# Decorators & Context Managers

sourceforge

Dave Brondsema
dave@brondsema.net

Friday, March 9, 2012

who i am
sprint

## Raise your hands

http://www.flickr.com/photos/meer/2658968648/

how many have used a decorator?
written a decorator?
used a context manager (with statement)?
written a context manager

# POWER

these are powerful language features - why don't we use them more often?
by the end of this presentation, i want you to know enough about decorators and context mangers,
that you can go back to your code and start using them next week

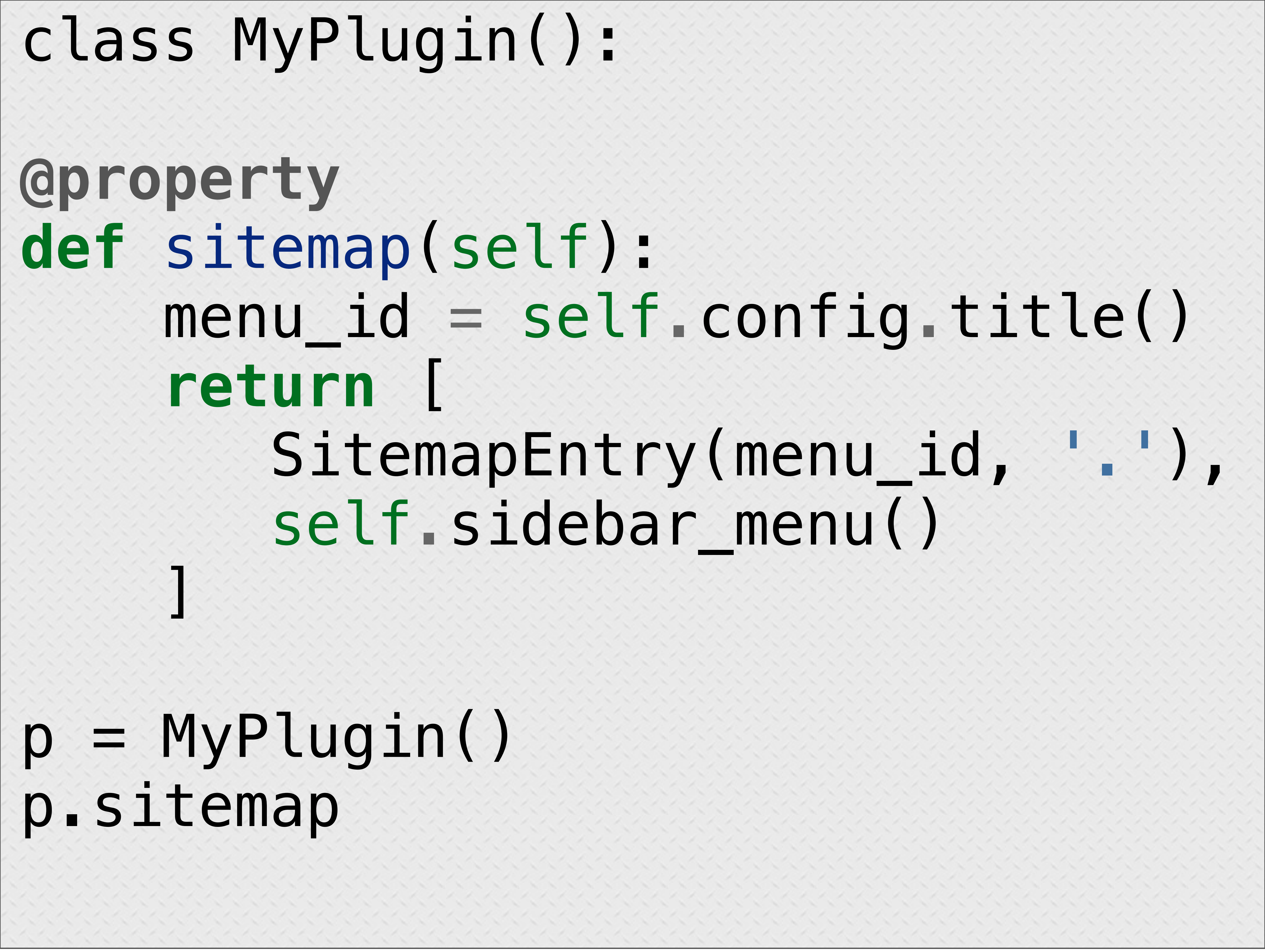
## Decorators

```
def my_function(foo, bar, baz=12):
    'this does boring stuff'
    return 123


run_this_later(my_function)


my_function.special_blah = True


print my_function.__doc__
'this does boring stuff'
```

this is a function, right
you all know that
but did you know that you can pass functions around?
and add attributes onto them?
they are objects, so we can do fun stuff with them

## Example

of USING a decorator



```
class MyPlugin():

@property
def sitemap(self):
    menu_id = self.config.title()
    return [
        SitemapEntry(menu_id, '.'),
        self.sidebar_menu()
    ]

p = MyPlugin()
p.sitemap
```

what it's actually doing doesn't matter.  just notice that it is complex, not a single attribute
but now we can access it like an attribute
@property is part of the python standard library

```
class MyPlugin():

def sitemap(self):
    menu_id = self.config.title()
    return [
        SitemapEntry(menu_id, '.'),
        self.sidebar_menu()
    ]
sitemap = property(sitemap)
```

Friday, March 9, 2012

a decorator wraps a function
it is shorthand for this
so a decorator itself is a function. it receives a function as the first argument. and it returns a function

## Your first decorator

in a webapp, decorators are used
to associate methods with urls, templates, etc

```
 def expose(func):
     func.exposed = True
     return func

@expose
def myview():
    # do stuff
    return dict(records=records)

>>> myview.exposed
True
```

Friday, March 9, 2012

so a decorator itself is a function. it receives a function as the first argument. and it returns a function, either the same one or a new one
as soon as this code is imported, python runs expose(myview)

```python
def expose(template):
    def mark_exposed(func):
        func.exposed = True
        func.template = template
        return func
    return mark_exposed

@expose('view.html')
def myview():
    # do stuff
    return dict(records=records)
```

parameters
you are not passing a parameter to a decorator
you are calling the expose function with a parameter
and *that* has to return the decorator function

## let's **DO** something

Friday, March 9, 2012

we can return a different function!

```python
def memoize(func):
    def check_cache(*args):
        if not hasattr(func, 'results'):
            func.results = {}
        if args not in func.results:
            func.results[args] = func(*args)
        return func.results[args]
    return check_cache


@memoize
def find_user(user_id):
    'query database and load User object'
    return User.m.get(_id=user_id)
```

"memoize" is caching for a particular function.
Explain details
of course your cache could get really big, be smart about where you use this
BUT we have a problem!

```
>>> print find_user
<function check_cache at 0x1028628

>>> print find_user.__name__
check_cache

>>> print find_user.__doc__
None
```

Friday, March 9, 2012

we replaced our function with check_cache

```
from decorator import decorator
@decorator
def memoize(func, *args):
    if not hasattr(func, 'results'):
        func.results = {}
    if args not in func.results:
        func.results[args] = func(*args)
    return func.results[args]


def memoize(func):
    def check_cache(*args):
        if not hasattr(func, '_results'):
        ...
```

Friday, March 9, 2012

So we use the @decorator decorator, which preserves that for us.
This also lets us remove the inner function, which will be nice if we want to use a parameter to our decorator so things won't get too confusing. There is a similar @deco in functools, but it doesn't flatten

flickr.com/photos/pinksherbet/2162778538/

you could have 3 nested functions: 1st to take parameters, 2nd is the actual decorator, 3rd is the function you return in place of the original

## callable objects

```
class say_something(object):

    def __init__(self, catchphrase):
        self.catchphrase = catchphrase

    def __call__(self):
        print self.catchphrase

buzz_lightyear = say_something('To infinity, '
                               'and beyond!')

>>> buzz_lightyear()
To infinity, and beyond!
```

you can call things that aren't functions
just make an object that has a __call__ method

```python
class memoize(object):
    def __init__(self, max):  # just an example
        self.max = max          # not used

    def __call__(self, func):
        return decorator(self.check_cache, func)

    def check_cache(self, func, *args):
        # TODO: use self.max
        if not hasattr(func, 'results'):
            func.results = {}
        if args not in func.results:
            func.results[args] = func(*args)
        return func.results[args]

@memoize(max=3)
def my_func...
```

Friday, March 9, 2012

doesn't have to be function; callable object instead
init for params, instead of nesting functions
more typical way to hold on to attributes
can still use decorator(), but differently

## stacking

```
@patch('smtplib.SMTP.sendmail')
@patch('sf.consume.controllers.user.g')
def test_registration(self, g, sendmail):
    g.user = None
    resp = self.app.post(...)
    email = sendmail.call_args[0][2]
    assert 'testuser@example.org' in email
```

You can have multiple decorators, the closest gets applied first, then the next
be careful, with decorators that replace functions vs ones that set attributes

# class decorator

```
from functools import total_ordering

@total_ordering
class Student:
    def __eq__(self, other):
        return ((self.lastname.lower(), self.f
                 ==
                (other.lastname.lower(), other
    def __lt__(self, other):
        return ((self.lastname.lower(), self.f
                 <
                (other.lastname.lower(), other
```

Friday, March 9, 2012

a function that takes a class, and should return a class
copied straight from python docs

## more ideas

@validate

@with_trailing_slash

@classmethod

@rest.restrict('POST')

@email_audit('team@...')

@synonym_for

@raises

These are just a few of the decorators we use at sourceforge

## Context Managers

[illegible]

flickr.com/photos/philipedmondson/6851697525/

Friday, March 9, 2012

there are a lot of repeated patterns to setup & teardown
look for these pairs that go together, and you will have happy peaceful kittens - i mean code

```python
with open('/etc/passwd') as f:
    for line in f:
        print line.split(':')[1]



try:
    f = open('/etc/passwd')
    for line in f:
        print line.split(':')[1]
finally:
    f.close()
```

open() returns f, a file object that is also a context manager
ctx manager makes sure f.close() is always called.
replaces try/finally

```
class working_dir(object):
    def __init__(self, new_dir):
        self.new_dir = new_dir
        self.orig_dir = os.getcwd()

    def __enter__(self):
        os.chdir(self.new_dir)

    def __exit__(self, exc_type, exc_val, exc_t
        os.chdir(self.orig_dir)

with working_dir('/etc'):
    # do whatever
# and back to original dir
```

init saves the directory and the original dir
enter changes it
exit restores it

```python
import contextlib

@contextlib.contextmanager
def working_dir(new_dir):
    orig_dir = os.getcwd()
    os.chdir(new_dir)
    try:
        yield              # end of __enter__
    finally:
        os.chdir(orig_dir)

with working_dir('/etc'):
    # do whatever
# and back to original dir
```

shortcut to do it as a function instead of a class
do stuff, yield (an object, optionally), do cleanup
yield makes this a generator function.  The deco converts it
try/finally needed to make sure cleanup happens even if an error occurs

# refactor!

```
  # doing regular stuff
user = request.user
request.user = the_admin
# do some special logic
# that you have to do as admin
request.user = user
# back to regular stuff
```

---

```
# doing regular stuff
with admin_user(req):
    # do some special logic
    # as an admin user
# back to regular stuff
```

Use a context manager to apply something to a section of code - it makes it clear!
setting "context"
likewise, use a decorator to apply something to a function

```
from urllib2 import urlopen
from contextlib import closing

with closing(urlopen(some_api)) as foo, \
     closing(urlopen(other_api)) as bar:
    # do your stuff
    # here
```

closing() automatically calls close()
python 2.7 added multiple ctx mgrs
contextlib has a nested() helper otherwise
you might not call close() anyway, since it goes out of scope.
good practice

## error handling patterns

```
try:
    do_stuff()
except UnicodeError:
    print 'annoying'
except ValueError:
    print 'hrmm'
except OSError:
    print 'alert sysadmins'
except:
    print 'other'
```

do you do the same complex error handling in multiple places? even if you don't have this many conditions, if you repeat the same handling - DONT REPEAT YOURSELF

```
class my_error_handling(object):
  def __enter__(self): pass

  def __exit__(self, exc_type, exc_val, exc_tb)
      if issubclass(exc_type, UnicodeError):
          print 'annoying'
      elif issubclass(exc_type, ValueError):
          print 'hrmm'
      elif issubclass(exc_type, OSError):
          print 'alert sysadmins'
      else:
          print 'other'
      return True

 with my_error_handling():
     do_stuff()
```

you get all the exception details
return True to suppress further error handling

# more ideas

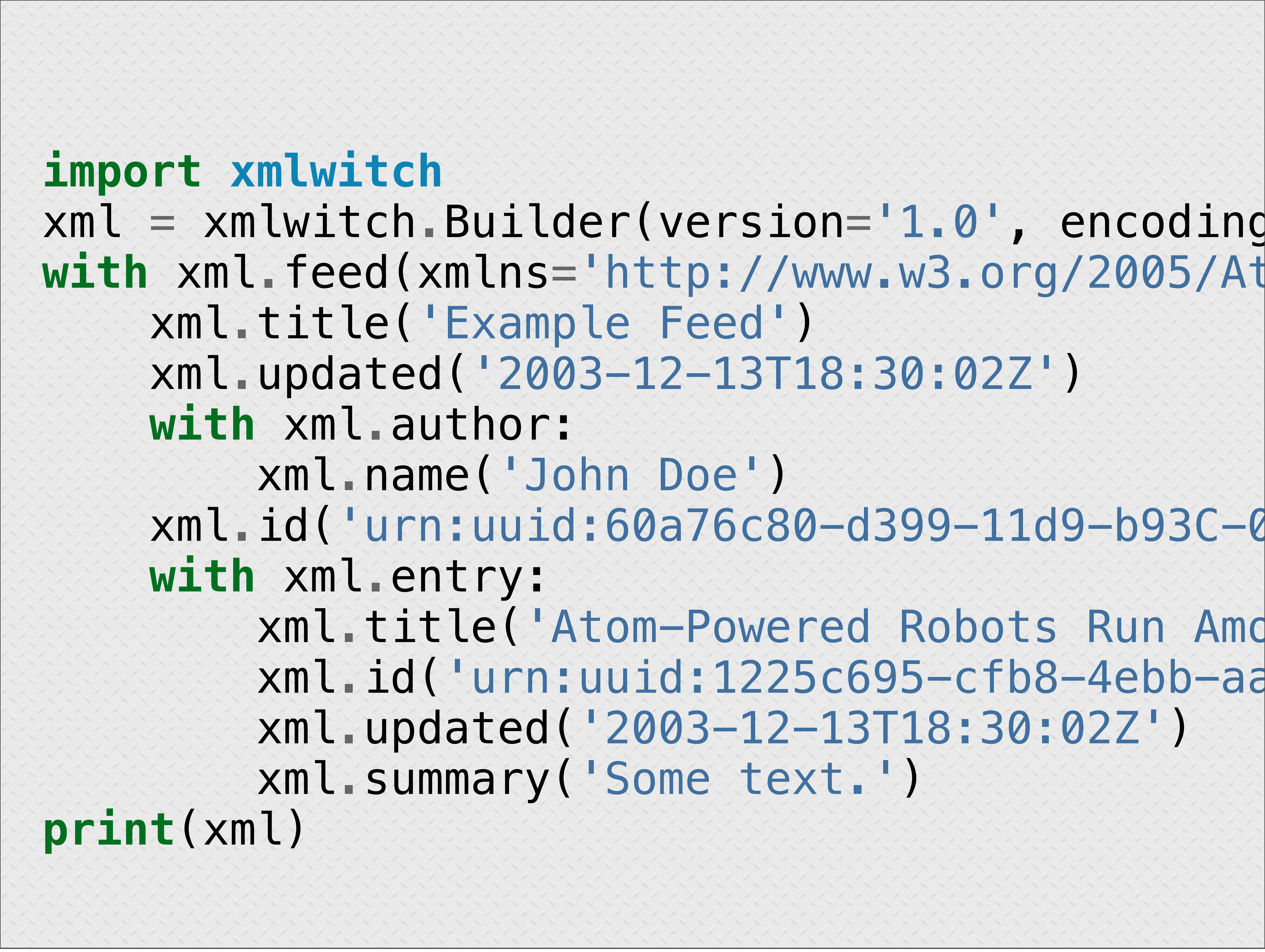
acquire locks

set global variables or flags

timing

monkey patching

transactions



```
import xmlwitch
xml = xmlwitch.Builder(version='1.0', encoding
with xml.feed(xmlns='http://www.w3.org/2005/At
    xml.title('Example Feed')
    xml.updated('2003-12-13T18:30:02Z')
    with xml.author:
        xml.name('John Doe')
    xml.id('urn:uuid:60a76c80-d399-11d9-b93C-0
    with xml.entry:
        xml.title('Atom-Powered Robots Run Amo
        xml.id('urn:uuid:1225c695-cfb8-4ebb-aa
        xml.updated('2003-12-13T18:30:02Z')
        xml.summary('Some text.')
print(xml)
```

Friday, March 9, 2012

general-purpose block indentation
make your code structure match your data and logic

# Use Your Tools

Friday, March 9, 2012

re-use

separation of concerns

organize your code better

go write them and use them

```
@end_with_question_time
def my_presentation():
    ...
    with thanks_for_listening():
        return
```

Dave Brondsema
SourceForge
dave@brondsema.net

# Slides at: speakerdeck.com/u/brondsem

- http://docs.python.org/glossary.html#term-decorator
- http://www.python.org/dev/peps/pep-0318/
- http://micheles.googlecode.com/hg/decorator/documentation.html
- http://hangar.runway7.net/decorators-wrappers-python
- http://docs.python.org/library/functools.html
- http://hairysun.com/blog/2012/02/04/learning-python-decorators-handout/
- http://www.voidspace.org.uk/python/mock/patch.html#nesting-patch-decorators

- http://docs.python.org/reference/compound_stmts.html#the-with-statement
- http://www.python.org/dev/peps/pep-0343/
- http://docs.python.org/library/contextlib.html
- http://tomerfiliba.com/blog/Code-Generation-Context-Managers/
- http://tomerfiliba.com/blog/Toying-with-Context-Managers/
- https://github.com/galvez/xmlwitch
- http://pypi.python.org/pypi/altered.states