For Those About to Mock

Why I Test

- Test because I'm prone to silly mistakes
- Because testing a function often highlights bad function signatures etc
- Test to Enforce Behaviors (Good and Bad)

Testing Toolkits

- Unittest / Nose
- PyTest
- DocTests (not covering today)
- So many other things

Unittest is our Quiet Riot

^ Part of the Standard Library

^ Often used in conjunction with Nose

^ Built as a python class

^ Regularly coupled with Nose

Py.Test is our Foo Fighters

Has to be pip installed (pytest)

^ Has a bunch of extra tricks

^ Fixtures

^ Parameterized Testing

^ Works with existing Unittest, Nose, and DocTest tests

Setting up a Unittest TestSuite

- built like a python module (needs a `__init__.py`)
- A class per group of tests

Unittest TestSuite Example

```
import unittest

from metal.hair import count_hairspray_cans
from rock.foo_fighters import are_the_foo_fighters_still_a_band

class TestRock(unittest.TestCase):

    def test_count_hairspray_cans(self):
        self.assertEqual(10, count_hairspray_cans())

    def test_is_rock_still_alive(self):
        self.assertTrue(are_the_foo_fighters_still_a_band())
```

Setting up a Py.Test TestSuite

- `pip install pytest`
- Just a collection of Functions
- Can be grouped with classes

Py.Test TestSuite Example

```
import pytest

from metal.hair import count_hairspray_cans
from rock.foo_fighters import are_the_foo_fighters_still_a_band

class TestRock:

    def test_count_hairspray_cans(self):
        assert count_hairspray_cans() == 10

    def test_is_rock_still_alive(self):
        assert are_the_foo_fighters_still_a_band() is True
```

Testing Vocabulary Lessons

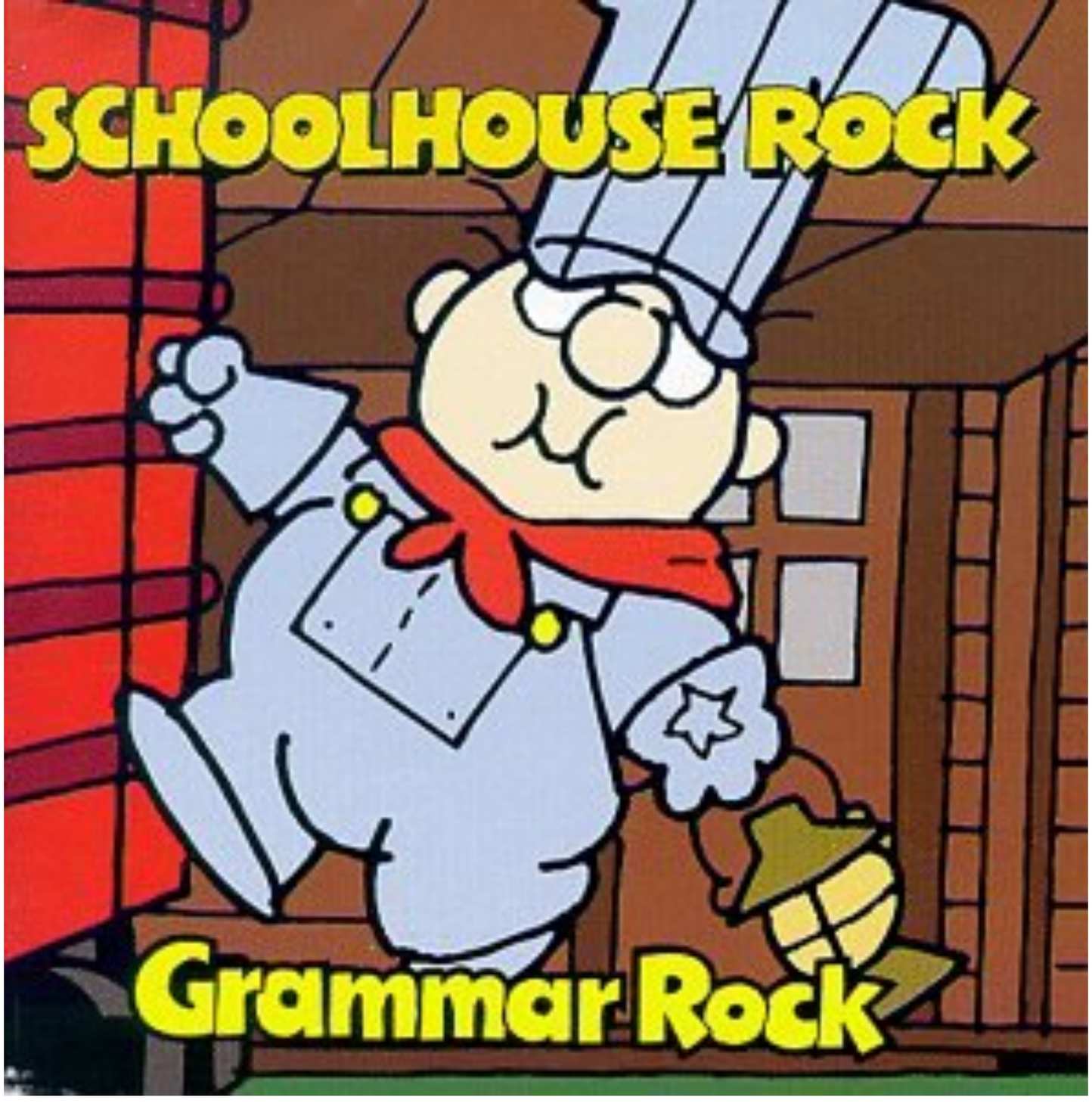
Disclaimer all these bands are awesome, okay maybe not spinal tap. I had to find a way to work in my favorite rock covers.

Fake: objects actually have working implementations, but usually take some shortcut which makes them not suitable for production.

Mock: a class that implements an interface and allows the ability to dynamically set the values to return/exceptions to throw from particular methods and provides the ability to check if particular methods have been called/not called.

Stub: provide canned answers to calls made during the test, usually not responding at all to anything outside what's programmed in for the test. (The Belles do originals sometimes.)

Rock ain't about Vocabulary



Okay fine maybe it is whatever man

Our First Code to Test

app.py

```
def hello_world():
    return {'message': 'Hello World'}
```

simple function that returns a dictionary

Our First Test

unittests.py

```
import unittest

from app import hello_world


class TestFlaskApp(unittest.TestCase):

    def test_hello_world(self):
        result = hello_world()
        self.assertDictEqual({'message': 'Hello World'}, result)
```

```
(for-those-about-to-mock) ~/d/f/rock ❯❯❯ python -m unittest unittests
.
----------------------------------------------------------------------
Ran 1 test in 0.000s

OK
```

dal.py

```
def process_results(results):
    if results.status_code == 404:
        return {'message': 'Rock Not found!'}
    elif results.status_code == 500:
        return {'message': 'Rock Imploded!'}
    elif results.status_code == 200:
        return results.json()
    else:
        return {'message': 'this function works like Ozzy - SHARON!'}
```

unittests.py

```
def test_process_results_not_found(self):
    HellsBelles = namedtuple("HellsBelles", ['status_code', ])
    hells_belles = HellsBelles(status_code=404)
    result = process_results(hells_belles)
    self.assertDictEqual({'message': 'Rock Not Found!'}, result)
```

```
(for-those-about-to-mock) ~/d/for-those-about-to-mock ❯❯❯ python -m unittest rock.unittests
..
----------------------------------------------------------------------
Ran 2 tests in 0.000s
```

dal.py

```
def process_results(results):
    if results.status_code == 404:
        return {'message': 'Rock Not found!'}
    elif results.status_code == 500:
        return {'message': 'Rock Imploded!'}
    elif results.status_code == 200:
        return results.json()
    else:
        raise ValueError('SHARON!')
```

To test this we need to pass in something for results that has a status_code attribute and a json() method... We could summon Spinal Tap and fake this, but what we really want is Mini Kiss, I need something that acts like a real result from requests, but isn't the full response object.

unittests.py

```
def test_process_results_success(self):
    mini_kiss = MagicMock()
    mini_kiss.status_code = 200
    mini_kiss.json.return_value = {'message': 'This is just a tribute!'}
    result = process_results(mini_kiss)
    self.assertDictEqual({'message': 'This is just a tribute!'}, result)
```

```
(for-those-about-to-mock) ~/d/for-those-about-to-mock ❯❯❯ python -m unittest rock.unittests
...
----------------------------------------------------------------------
Ran 3 tests in 0.001s
```

But Umm it called that function on our test_object...

```
from mock import call

class TestDal(unittest.TestCase)
    def test_process_results_success(self):
          mini_kiss = MagicMock()
          mini_kiss.status_code = 200
          mini_kiss.json.return_value = {'message': 'This is just a tribute!'}
          result = process_results(mini_kiss)
          self.assertDictEqual({'message': 'This is just a tribute!'}, result)

          expected_calls = [call.json()]
          self.assertListEqual(expected_calls, mini_kiss.mock_calls)
```

Let's check that call to make sure it happened and did the right thing

okay breath deep for a sec...

What have we learned here? That we can summon mini kiss via python at will?

This is just the surface of the awesome things you can do with MagicMock. For example, you can feed it a class as a spec and it will model the mock after that class!!! *mind blow*

Mountain of Mississippi Queen Fame.

Dealing with exceptions

- There's a method for that
- Drugs have side_effects

Far too soon Jim... far too soon.

dal.py

```
def process_results(results):
    if results.status_code == 404:
        return {'message': 'Rock Not found!'}
    elif results.status_code == 500:
        return {'message': 'Rock Imploded!'}
    elif results.status_code == 200:
        return results.json()
    else:
        raise ValueError('SHARON!')
```

```
def test_process_results_bad_status(self):
      mini_kiss = MagicMock()
      mini_kiss.status_code = 'cookies'
      self.assertRaises(ValueError, process_results, mini_kiss)

      expected_calls = []
      self.assertListEqual(expected_calls, mini_kiss.mock_calls)
```

```
def test_process_results_bad_status_message(self):
      mini_kiss = MagicMock()
      mini_kiss.status_code = 'cookies'
      with self.assertRaises(ValueError) as exc_info:
          process_results(mini_kiss)

          self.assertTrue('SHARON!' in exc_info.exception)

      expected_calls = []
      self.assertListEqual(expected_calls, mini_kiss.mock_calls)
```

We can validate the message the error raises as well

dal.py

```
def process_results(results):
    if results.status_code == 404:
        return {'message': 'Rock Not found!'}
    elif results.status_code == 500:
        return {'message': 'Rock Imploded!'}
    elif results.status_code == 200:
        return results.json()
    else:
        raise ValueError('SHARON!')
```

```
def test_process_results_bad_json_call(self):
      mini_kiss = MagicMock()
      mini_kiss.status_code = 200
      mini_kiss.json.side_effects = ValueError('Nickleback')
      with self.assertRaises(ValueError) as exc_info:
          process_results(mini_kiss)

          self.assertTrue('Nickleback' in exc_info.exception)

      expected_calls = [call.json()]
      self.assertListEqual(expected_calls, mini_kiss.mock_calls)
```

We can also have a mock throw an error for us to test our handling of that error.

Mocking built ins is ... odd

Mocking builtins

- `__builtin__` in Python 2 (boo/hiss)
- `builtins` in Python 3 (cheers)

```
@patch('__builtin__.open')
def test_process_file(self, open_mock):
    spinal_tap = BytesIO(b'Joan Jett\nJanis Joplin\nAlanis Morissette')
    expected_results = ['Joan Jett', 'Janis Joplin', 'Alanis Morissette']
    open_mock.return_value = spinal_tap
    results = process_file('cookies.csv')

    self.assertListEqual(expected_results, results)
```

Let's take a breather like the interludes in a great Stevie Nicks song...

Requests

requests

```
def fetch_some_data(url):
    session = requests.Session()
    results = None
    try:
        results = session.get(url)
    except ConnectionError as exc_info:
        print(str(exc_info))
    return results
```

```
from mock import patch

from rock.dal import fetch_some_data


@patch('rock.dal.requests.Session')
def test_fetch_some_data(requests_mock):
    requests_mock.return_value.get.return_value = 'Metalica'
    test_url = 'http://cookies.me/eat'
    results = fetch_some_data(test_url)
    assert 'Metalica' == results
    requests_mock.return_value.get.assert_called_once_with(test_url)
```

```
from requests import ConnectionError

@patch('rock.dal.requests.Session')
def test_fetch_some_data_conn_error(requests_mock):
    requests_mock.return_value.get.side_effect = ConnectionError('boom')
    test_url = 'http://cookies.me/eat'
    results = fetch_some_data(test_url)
    assert results is None
    requests_mock.return_value.get.assert_called_once_with(test_url)```
```

Betamax

Allows you to record and play back requests

```
def main():
    session = requests.Session()
    recorder = betamax.Betamax(
        session, cassette_library_dir=CASSETTE_LIBRARY_DIR
    )

    with recorder.use_cassette('fetch_some_data'):
        session.get('https://jsonplaceholder.typicode.com/posts')
```


B
INDEX
100
50
0
SONY
LOW-NOISE
Compact Cassette
JAPAN
C60

{"http_interactions": [{"request": {"body": {"string": "", "encoding": "utf-8"},
"headers": {"Connection": ["keep-alive"], "Accept-Encoding": ["gzip, deflate"],
"Accept": ["*/*"], "User-Agent": ["python-requests/2.10.0"]}, "method": "GET",
"uri": "https://jsonplaceholder.typicode.com/posts"}, "response":
{"body": {"base64_string": ...}

```
import betamax

with betamax.Betamax.configure() as config:
    current_dir = os.path.abspath(os.path.dirname(__file__))
    config.cassette_library_dir = os.path.join(current_dir, 'cassettes')

@patch('rock.dal.requests')
def test_fetch_some_data_betamax(requests_mock):
    session = requests.Session()
    requests_mock.Session.return_value = session
    with betamax.Betamax(session).use_cassette('fetch_some_data'):
        results = fetch_some_data('https://jsonplaceholder.typicode.com/posts')
    assert 'body' in results.text
    assert 1 == results.json()[0]['userId']
```

```
(for-those-about-to-mock) ~/d/f/rock ❯❯❯ py.test
==================== test session starts ============================
platform darwin -- Python 2.7.12, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /Users/jasonmyers/dev/for-those-about-to-mock/rock, inifile:
plugins: betamax-0.7.1
collected 4 items

tests/test_app.py .
tests/test_dal.py ...

=================== 4 passed in 0.08 seconds ========================
```

Testing API Endpoints

There are a ton of ways to test API endpoints. In fact you could say there is a Motley Crue of tesing methods... for example, You can use the built in django and flask test clients. Or we could use requests.

```
from flask import Flask, jsonify, Response


app = Flask(__name__)


def hello_world():
    return {'message': 'Hello World'}
```

```
@app.route('/hello/')
def api_hello_world():
    return jsonify(hello_world())
```

```
from flask_testing import LiveServerTestCase
import requests

from rock.app import app

class ApiTest(LiveServerTestCase):

    def create_app(self):
        app.config['TESTING'] = True
        # Default port is 5000
        app.config['LIVESERVER_PORT'] = 8943
        return app

    def test_hello_endpoint(self):
        response = requests.get(self.get_server_url() + '/hello/')
        self.assertEqual(response.status_code, 200)
```

Gabbi

```
tests:
- name: wait for server to boot
  GET: /hello/
  poll:
      count: 10
      delay: 0.1
  request_headers:
      Accept: application/json

  status: 200
  response_headers:
      Content-Type: application/json
  response_json_paths:
      $.message: Hello World
```

```
(for-those-about-to-mock) ~/d/for-those-about-to-mock ❯❯❯ sh ./rock/tests/test_api_gabbi.sh
* Running on http://127.0.0.1:5000/ (Press CTRL+C to quit)
... 127.0.0.1 - - [28/Jul/2016 18:54:58] "GET /hello/ HTTP/1.1" 200 -
✓ wait for server to boot

----------------------------------------------------------------------
Ran 1 test in 0.034s

OK
```

Multiple returns

```
def str_to_int(value):
    return int(value)


def string_adder(value):
    numbers = value.split(',')
    clean_numbers = []
    for number in numbers:
        clean_numbers.append(str_to_int(number))
    return sum(clean_numbers)
```

```
@patch('rock.dal.str_to_int')
def test_string_adder(s2i_mock):
    s2i_mock.side_effect = [2, 3]
    result = string_adder("2,3")
    assert result == 5
    assert s2i_mock.mock_calls == [call("2"), call("3")]
```

parametrized testing

```
@pytest.mark.parametrize("test_input,expected", [("3", 3), ("-1", -1)])
def test_str_to_int(test_input, expected):
    assert str_to_int(test_input) == expected
```


GET