Up and Running with `pytest`, Hypothesis and `tox`

Jamie Bayne

28 January 2020

Refresher: what is unit testing?

Superset of
- Example-based testing
- Property-based testing

Distinct from
- Integration testing
- Fuzz-testing (usually)
- Linting/Static analysis

Refresher: what is unit testing?

A unit test

- tests one[1] "thing"
- has no dependencies
- Pass or Fail
- is automated

What is a thing? A property or behaviour we expect from a function.

NOT "didn't throw/error" – too broad! But can test that it *does* throw something.

[1] unit: one

Typical unit test:

```
def test_foo_calcsum_sums_floatlist():
    # Set up one example
    foo = Foo(...)
    test_list = [1., 6.343, -100., 1e-4, 1e5]

    # Run one function
    result = foo.calcsum(test_list)

    # Assert one expected property
    assertEquals(result, sum(test_list))
```

Have we fully tested `Foo.calcsum`?

Python Test Libraries

`pytest` Pythonic no-API unit-tests

Hypothesis *property-based testing*

`tox` test-runner: multiple Python versions and isolated environments

Why not `unittest`[2], `nose2`? ***Unpythonic***

[2] JUnit-based framework provided in the standard library

Setup

Debian/Ubuntu

```
apt install python3-{pytest,hypothesis,tox}
```

Arch/MSYS

```
pacman -S python-{pytest,hypothesis,tox}
```

pip

```
pip install pytest hypothesis tox
```

Anaconda

```
conda install -n [env] -c conda-forge pytest hypothesis tox
```

pytest

pytest

```python
# ./test/test_all.py
def mul1(x):
    return x*1

def test_mul1_identity():
    assert mul1(10) == 10
```

```
$ pytest
================== test session starts ==================
platform linux -- Python 3.8.1, pytest-5.3.4, py-1.8.1, pluggy-0.13.1
rootdir: /home/jamie/pytesttest
plugins: hypothesis-4.54.2
collected 1 item

test/test_all.py .                                  [100%]

================== 1 passed in 0.01s ====================
```

Splitting things up

Example project hierarchy

```
project
- setup.py
- my_pkg
    __init__.py
    module.py
- tests
    test_A.py
    test_B.py
    testconf.py
```

Splitting things up

Example project files

```
#my_pkg/__init__.py
def mul1(x):
    return x*1
```

—

```
#test/test_all.py
import my_pkg

def test_mul1_identity():
    assert my_pkg.mul1(10) == 10
```

Running our project

Runnning `pytest` again, we get... a big fat error.

Solution:

```
$ python -m pytest
================== test session starts ==================
platform linux -- Python 3.8.1, pytest-5.3.4, py-1.8.1, pluggy-0
rootdir: /home/jamie/pytesttest
plugins: hypothesis-4.54.2
collected 1 item

test/test_all.py .                                  [100%]

=================== 1 passed in 0.02s ===================
```

Questions?

Meeting test dependencies with fixtures

```
from pytest import fixture

@fixture()
def context():
    return {"a": 7,
            "b": "hello",
            "foo": [1,2,3]}

def test_frobnicator(context):
    assert frobnicate(context) is not None
    assert context["a"] == 7
```

Scoped fixtures

Expensive fixtures can be scoped: `"function"`, `"class"`, `"module"`, `"package"`[3] or `"session"`.

```python
@fixture(scope="session")
def image():
    # Only called once per pytest execution
    return imread("data/apples.png")

def test_count_apples(image):
    assert count_apples(image) == 4
```

[3] experimental

Fixture teardown

Simply `yield` the result[4], and clean up after.

```
@fixture()
def read_from_singleton():
    evil_singleton.instance.read_flag = True
    yield
    evil_singleton.instance.read_flag = False
```

[4] here, we yield nothing

Fixture RAII with `with`

Even better, you can use `with` and `yield` for objects that support it:

```
@fixture()
def csv_file():
    with open("data/test_data.csv") as f:
        yield f
```

Parameterised fixtures

```
@fixture(parameters=["s3://test-bucket.amazonaws.com",
                     "gs://test-project/"])
def bucket_handle(uri):
    # Imaginary "BucketHandle" class
    return BucketHandle(uri)

def test_load(bucket_handle):
    f = bucket_handle.load("test_file")
    assert f.read() == "test string"
```

Questions?

Hypothesis

Problem

We have numeric `func(x)` which must pass

```
assert func(x) != 0
```

How do we adequately test `x`?

- `test_func1`, `test_func2`, `test_func3`, ...
- Fixture parameters?
- Manual search?

Need something smarter...

Hypothesis: Property-based Testing in Python

$f(x) \neq 0$ must hold "for all x in X". What is X?

Could be a type `str`, `np.uint8`,

```
namedtuple("Rect", ["x","y","w","h"])
```

Could be a mathematical construct $\mathbb{N}$, $\mathbb{P}(\mathbb{R})$,
$2k \ \forall k \in \mathbb{Z} - \{0\}$

Could be an arbitrary set of constraints

$r \in \texttt{Rect}, \text{area}(r) < 10$

Hypothesis constructs a *search strategy* for X.

Hypothesis: Property-based Testing in Python

Hypothesis strategies generate test-cases:

```
from my_pkg import func
from hypothesis import given
from hypothesis.strategies import integers

@given(integers())
def test_func_must_be_nonzero(i):
    assert func(i) != 0
```

Many built-in strategies. Your best friend:
https://hypothesis.readthedocs.io/en/latest/data.html

Hypothesis: Property-based Testing in Python

Hypothesis finds a failure case, then *simplifies it* for us:

```
$ python -m pytest

...

Falsifying example: test_func_must_be_nonzero(
    i=-1,
)
```

Worked example

More complicated data – `lists`

Hypothesis has strategies for container types: `lists`, `iterables`, *&c.*[5]

They take a strategy as an argument:

```
from my_pkg import func
from hypothesis import given
from hypothesis.strategies import lists, integers

@given(lists(integers(), max_size=1000))
def test_func_vectorised_must_be_nonzero(int_list):
    assert not any(x == 0 for x in func(int_list))
```

[5] Most exciting: NumPy `arrays` strategy.

More complicated data – `builds`

We have a class `Foo(int, str)` - how can we use Hypothesis?

```
from hypothesis import given
from hypothesis.strategies import builds, integers, text

foos = builds(Foo, integers(), text())

@given(foos)
def test_foo_property(foo): pass
```

—

Or if `Foo(i: int, s: str)`,

```
foos = builds(Foo)
```

More complicated data – `composite`

We have a class `Foo(int, str)`. What if the int gives the maximum length of the string?

```
from hypothesis import given, example
from hypothesis.strategies import composite, integers, text

@composite
def foos(draw):
    i = draw(integers())
    s = draw(text(max_size=i))
    return Foo(i,s)

@given(foos)
def test_foo_property(foo): pass
```

Questions?

tox

One step automation

Run your tests on Python versions 2.7, 3.6 and 3.8:

```
# tox.ini
[tox]
envlist = py27,py36,py38

[testenv]
deps =
    pytest
    hypothesis
commands = python -m pytest
```

One (and a half) step automation

`tox` reads our `setup.py` to run tests

```
# setup.py
from setuptools import setup, find_packages
setup(
    name="my_pkg",
    version="0.1",
    packages=find_packages(),
)
```

One step execution

On my machine:

```
$ tox

... errors

ERROR:   py27: commands failed
ERROR:  py36: InterpreterNotFound: python3.6
  py38: commands succeeded
```

Need Python versions installed on system PATH!

Fin

Questions?

Revision: https://jamiebayne.co.uk/blog/pytest