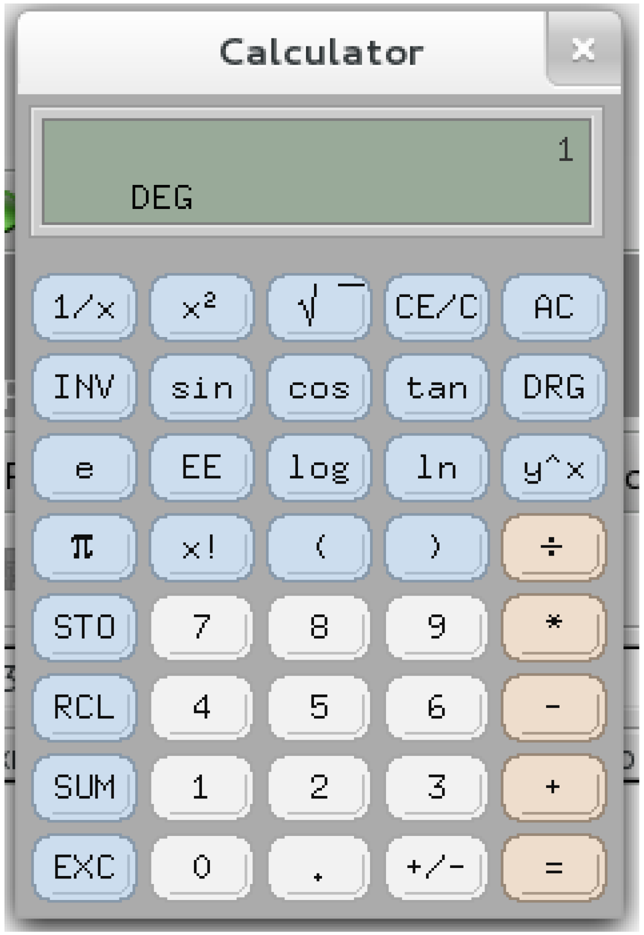
Mock

For testing code used here.

The First Three Rules

1. Writing code is easy.
   a. Writing good code is much harder.
2. Writing tests is tedious and does not impress girls or people who wear suits [unless the girl is a hacker, maybe not even then].
   a. But the only way to know if code is good code is to test it [somehow]; see #1.
3. Writing documentation is very hard.
   a. But the only useful, and thus good, code is documented code; see #1.
      1. No your inline code comments and doc strings are not @&*^$&*@ documentation. Documentation is in the form of **documents** which explain how to use the code and the why of how it works the way it does.

Warning



1. Nowhere in this presentation will there be an example of how to test if some method can add two integers together [correctly].
   1. This is in flagrant violation of the international statutes regarding presentations concerning unit tests.
      1. The U.N. is probably dispatching black helicopters as we speak.
   2. If you ever write a method that merely adds two integers together... you are an idiot.

```
x = lambda a, b : a + b
```

unittest

```
def test_update_from_fields1(self):

    SQL_TEXT = \
        'UPDATE earth_moving SET shovelcolor = :1,shovelpower = :2 ' \
        'WHERE record_id = :3 AND first_name = :4 AND last_name = :5'

    x = SQLCommand()
    sql, values = x._create_update_from_fields(
        connection=self.db,
        table='earth_moving',
        keys={'record_id': 123, 'first_name': 'mike', 'last_name': 'mulligan', },
        fields={'shovelpower': 'steam',   'shovelcolor': 'red'})

    self.assertEquals(sql, SQL_TEXT)
    self.assertEquals(len(values), 5)
    self.assertEquals(values[0], 'red')
    self.assertEquals(values[1], 'steam')
    self.assertEquals(values[2], 123)
    self.assertEquals(values[3], 'mike')
    self.assertEquals(values[4], 'mulligan')
```

but real code actually does stuff

Looks in a database

Calls Logic layers

```
def _load_context_ids(self, account_id)
    Context._load_context_ids(self, account_id)
    query = self.db_session( ).\
        query(Team).\
        join(CompanyAssignment).\
        filter(CompanyAssignment.child_id == account_id, ).\
        enable_eagerloads(False)
    self._C_context_ids.extend([t.object_id for t in query.all()])
    query = None
    self._C_context_ids.extend(
      [c.object_id for c in
       self.r_c('account::get-proxied-contacts')])
```

truth

testing hard :(

mock

mock is awesome

making testing less hard

does nothing

```
>>> import mock
>>> m = mock.Mock()
>>> m.fred
<Mock name='mock.fred' id='140181180757904'>
>>> m.fred.george.stanley
<Mock name='mock.fred.george.stanley'
id='140181180759632'>
```

Mocks are callable, they return more Mock objects.

This is why hammers are awesome and never become obsolete – they don't care what you are hitting.

mock & properties

```
x=mock.Mock()
p=mock.Mock(return_value='fred')
x.stanley = p
print x.stanley
print x.stanley()

print('-')
x=mock.Mock()
p=mock.PropertyMock(return_value='fred')
type(x).stanley = p
print x.stanley
```

```
<Mock name='mock.stanley' id='...'>
fred
-
fred
```

wraps=instance

```
>>> class MyObject(object):
...     def hello(self):
...         print('world')
...
>>> x = MyObject()
>>> m = mock.Mock(wraps=x)
>>> x.hello()
world
>>> x.fred()
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
AttributeError: 'MyObject' object has no attribute 'fred'
```

what happens when I wrap

```
import mock
class Nitrox(object):

    def red_line(self, percent=32):
        if percent == 32:
            return 110
        raise Exception('WTF?')

m = mock.Mock(wraps=Nitrox())
print m.red_line
print m.red_line()
```

```
<Mock name='mock.red_line' id='139897343965776'>
110
```

override via wraps

```
import mock

class Nitrox(object):

    def red_line(self, percent=32):
        print('ping')
        if percent == 32:
            return 110
        raise Exception('Seizures are bad')

m = mock.Mock(wraps=Nitrox())
m.red_line.return_value = 0
print m.red_line
print m.red_line()
```

```
<Mock name='mock.red_line' id='140297648237136'>
0
```

forcing exceptions

```
m = mock.Mock(wraps=Nitrox())
m.red_line.side_effect = ValueError
print m.red_line
try:
    print m.red_line()
except ValueError:
    print('pong')
```

```
<Mock name='mock.red_line' id='139877572401744'>
pong
```

spec=instance

```
>>> m=mock.Mock(spec=int())
>>> m.__class__
<type 'int'>
>>> m.numerator
<Mock name='mock.numerator' id='140181180798480'>
>>> m + 3
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
TypeError: unsupported operand type(s) for +: 'Mock' and 'int'
```

spec vs. wrap

```
>>> m = mock.Mock(spec=3)
>>> isinstance(m, int)
True
>>> m = mock.Mock(spec=int())
>>> isinstance(m, int)
True
>>> m = mock.Mock(wraps=int())
>>> isinstance(m, int)
False
```

spec=[att1, attr2, attr3, ...]

```
>>> m = mock.Mock(spec=['george', 'fred', 'stanley'], )
>>> m.george
<Mock name='mock.george' id='140181151544656'>
>>> m.fred
<Mock name='mock.fred' id='140181151544720'>
>>> m.stanley
<Mock name='mock.stanley' id='140181151544848'>
>>> m.henry
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
  File "/home/awilliam/projects/coils/lib/python2.7/site-
packages/mock.py", line 658, in __getattr__
    raise AttributeError("Mock object has no attribute %r" % name)
AttributeError: Mock object has no attribute 'henry'
```

configured via dict

```
import mock
x = mock.Mock(**{'get_name.return_value': 'stanley', })
print x.get_name
print x.get_name()
```

```
<Mock name='mock.get_name' id='140036218874512'>
stanley
```

configured via method

```
x = mock.Mock()
config={'get_name.return_value': 'stanley', }
x.configure_mock(**config)
print x.get_name
print x.get_name()
```

```
<Mock name='mock.get_name' id='140036218874512'>
stanley
```

chained spec

Suddenly this begins to smell like useful!

```
from mock import Mock
from coils.foundation import Contact

sqla = Mock()
config={'query.return_value.filter.return_value.all.return_value': [], }
sqla.configure_mock(**config)
x = sqla.query(Contact).\
    filter(Contact.first_name=='stanley').\
    all()

print('Result:{0}'.format(x, ))
```

```
Result:[]
```

patchyness

```
class ABC(object):

    def calc(self):
        return 1.74

class XYZ(object):

    @mock.patch.object(ABC, 'calc', return_value=42)
    def  test_abc_calc(self, mock_abc):
        obj = ABC()
        return obj.calc()

x = XYZ()
print x.test_abc_calc()
```

```
                                                    42
```

side_effects callable

Full-monty programmatic fake-out!

```
def side_effect_example(*args, **kargs):
   if args[0] == 'account::get-proxied-contacts':
       a = mock.Mock(wraps=coils.core.Contact)
       type(a).object_id=mock.PropertyMock(return_value=8999)
       b = mock.Mock(wraps=coils.core.Contact)
       type(b).object_id=mock.PropertyMock(return_value=9999)
       return [a, b, ]
   return None

ctx = mock.Mock()
ctx.r_c.side_effect = side_effect_example
for c in ctx.r_c('account::get-proxied-contacts'):
    print c.object_id
```

```
8999
9999
```

mock the command

```
def mock_run_command(*args, **kargs):
   if args[0] == 'account::get-proxied-contacts':
       a = mock.Mock(wraps=Contact)
       type(a).object_id=mock.PropertyMock(return_value=8999)
       b = mock.Mock(wraps=Contact)
       type(b).object_id=mock.PropertyMock(return_value=9999)
       return [a, b, ]
   return None
```

mock the orm

```
t1 = mock.Mock(wraps=Team)
type(t1).object_id = mock.PropertyMock(return_value=4000)
t2 = mock.Mock(wraps=Team)
type(t2).object_id = mock.PropertyMock(return_value=4001)

db_session=mock.Mock()

config = {
    'query.return_value.'
    'join.return_value.'
    'filter.return_value.'
    'enable_eagerloads.return_value':
        mock.Mock(**{'all.return_value': [t1, t2, ], }), }

db_session.configure_mock(**config)
```

use the mojo

```
class XYZ(object):

    @mock.patch.object(
          UserContext, 'db_session',
          return_value=db_session, )
    @mock.patch.object(
          UserContext, 'run_command',
           side_effect=side_effect_example, )
    def test_user_ctx(self, mock1, mock2):
        obj = UserContext(10100, {})
        print obj.context_ids

z = XYZ()
z.test_user_ctx()
```

Supposed to return a summation of the context ids of the user, the group memberships, and the proxy relationships.

[10100, 4000, 4001, 8999, 9999]

what happened

```
@mock.patch.object(
      UserContext, 'db_session',
      return_value=db_session, )
@mock.patch.object(
      UserContext, 'run_command',
      side_effect=mock_run_command, )
def test_user_ctx(self, runcmd_mock, db_session_mock):
    obj = UserContext(10100, {})
    print obj.context_ids
   runcnd_mock.assert_called_once_with(
           'account::get-proxied-contacts')
    db_session_mock.assert_called_once()
```

```
[10100, 4000, 4001, 8999, 9999]
```

more probing

```
def test_search_tasks_criteria(self):

    context = Mock(**{'run_command.return_value': [], })
    api = ZOGIAPI(context)
    api.search_task(
            [{'key': 'name', 'value': 'something'}, ],
            {'limit': 150, },
            65535)
    name, args, kwargs = context.mock_calls[0]
    self.assertEquals(name, 'run_command')
    self.assertEquals(args[0], 'task::search')
    self.assertIsInstance(kwargs['criteria'], list)
    self.assertIsInstance(kwargs['criteria'][0], dict)
    self.assertEquals(kwargs['orm_hints'], ('zogi', 65535, ))
    self.assertEquals(kwargs['limit'], 150)
```