

Installation and Getting Started

Pythons: Python 2.6,2.7,3.3,3.4,3.5, Jython, PyPy-2.3

Platforms: Unix/Posix and Windows

PyPI package name: [pytest](#)

dependencies: [py](#), [colorama \(Windows\)](#), [argparse \(py26\)](#), [ordereddict \(py26\)](#).

documentation as PDF: [download latest](#)

Installation

Installation:

```
pip install -U pytest
```

To check your installation has installed the correct version:

```
$ pytest --version
```

```
This is pytest version 3.x.y, imported from $PYTHON_PREFIX/lib/python3.5/site-packages/pytest.py
```

Our first test run

Let's create a first test file with a simple test function:

```
# content of test_sample.py
def func(x):
    return x + 1

def test_answer():
    assert func(3) == 5
```

That's it. You can execute the test function now:

```
$ pytest
===== 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 1 item

test_sample.py F


===== FAILURES =====
_____ test_answer _____

    def test_answer():
>     assert func(3) == 5
E       assert 4 == 5
E         + where 4 = func(3)

test_sample.py:5: AssertionError
===== 1 failed in 0.12 seconds =====
```

We got a failure report because our little `func(3)` call did not return 5.

Note:

You can simply use the `assert` statement for asserting test expectations. [pytest's Advanced assertion introspection](#)  [v: latest](#) ▼

lignently report intermediate values of the assert expression freeing you from the need to learn the many names of JUnit legacy methods.

Running multiple tests

pytest will run all files in the current directory and its subdirectories of the form `test_*.py` or `*_test.py`. More generally, it follows standard test discovery rules.

Asserting that a certain exception is raised

If you want to assert that some code raises an exception you can use the `raises` helper:

```
# content of test_sysexit.py
import pytest
def f():
    raise SystemExit(1)

def test_mytest():
    with pytest.raises(SystemExit):
        f()
```

Running it with, this time in “quiet” reporting mode:

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

Grouping multiple tests in a class

Once you start to have more than a few tests it often makes sense to group tests logically, in classes and modules. Let’s write a class containing two tests:

```
# content of test_class.py
class TestClass(object):
    def test_one(self):
        x = "this"
        assert 'h' in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, 'check')
```

The two tests are found because of the standard Conventions for Python test discovery. There is no need to subclass anything. We can simply run the module by passing its filename:

```
$ pytest -q test_class.py
.F
===== FAILURES =====
_____ TestClass.test_two _____

self = <test_class.TestClass object at 0xdeadbeef>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, 'check')
E       AssertionError: assert False
E       + where False = hasattr('hello', 'check')

test_class.py:8: AssertionError
1 failed, 1 passed in 0.12 seconds
```

 v: latest ▼

The first test passed, the second failed. Again we can easily see the intermediate values used in the assertion, helping us to understand the reason for the failure.

Going functional: requesting a unique temporary directory

For functional tests one often needs to create some files and pass them to application objects. pytest provides [Builtin fixtures/function arguments](#) which allow to request arbitrary resources, for example a unique temporary directory:

```
# content of test_tmpdir.py
def test_needsfiles(tmpdir):
    print (tmpdir)
    assert 0
```

We list the name `tmpdir` in the test function signature and pytest will lookup and call a fixture factory to create the resource before performing the test function call. Let's just run it:

```
$ pytest -q test_tmpdir.py
F
===== FAILURES =====
_____ test_needsfiles _____

tmpdir = local('PYTEST_TMPDIR/test_needsfiles0')

    def test_needsfiles(tmpdir):
        print (tmpdir)
>       assert 0
E       assert 0

test_tmpdir.py:3: AssertionError
----- Captured stdout call -----
PYTEST_TMPDIR/test_needsfiles0
1 failed in 0.12 seconds
```

Before the test runs, a unique-per-test-invocation temporary directory was created. More info at [Temporary directories and files](#).

You can find out what kind of builtin [pytest fixtures: explicit, modular, scalable](#) exist by typing:

```
pytest --fixtures # shows builtin and custom fixtures
```

Where to go next

Here are a few suggestions where to go next:

- [Calling pytest through python -m pytest](#) for command line invocation examples
- [good practices](#) for virtualenv, test layout
- [Using pytest with an existing test suite](#) for working with pre-existing tests
- [pytest fixtures: explicit, modular, scalable](#) for providing a functional baseline to your tests
- [Writing plugins](#) managing and writing plugins