You Vs. The Real World: Testing With Fixtures

Kumar McMillan

-What is this real world?
-Anything can happen
-the type of world our python programs live in

Overview

- Why testing the "real world" is important
  - Upsides/ downsides
- Testing with the "fixture" module
  - How to use it
  - Why it was created, some history

-Feel free to interrupt if you have a question

What is a real world test?

- Integration test, behavior test, functional test, black box test
- Runs your program in the most “real” way possible
- Doesn’t know about implementation
- Not a unit test

-runs your program the most real way possible without knowing about implementation
-by contrast, a unit test tests the implementation of your program, the units that make up the whole

How would you test the real world?

- GET/ POST request
- main()
- assert output rendered something
- assert new artifacts exist

-make a GET or POST request if it's a web application
-call main with options and arguments if it's a command line script
-assert something happened
-a database was updated somehow

Why is this the first test you should write?

- Shopping cart as counter-example
  - You've unit tested the Cart
  - You've unit tested the Order object
  - still, the front page doesn't load
  - Money lost

-I'll suggest this is the first test you should write
-Money lost
-extreme case: you're fired, QA team fired

Why is this the first test you should write?

- An integration test might ...
  - Load the product page, add a product to a cart, and place an order
  - Assert that money was collected
  - Doesn't care about the Cart or Order objects
  - Test-driven development

-a lot of people don't understand test driven development because it makes no sense to start a project by testing a unit to implement a whole that doesn't even exist

Integration Tests vs. Unit Tests

An integration test proves the system works	A unit test proves "how" the system works
...lets the implementation evolve freely	...ensures the implementation is solid

-not saying a unit test is unimportant
-let's take a look at them side by side
-agile development
-birth of a project
-later stages of a project

An integration test needs fixtures

- Environment of your program
- Inputs to your program
- Data your program consumes
- The test's "setup" function

-specific to the test, this is the setup
-this is separate code from the test itself

Use cases for testing with fixtures...

- Test a user login (users in the database)
- Test a data import script (files to consume)
- Test a billing calculation process (payable events)
- Model logic / stored procedures
- re-produce a bug

-to grab the state that an application was in when a specific bug happened, you need to recreate that environment

Mock objects or fixtures?

- Fixtures are not mock objects
- Real objects and real data result in more accurate tests, better coverage
- Good reasons to use mock objects:
  - speed
  - a reliable resource
- "switchable" mock objects

-mock objects that can switch to real objects using environment variables
-this allows you to run a quick run of your test suite locally
-then use a continuous integration tool to schedule a longer, more thorough run after you've checked in

Enter “fixture”

- A python module for loading and referencing test data
  - provides an interface for loading tabular data into storage media
    - designed primarily for databases
- easy_install fixture
- code.google.com/p/fixture

-by tabular, meaning the idea of rows and columns, as in a database
-can find the source on google code

The idea

- Define data by subclassing DataSet
- Use a fixture object that knows how to load data sets
- Use a data instance to reference loaded data
- A DataSet subclass would point to a table
- Its inner classes each represent a row to be loaded

-in your test, you use a data instance...

A DataSet

```
class ClientData(DataSet):
    class joe:
        company=”Joe, Inc.”
        contact=”Joe The Client”
```

-here is a simple example of some client data, targeting a table named client in a database
-the inner class joe marks a row that will be inserted
-the name “joe” provides context for this row (something more meaningful then an ID number) for when we want to reference this data

Referencing DataSet values

```
class SiteData(DataSet):
    class joes_site:
        url=”joe.com”
        client_id = \
            ClientData.joe.ref(‘id’)
```

-bear with my hoky model here
-here is some site data with rows that link back to the client data via foreign key
-the reference to the id attribute is made by accessing the inner class object directly
-notice that id wasn’t defined in the dataset
-this is a lazy property fetched later on using the actual object that was loaded into the database

Inheriting rows

```
class SiteData(DataSet):
    class joes_site:
        url=”joe.com”
        client_id = \
            ClientData.joe.ref(‘id’)
    class bobs_site(joes_site):
        url=”joesbrotherbob.com”
```

-inheriting works how you’d think it would since these are just python classes
-this new site for bob is going to be managed by the same client, Joe
-keep in mind you can’t inherit primary keys; this is handled for you

Defining a fixture

```
db = SQLAlchemyFixture(
                env=mapped.classes.module,
                session=session,
                style=NamedDataStyle())
```

-next you define a module level, configuration object that knows how to load data sets
-the env is a module or dict where mapped classes would be fetched from
-a session is used to make the connection
-a style is used to translate data set names into mapped class names

Behind the scenes (mapped class example)

```
name = style.guess_storable_name(
                                ‘ClientData’)
Client = env.Client
row = Client()
row.company = “Joe, Inc.”
row.contact = ”Joe The Client”
row.save()
```

-pseudo code to shed some light on what’s happening
-this is of course just one way of loading data sets

But... the magic!

```
class AnythingData(DataSet):
    class Meta:
        storable=Client
        primary_key=['email']
    class joe:
        email="joe@joe.com"
        company="Joe, Inc."

db = SQLAlchemyFixture(session=s)
```

-the magic here is really just to make data sets transparent
-they know nothing about databases or mapped classes
-the fixture object is responsible for loading data sets into objects
-special inner class Meta can make a data set explicit
-set the storable object
-set an unusual primary key or composite key

DataTestCase

```
class TestSite(DataTestCase,
               unittest.TestCase):
    fixture = db
    datasets = [SiteData]

    def test_joes_site(self):
        Client.get(
          self.data.ClientData.joe.id)
```

-now you need a way to run your tests with loaded data
-several ways to do this
-example here using a mixin class designed for unittest.TestCase

@db.with_data (for nose)

```
@db.with_data(ClientData, SiteData)
def test_joes_site(data):
    Client.get(data.ClientData.joe.id)
```

-somewhat simpler decorator approach
-reads: test_joes_site using db with_data ClientData, SiteData
-notice that the test function grows a new attribute “data”, a reference to loaded data

nose

- not required
- easy_install nose
- `nosetests`

- discovers tests and runs them
- code.google.com/p/python-nose/

-can come back to this if there are questions afterwards

with db.data() as d

```
with db.data(SiteData) as d:
    assert Site.get(
                d.SiteData.joes_site.id)
```

-if you're in python 2.5 or greater you can use the with statement
-a more natural, in line test
-reads: with db data SiteData as d, start testing with a reference to the data

Accessing Data

```
with db.data(SiteData,
             ClientData) as d:
    d.ClientData.joe.id
    d.ClientData.joe.company
    d.SiteData.joes_site.url
    d.SiteData.bobs_site.url
```

-covered most of this?

What media is supported?

- from fixture import SQLAlchemyFixture
- from fixture import SQLObjectFixture
- CSV?
- Django?

-sqlobject fixture takes a connection keyword instead of a session
-loading data sets into CSV files, say for testing a csv file parser
-been holding off on django support simply because there is a very active branch in django that adds a similar fixture interface

Regression testing with generated data

```
fixture my_sqlalchemy.table.foo \
        --dsn=”postgres://...” \
        --query=”id=1234”
```

-when you easy install fixture it also provides a command line tool named fixture
-sticking with sqlalchemy you can pass it a path to a table object or mapped class, send it a sql query and capture the results in a data set for testing
-when building on top of an application that already has a data model this comes in very handy

The “fixture” command

- Send it a “path” to an object
  - sqlalchemy: a Table, mapped class
  - SQLObject class
  - configure query parameters
  - you get DataSet code, foreign keys expanded
  - A complete “snapshot” of the query

–the code you get back cascades through all dependent foreign keys

Where fixture came from

- inspired by Ruby on Rails' fixtures
- python code, not YAML
- in 2005 created python module testtools.fixtures

- found many problems with the testtools.fixtures interface
- foreign keys, oh my

-I believe the rails interface was inspired by DbUnit for java or something similar
-I wanted to use python code; it was more flexible and made more sense to me
-the first interface was used quite a bit in my company
-a lot of headaches came from dealing with foreign keys

The new fixture

- fixture is a 2nd generation interface
- fixture attempts to be even more pythonic
- both testtools.fixtures and fixture developed for a large ETL test suite
- fixture still has little real world experience

-developed for a large ETL system, a tool that extracts, transforms and loads into a data warehouse
-many of the rules depend on an extensive data model of event classes, clients, partners, etc
-the new fixture still doesn't have much real world experience; haven't ported over too many tests to use it yet.

Where fixture is going

- In need of brave souls to incorporate fixture into their test suites
- submit issues to code.google.com/p/fixture
- In need of end-user documentation!

-most importantly people just need to start using the model
-I've actually started the docs a little while at pycon
-lots of doctests are in the code, possibly some helpful docstrings
END