py.test All Versions

Examples

Setting up py.test

py.test is one of several third party testing libraries that are available for Python. It can be installed using pip with

```
pip install pytest
```

The Code to Test

Say we are testing an addition function in projectroot/module/code.py :

```
# projectroot/module/code.py
def add(a, b):
    return a + b
```

The Testing Code

We create a test file in projectroot/tests/test_code.py . The file **must begin with test_** to be recognized as a testing file.

```
# projectroot/tests/test_code.py
from module import code


def test_add():
    assert code.add(1, 2) == 3
```

Running The Test

From projectroot we simply run py.test :

```
# ensure we have the modules
$ touch tests/__init__.py
$ touch module/__init__.py
$ py.test
============================================= test session starts ==========================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py .

========================================== 1 passed in 0.01 seconds ====================
```

Intro to Test Fixtures

More complicated tests sometimes need to have things set up before you run the code you want to test. It is possible to do this in the test function itself, but then you end up with large test functions doing so much that it is difficult to tell where the setup stops and the test begins. You can also get a lot of duplicate setup code between your various test functions.

Our code file:

```
# projectroot/module/stuff.py
class Stuff(object):
    def prep(self):
        self.foo = 1
        self.bar = 2
```

Our test file:

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def test_foo_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000
```

```
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

These are pretty simple examples, but if our `Stuff` object needed a lot more setup, it would get unwieldy. We see that there is some duplicated code between our test cases, so let's refactor that into a separate function first.

```
# projectroot/tests/test_stuff.py
import pytest
from module import stuff


def get_prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff


def test_foo_updates():
    my_stuff = get_prepped_stuff()
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates():
    my_stuff = get_prepped_stuff()
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

This looks better but we still have the `my_stuff = get_prepped_stuff()` call cluttering up our test functions.

py.test fixtures to the rescue!

Fixtures are much more powerful and flexible versions of test setup functions. They can do a lot more than we're leveraging here, but we'll take it one step at a time.

First we change `get_prepped_stuff` to a fixture called `prepped_stuff` . You want to name your fixtures with nouns rather than verbs because of how the fixtures will end up being used in the test functions themselves later. The `@pytest.fixture` indicates that this specific function should be handled as a fixture rather than a regular function.

```
@pytest.fixture
def prepped_stuff():
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    return my_stuff
```

Now we should update the test functions so that they use the fixture. This is done by adding a parameter to their definition that exactly matches the fixture name. When py.test executes, it will run the fixture before running the test, then pass the return value of the fixture into the test function through that parameter. (Note that fixtures don't **need** to return a value; they can do other setup things instead, like calling an external resource, arranging things on the filesystem, putting values in a database, whatever the tests need for setup)

```
def test_foo_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 1 == my_stuff.foo
    my_stuff.foo = 30000
    assert my_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    my_stuff = prepped_stuff
    assert 2 == my_stuff.bar
    my_stuff.bar = 42
    assert 42 == my_stuff.bar
```

Now you can see why we named it with a noun. but the `my_stuff = prepped_stuff` line is pretty much useless, so let's just use `prepped_stuff` directly instead.

```
def test_foo_updates(prepped_stuff):
    assert 1 == prepped_stuff.foo
    prepped_stuff.foo = 30000
    assert prepped_stuff.foo == 30000


def test_bar_updates(prepped_stuff):
    assert 2 == prepped_stuff.bar
    prepped_stuff.bar = 42
```

```
    prepped_stuff.bar = 42
    assert 42 == prepped_stuff.bar
```

Now we're using fixtures! We can go further by changing the scope of the fixture (so it only runs once per test module or test suite execution session instead of once per test function), building fixtures that use other fixtures, parametrizing the fixture (so that the fixture and all tests using that fixture are run multiple times, once for each parameter given to the fixture), fixtures that read values from the module that calls them... as mentioned earlier, fixtures have a lot more power and flexibility than a normal setup function.

Cleaning up after the tests are done.

Let's say our code has grown and our Stuff object now needs special clean up.

```
# projectroot/module/stuff.py
class Stuff(object):
def prep(self):
    self.foo = 1
    self.bar = 2

def finish(self):
    self.foo = 0
    self.bar = 0
```

We could add some code to call the clean up at the bottom of every test function, but fixtures provide a better way to do this. If you add a function to the fixture and register it as a **finalizer** , the code in the finalizer function will get called after the test using the fixture is done. If the scope of the fixture is larger than a single function (like module or session), the finalizer will be executed after all the tests in scope are completed, so after the module is done running or at the end of the entire test running session.

```
@pytest.fixture
def prepped_stuff(request):  # we need to pass in the request to use finalizers
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    def fin():  # finalizer function
        # do all the cleanup here
        my_stuff.finish()
    request.addfinalizer(fin)  # register fin() as a finalizer
    # you can do more setup here if you really want to
    return my_stuff
```

Using the finalizer function inside a function can be a bit hard to understand at first glance, especially when you have more complicated fixtures. You can instead use a **yield fixture** to do the same thing with a more human readable execution flow. The only real difference is that instead of using `return` we use a `yield` at the part of the fixture where the setup is done and control should go to a test function, then add all the cleanup code after the `yield` . We also decorate it as a `yield_fixture` so that py.test knows how to handle it.

```
@pytest.yield_fixture
def prepped_stuff():  # it doesn't need request now!
    # do setup
    my_stuff = stuff.Stuff()
    my_stuff.prep()
    # setup is done, pass control to the test functions
    yield my_stuff
    # do cleanup
    my_stuff.finish()
```

And that concludes the Intro to Test Fixtures!

For more information, see the official py.test fixture documentation and the official yield fixture documentation

---

Failing Tests

A failing test will provide helpful output as to what went wrong:

```
# projectroot/tests/test_code.py
from module import code


def test_add__failing():
    assert code.add(10, 11) == 33
```

Results:

```
$ py.test
============================================== test session starts ==========================
platform darwin -- Python 2.7.10, pytest-2.9.2, py-1.4.31, pluggy-0.3.1
rootdir: /projectroot, inifile:
collected 1 items

tests/test_code.py F
```

```
=================================================== FAILURES ===================================
_____________________________________________ test_add__failing ______________________________

    def test_add__failing():
>       assert code.add(10, 11) == 33
E       assert 21 == 33
E        +  where 21 = <function add at 0x105d4d6e0>(10, 11)
E        +    where <function add at 0x105d4d6e0> = code.add

tests/test_code.py:5: AssertionError
============================================== 1 failed in 0.01 seconds ========================
```

Syntax

Parameters

Remarks