unittest.TestCase Support

`pytest` supports running Python `unittest`-based tests out of the box. It's meant for leveraging existing `unittest`-based test suites to use pytest as a test runner and also allow to incrementally adapt the test suite to take full advantage of pytest's features.

To run an existing `unittest`-style test suite using `pytest`, type:

```
pytest tests
```

pytest will automatically collect `unittest.TestCase` subclasses and their `test` methods in `test_*.py` or `*_test.py` files.

Almost all `unittest` features are supported:

- `@unittest.skip` style decorators;
- `setUp/tearDown`;
- `setUpClass/tearDownClass()`;

Up to this point pytest does not have support for the following features:

- load_tests protocol;
- setUpModule/tearDownModule;
- subtests;

Benefits out of the box

By running your test suite with pytest you can make use of several features, in most cases without having to modify existing code:

- Obtain more informative tracebacks;
- stdout and stderr capturing;
- Test selection options using `-k` and `-m` flags;
- Stopping after the first (or N) failures;
- –pdb command-line option for debugging on test failures (see note below);
- Distribute tests to multiple CPUs using the pytest-xdist plugin;
- Use plain assert-statements instead of `self.assert*` functions (unittest2pytest is immensely helpful in this);

pytest features in `unittest.TestCase` subclasses

The following pytest features work in `unittest.TestCase` subclasses:

- Marks: skip, skipif, xfail;
- Auto-use fixtures;

The following pytest features do not work, and probably never will due to different design philosophies:

- Fixtures (except for `autouse` fixtures, see below);
- Parametrization;
- Custom hooks;

Third party plugins may or may not work well, depending on the plugin and the test suite.

Mixing pytest fixtures into `unittest.TestCase` subclasses using marks

Running your unittest with `pytest` allows you to use its fixture mechanism with `unittest.TestCase` style tests. Assuming you have at least skimmed the pytest fixture features, let's jump-start into an example that integrates a pytest `db_class` fixture, setting up a class-cached database object, and then reference it from a unittest-style test:

```
# content of conftest.py

# we define a fixture function below and it will be "used" by
# referencing its name from tests

import pytest

@pytest.fixture(scope="class")
def db_class(request):
    class DummyDB(object):
        pass
    # set a class attribute on the invoking test context
    request.cls.db = DummyDB()
```

This defines a fixture function `db_class` which - if used - is called once for each test class and which sets the class-level `db` attribute to a `DummyDB` instance. The fixture function achieves this by receiving a special `request` object which gives access to the requesting test context such as the `cls` attribute, denoting the class from which the fixture is used. This architecture decouples fixture writing from actual test code and allows re-use of the fixture by a minimal reference, the fixture name. So let's write an actual `unittest.TestCase` class using our fixture definition:

```
# content of test_unittest_db.py

import unittest
import pytest

@pytest.mark.usefixtures("db_class")
class MyTest(unittest.TestCase):
    def test_method1(self):
        assert hasattr(self, "db")
        assert 0, self.db   # fail for demo purposes

    def test_method2(self):
        assert 0, self.db   # fail for demo purposes
```

The `@pytest.mark.usefixtures("db_class")` class-decorator makes sure that the pytest fixture function `db_class` is called once per class. Due to the deliberately failing assert statements, we can take a look at the `self.db` values in the traceback:

```
$ pytest test_unittest_db.py
======= test session starts ========
platform linux -- Python 3.x.y, pytest-3.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 2 items

test_unittest_db.py FF

======= FAILURES ========
_______ MyTest.test_method1 ________

self = <test_unittest_db.MyTest testMethod=test_method1>

    def test_method1(self):
        assert hasattr(self, "db")
>       assert 0, self.db   # fail for demo purposes
E       AssertionError: <conftest.db_class.<locals>.DummyDB object at 0xdeadbeef>
E       assert 0

test_unittest_db.py:9: AssertionError
_______ MyTest.test_method2 ________

self = <test_unittest_db.MyTest testMethod=test_method2>
```

v: latest

```
    def test_method2(self):
>       assert 0, self.db   # fail for demo purposes
E       AssertionError: <conftest.db_class.<locals>.DummyDB object at 0xdeadbeef>
E       assert 0

test_unittest_db.py:12: AssertionError
======= 2 failed in 0.12 seconds ========
```

This default pytest traceback shows that the two test methods share the same `self.db` instance which was our intention when writing the class-scoped fixture function above.

Using autouse fixtures and accessing other fixtures

Although it's usually better to explicitly declare use of fixtures you need for a given test, you may sometimes want to have fixtures that are automatically used in a given context. After all, the traditional style of unittest-setup mandates the use of this implicit fixture writing and chances are, you are used to it or like it.

You can flag fixture functions with `@pytest.fixture(autouse=True)` and define the fixture function in the context where you want it used. Let's look at an `initdir` fixture which makes all test methods of a `TestCase` class execute in a temporary directory with a pre-initialized `samplefile.ini`. Our `initdir` fixture itself uses the pytest builtin tmpdir fixture to delegate the creation of a per-test temporary directory:

```python
# content of test_unittest_cleandir.py
import pytest
import unittest

class MyTest(unittest.TestCase):

    @pytest.fixture(autouse=True)
    def initdir(self, tmpdir):
        tmpdir.chdir() # change to pytest-provided temporary directory
        tmpdir.join("samplefile.ini").write("# testdata")

    def test_method(self):
        with open("samplefile.ini") as f:
            s = f.read()
        assert "testdata" in s
```

Due to the `autouse` flag the `initdir` fixture function will be used for all methods of the class where it is defined. This is a shortcut for using a `@pytest.mark.usefixtures("initdir")` marker on the class like in the previous example.

Running this test module ...:

```
$ pytest -q test_unittest_cleandir.py
.
1 passed in 0.12 seconds
```

... gives us one passed test because the `initdir` fixture function was executed ahead of the `test_method`.

Note:

`unittest.TestCase` methods cannot directly receive fixture arguments as implementing that is likely to inflict on the ability to run general unittest.TestCase test suites.

The above `usefixtures` and `autouse` examples should help to mix in pytest fixtures into unittest suites.

You can also gradually move away from subclassing from `unittest.TestCase` to *plain asserts* and then start to benefit from the full pytest feature set step by step.

v: latest

Note:

Running tests from `unittest.TestCase` subclasses with `--pdb` will disable tearDown and cleanup methods for the case that an Exception occurs. This allows proper post mortem debugging for all applications which have significant logic in their tearDown machinery. However, supporting this feature has the following side effect: If people overwrite `unittest.TestCase` `__call__` or `run`, they need to to overwrite `debug` in the same way (this is also true for standard unittest).

v: latest