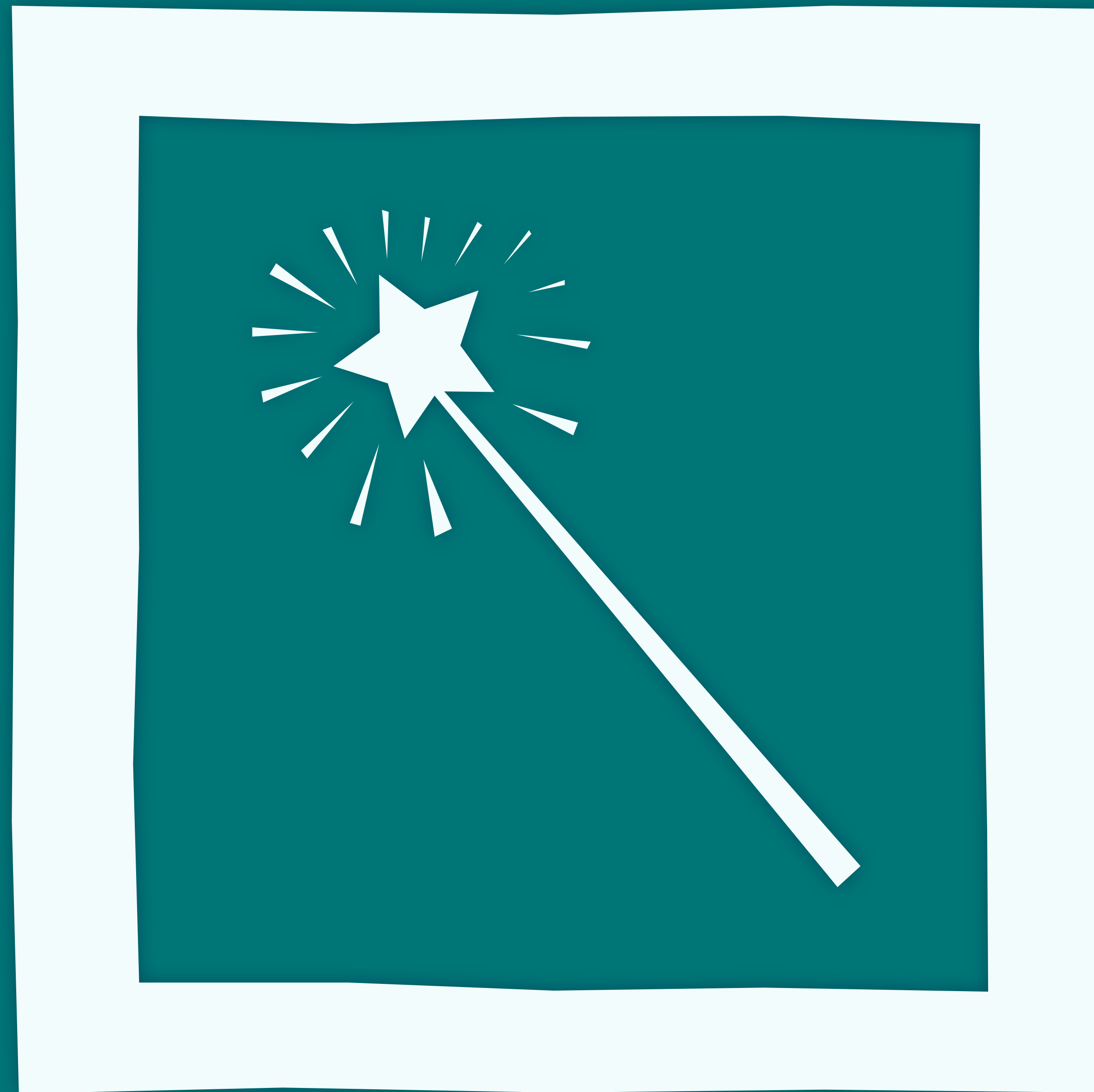
— Zed Shaw about Magic

Magic as Enabler

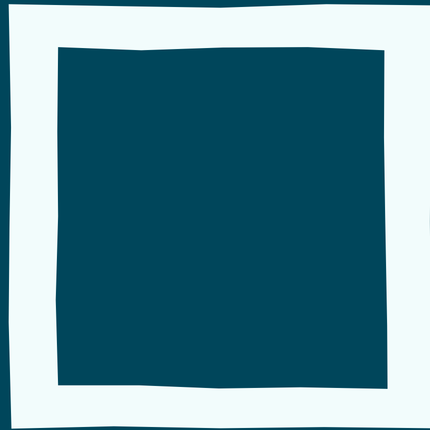
- Certain things require magic (Having an interactive traceback in your browser when a web application fails)
- Stepping through code with a debugger
- Properly post-processed traceback frames for generated code.
- Generated code in general.

Magic with Fallbacks

- Magic is fine for as long as there is a fallback.
- Magic becomes a burden if it's the only way to achieve something in production code.



Common Magic



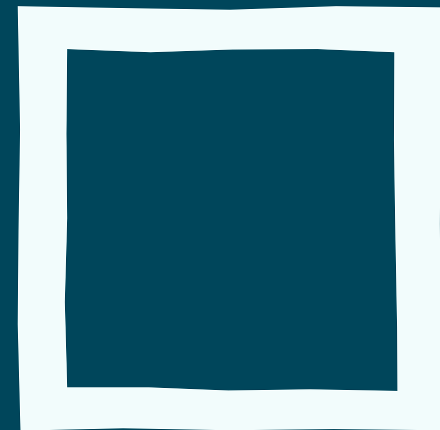
sys._getframe

```python
import sys


def caller_lineno():
    return sys._getframe(1).f_lineno

def caller_filename():
    return sys._getframe(1).f_code.co_filename
```

```python
class User(db.Model):
    implements(IRenderable)

    ...

    def render(self, renderer):
        renderer.text(self.username)
        renderer.img(src=self.icon_avatar_url,
                     class_='small-avatar')
```

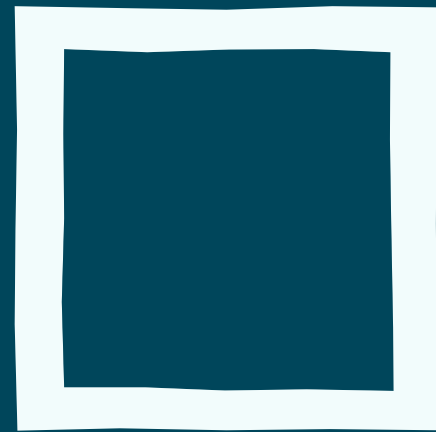
```python
import sys


def implements(*interfaces):
    cls_scope = sys._getframe(1).f_locals
    metacls = cls_scope.get('__metaclass__')
    new_metacls = magic_metaclass_factory(
       interfaces, metacls)
    cls_scope['__metaclass__'] = new_metaclas
```

```python
import sys


def find_request():
    frm = sys._getframe(1)
    while frm is not None:
        if 'request' in frm.f_locals and \
           hasattr(frm.f_locals['request'], 'META'):
            return frm.f_locals['request']
        frm = frm.f_back
```



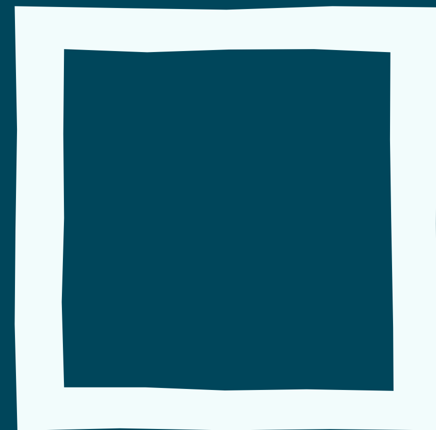
Metaclasses

```python
class AwesomeRegistry(type):
    registry = {}

    def __new__(cls, name, bases, d):
        rv = type.__new__(cls, name, bases, d)
        registry[name] = rv
        return rv

    def __getitem__(cls, name):
        return cls.registry[name]


class AwesomeBase(object):
    __metaclass__ = registry
```



exec / compile

```python
def func_from_file(filename, function):
    namespace = {}
    execfile(filename, namespace)
    return namespace[function]

func = func_from_file('hello.cfg', 'main')
func()
```



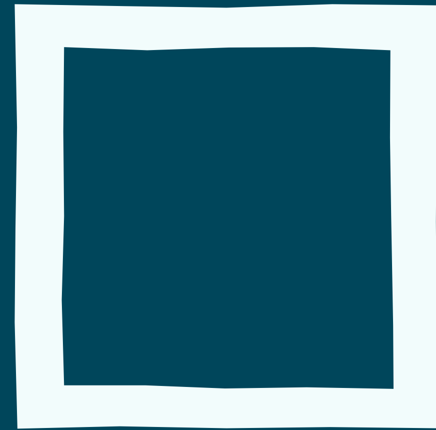
AST Hacks

```
>>> import ast
>>> x = ast.parse('1 + 2', mode='eval')
>>> x.body.op = ast.Sub()
>>> eval(compile(x, '<string>', 'eval'))
-1
```

Things you can do

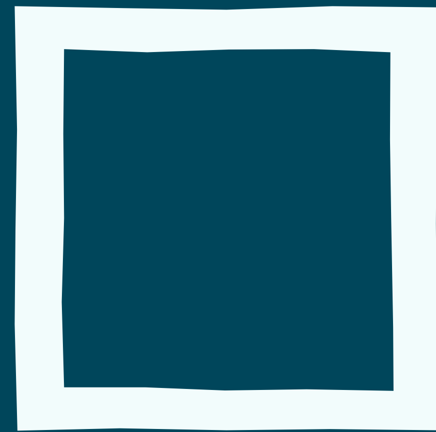
- Coupled with an import hook, rewrite assert statements to method calls
- Implement macros
- Use it for code generation

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


@macro
def times(i):
    for __x in range(i):
        __body__


def doing_something():
    a = add(1, 2)
    with times(10):
        print 'iterating ...'
```



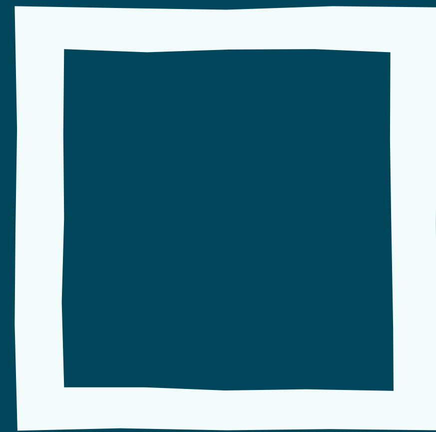
Import Hooks

```
>>> from githubimporter import GitHubImporter
>>> sys.path_hooks.append(GitHubImporter)
>>> import sys
>>> sys.path.append('github://mitsuhiko/jinja2')
>>> import jinja2
>>> jinja2.__file__
'github://mitsuhiko/jinja2/__init__.py'
>>> from jinja2 import Template
>>> t = Template('Hello from {{ hell }}!')
>>> t.render(hell='Import Hook Hell')
u'Hello from Import Hook Hell'
```



Monkeypatching

```python
from a_horrible_library import SomeClass

original_init = SomeClass.__init__


def new__init__(self, *args, **kwargs):
    original__init__(self, *args, **kwargs)
    self.something_else = Thing()

SomeClass.__init__ = new__init__
```

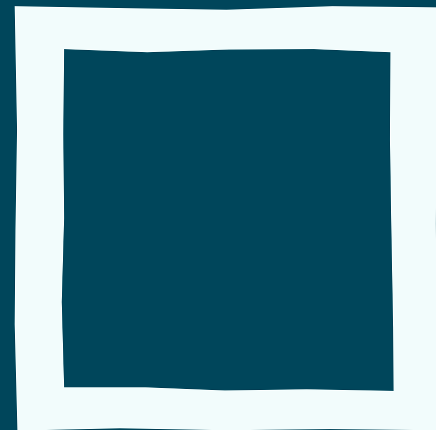


__builtin__ patching

```python
de = Translations.load(['de_DE', 'de', 'en'])

import __builtin__
__builtin__._ = de.ugettext
```

```python
import __builtin__
__builtin__.__import__ = my_fancy_import
```

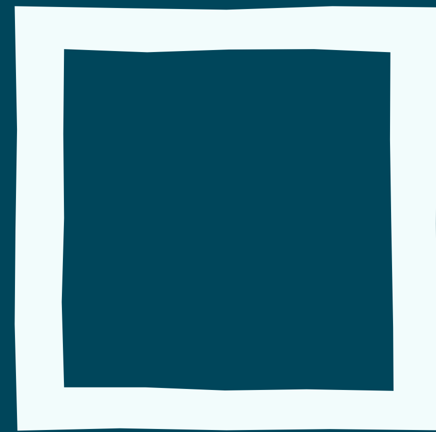
Uncommon Magic



sys.modules behavior

```python
import sys
from os.path import join, dirname
from types import ModuleType


class MagicModule(ModuleType):

    @property
    def git_hash(self):
        fn = join(dirname(__file__),
                  '.git/refs/heads/master')
        with open(fn) as f:
            return f.read().strip()


old_mod = sys.modules[__name__]
sys.modules[__name__] = mod = MagicModule(__name__)
mod.__dict__.update(old_mod.__dict__)
```

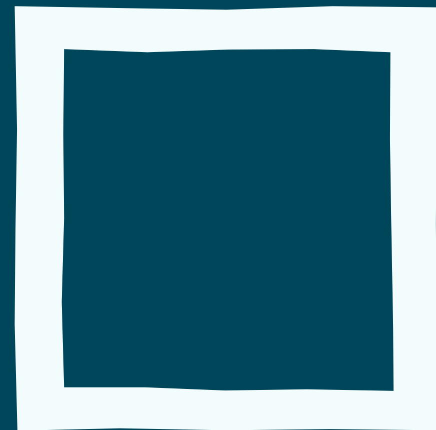
```
>>> import magic_module
>>> magic_module
<module 'magic_module' from 'magic_module.py'>
>>> magic_module.git_hash
'da39a3ee5e6b4b0d3255bfef95601890afd80709'
```



Custom Namespaces

```python
from collections import MutableMapping


class CaseInsensitiveNamespace(MutableMapping):
    def __init__(self):
        self.ns = {}
    def __getitem__(self, key):
        return self.ns[key.lower()]
    def __delitem__(self, key):
        del self.ns[key.lower()]
    def __setitem__(self, key, value):
        self.ns[key.lower()] = value
    def __len__(self):
        return len(self.ns)
    def __iter__(self):
        return iter(self.ns)
```

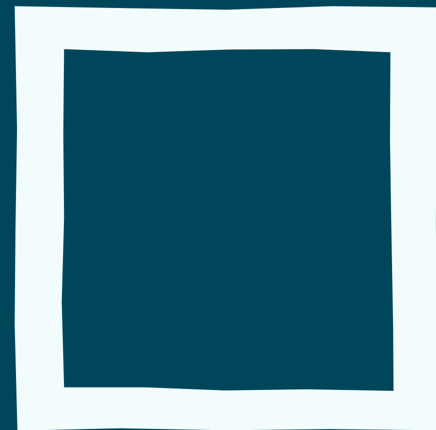
```
exec '''
foo = 42
Bar = 23
print (Foo, BAR)
''' in {}, CaseInsensitiveNamespace()
```



“Code Reloading”

True Reloading

- Design your application for reloading
- Acquire lock
- purge all entries in sys.modules with your module and that are linked to it.
- Reload the module in question and all dependencies.



Multiversioning

The Problem

- Want multiple versions of the same library loaded.
- But libraries are actual Python modules
- and modules are cached in sys.modules
- So how can we do that?

The Plan

- Import-hooks are no good. But …
- … __import__ can be monkey patched on __builtin__
- And __import__ can use _getframe() or the globals dict to look into the callers namespace
- And can that way find out what is the required version.

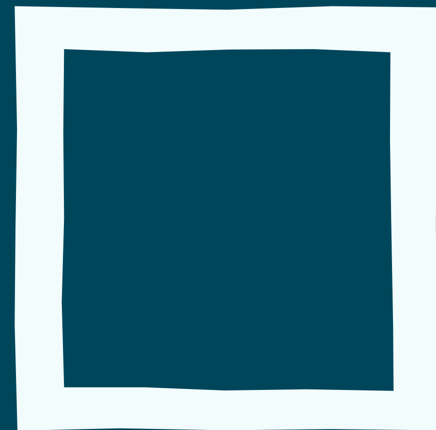
The Implementation

- You really don't want to know

```python
import multiversion
multiversion.require('mylib', '2.0')

# this will be version 2.0 of mylib
import mylib
```

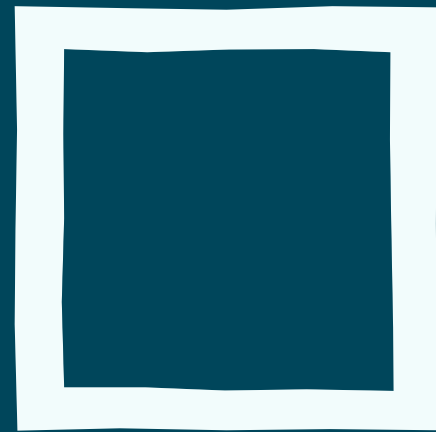
```
>>> import mylib
>>> mylib.__name__
'multiversion.space.mylib___322e30.mylib'
>>> mylib.__file__
'mylib-2.0/mylib.py'
```

Interpreter Warfare



Force Closures

```python
def some_generated_code():
    a = 1
    b = 2
    def inner_function():
        if 0: unused(a, b)
        return locals()
    return inner_function
```

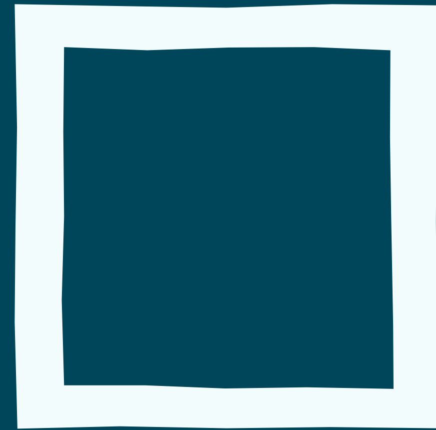


Patching Tracebacks

```python
try:
    ...
except Exception:
    exc_type, exc_value, tb = sys.exc_info()
    frames = make_frame_list(tb)
    frames = rewrite_frames(frames)
    prev_frame = None
    for frame in frames:
        if prev_frame is not None:
            tb_set_next(prev_frame, frame)
        prev_frame = frame
```

```python
import sys
import ctypes
from types import TracebackType

if hasattr(ctypes.pythonapi, 'Py_InitModule4_64'):
    _Py_ssize_t = ctypes.c_int64
else:
    _Py_ssize_t = ctypes.c_int

class _PyObject(ctypes.Structure):
    pass
_PyObject._fields_ = [
    ('ob_refcnt', _Py_ssize_t),
    ('ob_type', ctypes.POINTER(_PyObject))
]

if hasattr(sys, 'getobjects'):
    class _PyObject(ctypes.Structure):
        pass
    _PyObject._fields_ = [
        ('_ob_next', ctypes.POINTER(_PyObject)),
        ('_ob_prev', ctypes.POINTER(_PyObject)),
        ('ob_refcnt', _Py_ssize_t),
        ('ob_type', ctypes.POINTER(_PyObject))
    ]
```

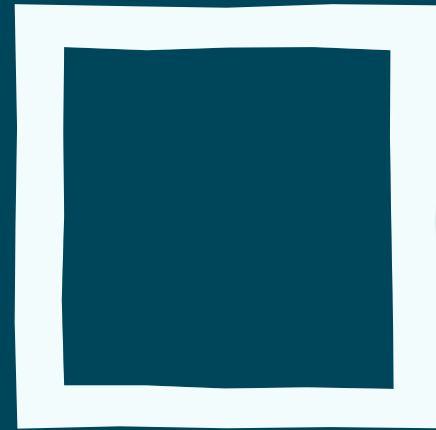
```python
class _Traceback(_PyObject):
    pass
_Traceback._fields_ = [
    ('tb_next', ctypes.POINTER(_Traceback)),
    ('tb_frame', ctypes.POINTER(_PyObject)),
    ('tb_lasti', ctypes.c_int),
    ('tb_lineno', ctypes.c_int)
]

def tb_set_next(tb, next):
    if not (isinstance(tb, TracebackType) and
            (next is None or isinstance(next, TracebackType))):
        raise TypeError('tb_set_next arguments must be tracebacks')
    obj = _Traceback.from_address(id(tb))
    if tb.tb_next is not None:
        old = _Traceback.from_address(id(tb.tb_next))
        old.ob_refcnt -= 1
    if next is None:
        obj.tb_next = ctypes.POINTER(_Traceback)()
    else:
        next = _Traceback.from_address(id(next))
        next.ob_refcnt += 1
        obj.tb_next = ctypes.pointer(next)
```



Names of Variables

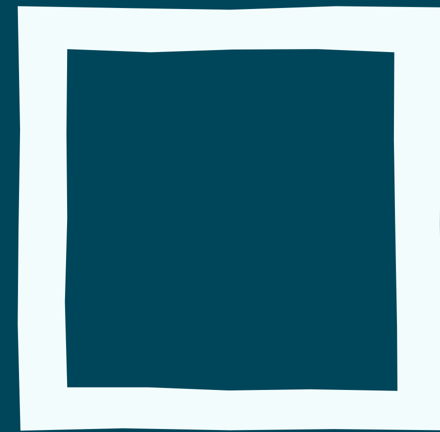
```python
import gc, sys

def find_names(obj):
    frame = sys._getframe(1)
    while frame is not None:
        frame.f_locals
        frame = frame.f_back
    result = set()
    for referrer in gc.get_referrers(obj):
        if isinstance(referrer, dict):
            for k, v in referrer.iteritems():
                if v is obj:
                    result.add(k)
    return tuple(result)
```

```
>>> a = []
>>> b = a
>>> find_names(a)
('a', 'b')
```



Bytecode Hacks

```python
from opcode import HAVE_ARGUMENT


def disassemble(code):
    code = map(ord, code)
    i = 0
    n = len(code)
    while i < n:
        op = code[i]
        i += 1
        if op >= HAVE_ARGUMENT:
            oparg = code[i] | code[i + 1] << 8
            i += 2
        else:
            oparg = None
        yield op, oparg
```

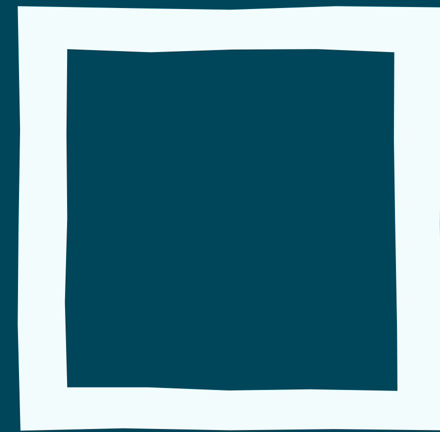


Implicit Self

```python
class User(ImplicitSelf):

    def __init__(username, password):
        self.username = username
        self.set_password(password)

    def set_password(pw):
        self.hash = sha1(pw).hexdigest()

    def check_password(pw):
        return sha1(pw).hexdigest() == self.hash
```

```python
def inject_self(code):
    varnames = ('self',) + tuple(n for i, n in
                                  enumerate(code.co_varnames))
    names = tuple(n for i, n in enumerate(code.co_names))
    bytecode = []

    for op, arg in disassemble(code.co_code):
        if op in (LOAD_FAST, STORE_FAST):
            arg = varnames.index(code.co_varnames[arg])
        elif op in (LOAD_GLOBAL, STORE_GLOBAL):
            if code.co_names[arg] == 'self':
                op = LOAD_FAST if op == LOAD_GLOBAL else STORE_FAST
                arg = 0
            else:
                arg = names.index(code.co_names[arg])
        elif op in (LOAD_ATTR, STORE_ATTR):
            arg = names.index(code.co_names[arg])
        bytecode.append(chr(op))
        if op >= opcode.HAVE_ARGUMENT:
            bytecode.append(chr(arg & 0xff))
            bytecode.append(chr(arg >> 8))

    return ''.join(bytecode), varnames, names
```

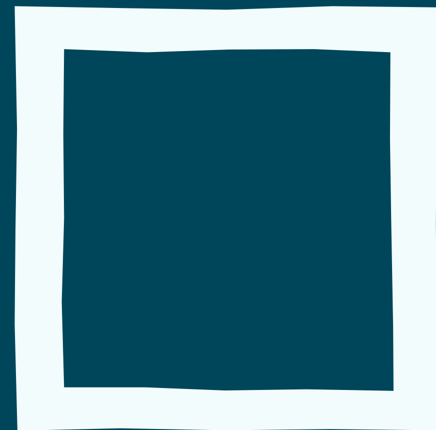
```python
from types import CodeType, FunctionType


def implicit_self(function):
    code = function.func_code
    bytecode, varnames, names = inject_self(code)
    function.func_code = CodeType(code.co_argcount + 1,
        code.co_nlocals + 1, code.co_stacksize, code.co_flags, bytecode,
        code.co_consts, names, varnames, code.co_filename, code.co_name,
        code.co_firstlineno, code.co_lnotab, code.co_freevars,
        code.co_cellvars)


class ImplicitSelfType(type):

    def __new__(cls, name, bases, d):
        for key, value in d.iteritems():
            if isinstance(value, FunctionType):
                implicit_self(value)
        return type.__new__(cls, name, bases, d)


class ImplicitSelf(object):
    __metaclass__ = ImplicitSelfType
```



Return Value Used?

```python
def menu_items():
    items = MenuItem.query.all()
    html = render_menu_items(items=items)
    if return_value_used():
        return html
    print html
```

```python
import sys, dis


def return_value_used():
    frame = sys._getframe(2)
    code = frame.f_code.co_code[frame.f_lasti:]
    try:
        has_arg = ord(code[0]) >= dis.HAVE_ARGUMENT
        next_code = code[3 if has_arg else 1]
    except IndexError:
        return True
    return ord(next_code) != dis.opmap['POP_TOP']
```

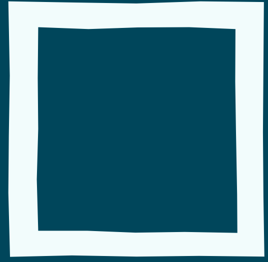


What'll be my name?

```
>>> class Module(object):
...  def __init__(self):
...   self.name = assigned_name()
...
>>> admin = Module()
>>> admin.name
'admin'
```

```python
import sys, dis


def assigned_name():
    frame = sys._getframe(2)
    code = frame.f_code.co_code[frame.f_lasti:]
    try:
        has_arg = ord(code[0]) >= dis.HAVE_ARGUMENT
        skip = 3 if has_arg else 1
        next_code = ord(code[skip])
        name_index = ord(code[skip + 1])
    except IndexError:
        return True
    if next_code in (dis.opmap['STORE_FAST'],
                     dis.opmap['STORE_GLOBAL'],
                     dis.opmap['STORE_NAME'],
                     dis.opmap['STORE_DEREF']):
        namelist = frame.f_code.co_names
        if next_code == dis.opmap['STORE_GLOBAL']:
            namelist = frame.f_code.co_names
        elif next_code == dis.opmap['STORE_DEREF']:
            namelist = frame.f_code.co_freevars
        return namelist[name_index]
```

Questions :-)



Legal

Slides: http://lucumr.pocoo.org/talks/
Code: http://github.com/mitsuhiko/badideas

armin.ronacher@active-4.com // @mitsuhiko

Slides licensed under the Creative Commons attribution-noncommercial-sharealike license. Code examples BSD licensed. Original implementation of `find_names()` by Georg Brandl