Pytest workshop

Who

Ionel Cristian Mărieș

Gabriel Muj ← actual teacher

Workshop content

- preparation & setting up tox/virtualenv/django/pytest
- writing tests for django app (the tutorial polls app) while demonstrating
  - test discovery
  - classes vs function tests
  - assertion helpers
  - marks, skipping & xfailing
  - parametrization
  - fixtures, scoping, finalization

Ask anytime and anything. Ask for pauses.

Running the project: virtualenv

Linux:

```
$ virtualenv ve
$ . ve/bin/activate
$ pip install -e .
$ python manage.py migrate
$ python manage.py runserver
```

Windows (at least use clink):

```
> py -mpip install virtualenv
> py -mvirtualenv ve
> ve\Scripts\activate.bat
> pip install -e .
> python manage.py migrate
> python manage.py runserver
```

Running the project: tox

http://tox.rtfd.io

Linux:

```
$ pip install tox
$ tox -- django-admin migrate
$ tox -- django-admin runserver
```

Windows (at least use clink):

```
> py -mpip install tox
> tox -- django-admin migrate
> tox -- django-admin runserver
```

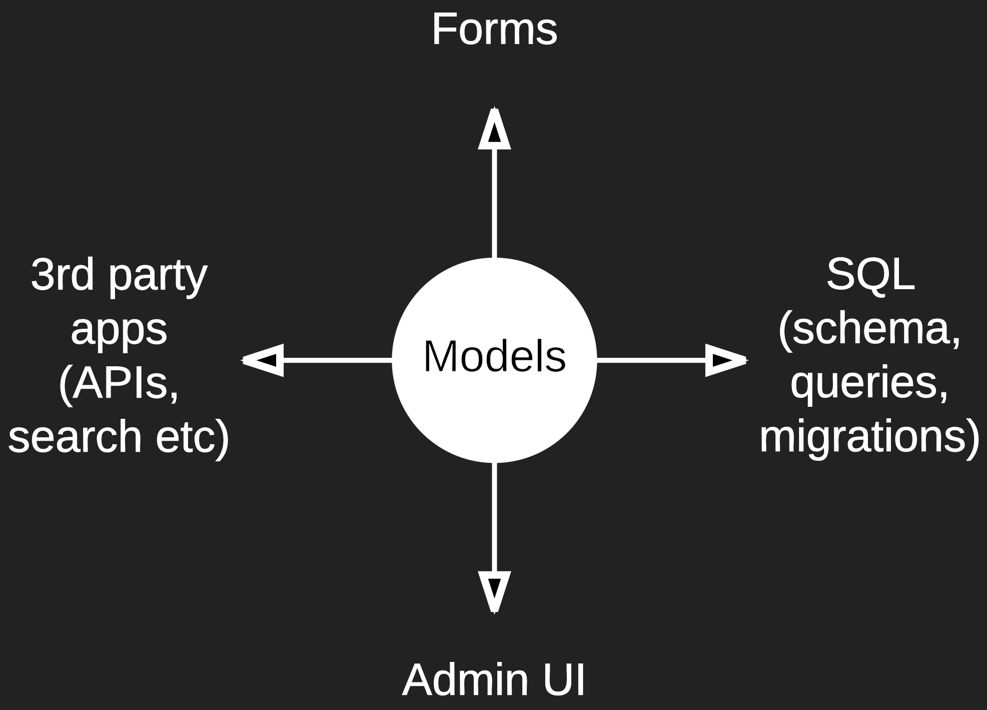
Django primer: management commands

Management commands:

Either through `manage.py` or `django-admin`:

- createsuperuser
- dbshell
- dumpdata
- loaddata
- makemigrations / migrate / showmigrations
- shell
- startapp / startproject
- runserver

Django primer: models


Forms
3rd party apps (APIs, search etc)
Models
SQL (schema, queries, migrations)
Admin UI

```python
from django.db import models



class Question(models.Model):
    question_text = models.CharField(max_length=200)
    pub_date = models.DateTimeField('date published')
```

Quick interlude: model magic

Ultra-simplified guts of Model/Form classes:

```
class Field:
    def __repr__(self):
        return 'Field(name={.name})'.format(self)

class Metaclass(type):
    def __new__(mcs, name, bases, attrs):
        fields = attrs.setdefault('fields', [])
        for name, value in attrs.items():
            if isinstance(value, Field):
                value.name = name; fields.append(value)
        return super(Metaclass, mcs).__new__(mcs, name, bases, attrs)

class Model(metaclass=Metaclass):
    a = Field()
    b = Field()


>>> print(MyModel.fields)
[Field(name=a), Field(name=b)]
```

Django primer: views

Views - two kinds:

1. Class-Based Views

```
class DetailView(generic.DetailView):
    model = Question
    template_name = 'polls/detail.html'
```

2. Function views

```
def vote(request, question_id):
    question = get_object_or_404(Question, pk=question_id)
    try:
        selected_choice = question.choice_set.get(pk=request.POST['choice'
    except (KeyError, Choice.DoesNotExist):
        return render(request, 'polls/detail.html', {
            'question': question,
            'error_message': "You didn't select a choice.",
        })
    else:
        selected_choice.votes += 1
        selected_choice.save()
```

Django primer: URLs

Views are mapped to URLs in `urls.py` files, eg:

- `mysite/urls.py`:

```
urlpatterns = [
    url(r'^', include('polls.urls')),
]
```

- `polls/urls.py`:

```
urlpatterns = [
    url(r'^(?P<pk>[0-9]+)/$', views.DetailView.as_view(), name='detail'),
    url(r'^(?P<question_id>[0-9]+)/vote/$', views.vote, name='vote'),
]
```

Django primer: templates

Templates automatically call and ignore missing attributes:

- `{{ foo.bar.missing }}` outputs nothing
- `{{ foo }}` calls foo if it's a callable (__call__)
- `{{ foo(1, 2, 3) }}` is not allowed (by design)
- `{{ foo|default:"}}" }}` is not possible (parser ain't very

```
<h1>{{ question.question_text }}</h1>

{% if error_message %}<p><strong>{{ error_message }}</strong></p>{% endif %}

<form action="{% url 'polls:vote' question.id %}" method="post">
{% csrf_token %}
{% for choice in question.choice_set.all %}
    <input type="radio" name="choice"
           id="choice{{ forloop.counter }}" value="{{ choice.id }}" />
    <label for="choice{{ forloop.counter }}">
        {{ choice.choice_text }}</label>
{% endfor %}
```

Tests

Some background:

- Django comes with own testing system, but it turns out `unittest.TestCase` ain't so good (in general).
    - There are three alternatives:

  - Nose (unmaintained)
  - Nose2 (unusable, it's missing almost all the Nose plugins)
  - Pytest

  Note that Nose is a fork of Pytest 0.8 (ancient, circa 2007)

Key features of pytest

Different way of test setup:

- Unittest uses setup/teardown methods. Inevitably that leads to multiple inheritance and mixins.
- Pytest uses composability and DI (dependency injection)

Different way of doing assertions:

- Unittest uses assertion methods. An army of `assertThis` and `assertThat`.
- Pytest uses simple assertions.

Key features of pytest

Different way of customizing behavior:

- Unittest makes it hard to customize collection, output and other handling. You end up subclassing and monkeypatching things.
- Pytest gives you hooks to customize almost anything. And it has builtin support for markers, selection, parametrization etc.

Note: there is some support for `unittest.TestCase` in pytest.

Pytest basics

Install it:

```
$ pip install pytest
```

Make a `tests\test_example.py`:

```
def test_simple():
    a = 1
    b = 2
    assert a + b == 3
    assert a + b == 4
```

Pytest basics

```
$ pytest tests/
========================= test session starts ============================
platform linux -- Python 3.6.2, pytest-3.2.2, py-1.4.34, pluggy-0.4.0 --
plugins: django-3.1.2
collected 1 item

tests/test_example.py F
=============================== FAILURES =================================
_____________________________ test_simple ______________________________

    def test_simple():
        a = 1
        b = 2
        assert a + b == 3
>       assert a + b == 4
E       assert (1 + 2) == 4

tests/test_example.py:5: AssertionError
======================= 1 failed in 0.05 seconds =========================
```

Pytest basics

Useful option and defaults, use `pytest.ini` for them:

```
[pytest]
; now we can just run `pytest` instead of `pytest tests/`
testpaths = tests

; note that `test_*.py` and `*_test.py` are defaults
python_files =
    test_*.py
    *_test.py
    tests.py
addopts =
; extra verbose
    -vv

; show detailed test counts
    -ra

; stop after 10 failures
    --maxfail=10
```

Quick interlude: imports

Import system uses a list of paths (`sys.path`) to lookup.
CWD is implicitly added to `sys.path`.
There is a module/package distinction.
Versioned imports ain't supported.
If `sys.path = ["/var/foo", "/var/bar"]` then:

- `/var/foo/module.py` - a module
- `/var/foo/package/__init__.py` - a package (`import package`)
- `/var/foo/package/module.py` - a module inside a package (`from package import module`)
- `/var/bar/module.py` - can't be imported, it's shadowed
- `/var/bar/package/extra.py` - can't be imported, its package

Pytest: test collection

Pytest has a file-based test collector:

- you give it a path
- it finds all the `test_*.py` files
- it messes up `sys.path` a bit: adds all the test roots into it

Suggested layout (flat, `tests` ain't a package, but everything in it is):

```
tests/
|-- foo\
|   |-- __init__.py
|   `-- test_foo.py
`-- test_bar.py
```

Pytest: fixtures

Not to be confused with (data) fixtures from Django (the result of `dumpdata` command).

```
@pytest.fixture
def myfixture(request):
    print('myfixture: do some setup')
    yield [1, 2, 3]
    print('myfixture: do some teardown')

@pytest.fixture
def mycomplexfixture(request, myfixture):
    print('myfixture: do some setup')
    yield myfixture + [4, 5]
    print('myfixture: do some teardown')

def test_fixture(myfixture):
    assert myfixture == [1, 2, 3]

def test_complexfixture(mycomplexfixture):
    assert myfixture == [1, 2, 3, 4, 5]
```

Quick interlude: simple DI implementation

```python
import functools, inspect
REGISTRY = {}
def dependency(func):
    REGISTRY[func.__name__] = func
def inject(func):
    sig = inspect.signature(func)
    for arg in sig.parameters:
        func = functools.partial(func, REGISTRY[arg]())
    return func

@dependency
def dep1():
    return 123
@dependency
def dep2():
    return 345
@inject
def fn(dep1, dep2):
    print(dep1, dep2)
```

```
>>> fn()
123 345
```

Pytest: fixture scoping

```
@pytest.fixture(scope="function", autouse=False)
def myfixture(request):
    ...
```

`scope` controls when and for how long the fixture is alive:

- `scope="function"` - default, fixture is created and teared down for every test.
- `scope="module"` - fixture is created for every module.
- `scope="session"` - fixture is created once.

`autouse` is for situations where you don't want to explicitly request the fixture for every test.

Pytest: markers

Are applied using decorators, eg:

```
@pytest.mark.skipif('platform.system() == "Windows"')
def test_nix_stuff():
    ...

@pytest.mark.mymark
def test_stuff():  # can select this later by runing pytest -m mymark
    ...

@pytest.mark.xfail('platform.system() == "Windows"', strict=True)
def test_shouldnt_work_on_windows():  # fail if it passes
    ...

@pytest.mark.skip
def test_deal_with_it_later():
    ...
```

Pytest: helpers

An alternative to the `skip` marker:

```
def test_deal_with_it_later():
    pytest.skip()
```

An alternative to the `skipif` marker (sometimes):

```
def test_linux_stuff():
    pytest.importorskip('signalfd')
```

The `raises` context manager:

```
def test_stuff():
    with raises(TypeError, match='Expected FooBar, not .*!'):
        raise TypeError('Expected FooBar, not asdf!')

    with raises(TypeError) as exc_info:
        raise TypeError('Expected FooBar, not asdf!')
    assert exc_info.value.startswith('Expected FooBar')
```

Pytest: parametrization

```
@pytest.mark.parametrize(['a', 'b'], [
    (1, 2),
    (2, 1),
])

def test_param(a, b):
    assert a + b == 3
```

```
collected 2 items

tests/test_param.py::test_param[1-2] PASSED
tests/test_param.py::test_param[2-1] PASSED
```

Pytest: parametrized fixtures

```python
@pytest.fixture(params=[len, max])
def func(request):
    return request.param

@pytest.mark.parametrize('numbers', [
    (1, 2),
    (2, 1),
])
def test_func(numbers, func):
    assert func(numbers) == 2
```

```
tests/test_param.py::test_func[func0-numbers0] PASSED
tests/test_param.py::test_func[func0-numbers1] PASSED
tests/test_param.py::test_func[func1-numbers0] PASSED
tests/test_param.py::test_func[func1-numbers1] PASSED
```

Pytest: parametrized fixtures

```python
@pytest.fixture(params=[len, max],
                ids=['len', 'max'])
def func(request):
    return request.param

@pytest.mark.parametrize('numbers', [
    (1, 2),
    (2, 1),
], ids=["white", "black"])
def test_func(numbers, func):
    assert func(numbers)
```

```
tests/test_param.py::test_func[len-white] PASSED
tests/test_param.py::test_func[len-black] PASSED
tests/test_param.py::test_func[max-white] PASSED
tests/test_param.py::test_func[max-black] PASSED
```

Pytest: test selection

We can select tests based on the parametrization:

```
$ pytest -k white -v
```

```
======================== test session starts ============================
platform linux -- Python 3.6.2, pytest-3.2.2, py-1.4.34, pluggy-0.4.0 --
cachedir: .cache
plugins: django-3.1.2
collected 9 items

tests/test_example.py::test_func[sum-white] PASSED
tests/test_example.py::test_func[len-white] PASSED
tests/test_example.py::test_func[max-white] PASSED
tests/test_example.py::test_func[min-white] PASSED

========================= 5 tests deselected ============================
================ 4 passed, 5 deselected in 0.07 seconds ================
```

Pytest: hooks

For now ... all you need to know about hooks:

- you can implement hooks in a `conftest.py` or a pytest plugin
- you put `conftest.py` files alongside your tests
- if there's a function that starts with `pytest_` - it's probably a hook.

Also, you put fixtures in your `conftest.py` (to use them in multiple test files)

We can talk all day long about hooks but we have to write those tests!

Pytest and Django

Install the plugin:

```
$ pip install pytest-django
```

Unfortunately it doesn't go through `manage.py` so we need to specify the settings module in `pytest.ini`:

```
[pytest]
DJANGO_SETTINGS_MODULE = mysite.settings
```

The `client` fixture

The `client` fixture makes an instance of django.test.Client.

Make a `tests/test_views.py`:

```python
def test_index_view_no_question(client, db):
    response = client.get('/')
    assert response.status_code == 200

    # use these in moderation (coupling)
    assert list(response.context_data['latest_question_list']) == []

    # a better assertion (end-to-end style):
    assert 'No polls are available.' in response.content.decode(
        response.charset)
    # if you use python 2 you can just do
    assert 'No polls are available.' in response.content
```

Technically these are not *"end to end"* tests but they are reasonably close for most apps.

What's with the decode?

The Unicode sandwich

$$\begin{tikzcd}
 & 1001110101010110010101 & \\
\boxed{\text{decode}} & & \textit{input} \\
\text{Unicode} \arrow[rr] & & \text{Unicode} \arrow[d] \\
\text{Unicode} \arrow[d] & & \text{Unicode} \arrow[ll] \\
\text{Unicode} \arrow[rr] & & \text{Unicode} \\
\textit{output} & & \boxed{\text{encode}} \\
 & 1001110101010110010101 &
\end{tikzcd}$$

See: https://nedbatchelder.com/text/unipain/unipain.html#35

Making a fixture for questions

```
from django.utils import timezone

@pytest.fixture
def question(db):
    return Question.objects.create(
        question_text="What is love?",
        pub_date=timezone.now()
    )

def test_index_view_one_question(client, question):
    response = client.get('/')
    assert response.status_code == 200
    # list cause it's an QuerySet
    assert list(response.context_data['latest_question_list']) == [
        question]
    # how much markup to include?
    assert 'href="/polls/1/">What is love?</a>' in response.content.decode(
        response.charset)
```

*

Pragmatic testing

1. write code
2. do some manual or sloppy tests
3. rewrite code cause it was a terrible terrible idea
4. a cycle of: write tests, find bugs, figure out what's untested

A cynic might add:

5. rewrite more code, suffer cause tests are too coupled with code
6. find more bugs, suffer cause tests are too lose

Having more question objects

We can't require a fixture more than once, thus:

```
@pytest.fixture
def question_factory(db):
    now = timezone.now()
    def create_question(question_text, pub_date_delta=timedelta()):
        return Question.objects.create(
            question_text=question_text,
            pub_date=now + pub_date_delta
        )
    return create_question

def test_index_view_two_questions(client, question_factory):
    question1 = question_factory("Question 1")
    question2 = question_factory("Question 2", -timedelta(hours=1))
    response = client.get('/')
    assert response.status_code == 200
    assert list(response.context_data['latest_question_list']) == [
        question1, question2]
    content = response.content.decode(response.charset)
    assert '/polls/1/' in content
```

Having tons of questions

Note that the view is set to only display the last 5 questions, thus:

```
def test_index_view_only_last_five_questions(client, question_factory):
    questions = [
        question_factory("Question {}".format(i), -timedelta(hours=i))
        for i in range(1, 10)
    ]

    response = client.get('/')
    assert response.status_code == 200
    assert list(
        response.context_data['latest_question_list']
    ) == questions[:5]

    content = response.content.decode(response.charset)
    for i in range(1, 6):
        assert 'href="/polls/{0}/">Question {0}</a>'.format(i) in content
    assert 'Question 6' not in content
```

Having future questions

Questions in the future shouldn't be displayed, thus:

```
def test_index_view_exclude_question_published_in_future(client,
                                                          question_factory):
    question_factory("Question 1", timedelta(hours=1))

    response = client.get('/')
    assert response.status_code == 200
    assert list(response.context_data['latest_question_list']) == []
    assert 'Question 1' not in response.content.decode(response.charset)
```

Bogus ids

Proper response should be returned on bogus IDs:

```
def test_detail_view_question_not_found(client, db):
    response = client.get('/999/')
    assert response.status_code == 404

def test_vote_question_not_found(client, db):
    response = client.get('/999/vote/')
    assert response.status_code == 404

def test_results_view_question_not_found(client, db):
    response = client.get('/999/results/')
    assert response.status_code == 404
```

Dealing with bad questions

Questions that don't have any answers, of course!

```
def test_detail_view_question_found(client, question):
    response = client.get('/%s/' % question.id)
    assert response.status_code == 200
    assert 'What is love?' in response.text_content
    assert 'Someone needs to figure out some answers!' \
        in response.text_content

    # assertions you'll be sorry for (coupling!)
    assert response.context_data['object'] == question
    assert 'polls/detail.html' in response.template_name
```

Isn't the client fixture a bit annoying?

It sure is, so lets fix it:

```python
@pytest.fixture
def client(client):
    func = client.request

    def wrapper(**kwargs):
        # instead of throwing prints all over the place
        print('>>>>', ' '.join('{}={!r}'.format(*item)
                               for item in kwargs.items()))
        resp = func(**kwargs)
        print('<<<<', resp, resp.content)
        # also, decode the content
        resp.text_content = resp.content.decode(resp.charset)
        # why not patch resp.content? well ...
        return resp

    client.request = wrapper
    return client
```

Watch the scope when patching stuff. In this case it was fine (`pytest_django.client` had the same scope - `"function"`).

Creating some answers

```
@pytest.fixture
def question_choice_factory(db):
    def create_question_choice(question, choice_text, votes=0):
        return Choice.objects.create(question=question,
                                     choice_text=choice_text,
                                     votes=votes)
    return create_question_choice


def test_vote_question_found_with_choice(client, question,
                                         question_choice_factory):
    choice1 = question_choice_factory(question, "Choice 1", votes=0)

    response = client.post('/%s/vote/' % question.id,
                           data={"choice": choice1.id})
    assert response.status_code == 302
    assert response.url == '/%s/results/' % (question.id,)

    choice1.refresh_from_db()
```

Testing the results

We should check the result page too.

An easy way is to just slap on some extra assertions in the previous test:

```
def test_vote_question_found_with_choice(...):
    ...


    response = client.get('/%s/results/' % question.id)
    assert '<li>Choice 1 -- 1 vote</li>' in response.text_content
```

The disadvantage is that test becomes bulky and debugging may be harder.

Guess what's missing, template has this:

```
{% for choice in question.choice_set.all %}
    <li>{{ choice.choice_text }} --
        {{ choice.votes }} vote{{ choice.votes|pluralize }}</li>
{% endfor %}
```

Testing the results

Problems with newlines?

An alternative is regexes but lets unpack this first:

```
assert re.findall(r'<li>Choice 1\s+--\s+1 vote</li>',
                  response.text_content)
```

- `re.findall` mean find all matches anywhere (don't fall for `re.match` - it matches at the start of the string)
- `r'foo\bar'` means no escapes (same as `'foo\\bar'`)
- `\s` means (in regex parlance) any space (same as `'[ \t\n\r\f\v]'` plus the damned Unicode whitespace characters)
- `+` is a qualifier, it means "one or more"
- `\s+` means "one of more space characters"

Testing bad requests

Test what happens when there's no form data:

```python
def test_vote_question_found_no_choice(client, question):
    response = client.post('/%s/vote/' % question.id)
    assert response.status_code == 200
    content = response.content.decode(response.charset)
    assert 'What is love?' in content
    assert "You didn&#39;t select a choice." in content
```

Getting ideas about missing tests

Suggested use:

```
$ pip install pytest-cov
$ pytest --cov=. --cov-report=term-missing --cov-branch
```

Alternatively, create a `.coveragerc`:

```
[run]
branch = true
source = src

[report]
show_missing = true
precision = 2
```

With that it's simpler to run, just:

```
$ pytest --cov
```

Note: having 100% coverage doesn't mean you have tested everything. But if you don't you probably haven't.

More on coverage: ignoring irrelevant stuff

In `.coveragerc`:

```
[report]
omit =
    *apps.py
    *manage.py
    *wsgi.py
```

Alternative, have these on the lines that don't need to be covered:

```
stuff_that_is_not_frequently_used()  # pragma: no cover
```

Browser tests with pytest-splinter (optional)

Get the right binary from:

https://github.com/mozilla/geckodriver/releases

Put it in CWD.

```
def test_index(pages, browser, live_server):
    browser.visit(live_server + '/')
    assert browser.is_text_present('Foo')
```

Explore api at: http://splinter.rtfd.io