Pytest - Installation and Getting Started Guide

This document contains information to help you accomplish the following:

- install and get started with `pytest`
- create a first test
- run complex functional tests that can leverage your application or library
- request temporary directories for functional tests
- resources for learning

System requirements

`Pytest` is tested and supported on the following environments:

- **Platform**: Linux, Windows
- **Python version**: 3.5, 3.6, 3.7, PyPy 3
- **PyPI package name**: pytest

Download in PDF

Download document in PDF

1. Install `pytest`

1. Run the following command in your command line:

```
$ pip install -U pytest
```

2. Run the following command to ensure you have installed the correct version of `pytest`:

```
$ pytest —version
This is pytest version 4.x.y, imported from
$PYTHON_PREFIX/lib/python3.6/site-packages/pytest.py
```

2. Create your first test

You can create and run two types of tests:

- a simple test
- a complex test, which asserts a mathematical statement and fails the assertion for testing purposes

1. Using your preferred code-editor, create a file named `test_sample.py` with the following content:

```
# content of test_sample.py
Def func(x):
return x+1
def test_answer():
        assert func(3) == 5
```

2. Using the command line, run the following command to execute the test function:

```
$ pytest
=========================== test session starts
platform linux -- Python 3.x.y, pytest-4.x.y, py-1.x.y, pluggy-
0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F
[100%]

================================= 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
=========================
```

The test returns a failure error because `func(3)` does not return **5**.

Note: You can use the `assert` statement to verify the expected behavior of the test. The Advanced assertion introspection, by default, reports intermediate values of the assert expression so you can avoid the many names of JUnit legacy methods.

Run multiple tests

`Pytest` follows the standard test discovery rules and runs all Python files that conform to the naming convention **test_*.py** in the current directory and its subdirectories.
Note: You can add a functional test file in the same directory that raises an `AssertionError` exception if the assert condition fails.

1. Create a file named `test_sysexit.py` with the following content:

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

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

2. Using the command line, run the following command to execute the test function:

```
$ pytest -q test_sysexit.py
[100%]
1 passed in 0.12 seconds
```

Note: `py.test -q` <file/directory> - is the default unit testing with a summarized report (quiet mode)

Group test functions into classes

pytest enables you to create a class with more than one test and group multiple test functions into classes. pytest follows the Conventions for Python test discovery to identify all tests, so it runs both test_* prefixed functions in your class.
Note: Subclass is not required as you can run the module by passing its filename.

1. Create a file named test_class.py with the following content:

```
# 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')
```

2. Using the command line, execute the test function by running the following command:

```
$ pytest -q test_class.py
.F                                                    [100%]
================================= 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
```

You can see that while the first test passed, the second test failed. The intermediate values in the assertion help you understand the reason for the failure.

Request temporary directories for functional tests

`pytest` enables you to request arbitrary resources, such as unique temporary directories, through its Builtin fixtures/function arguments.

In the following exercise, you create a unique temporary directory. `pytest`, then, identifies the temp directory and calls a fixture factory to create the resource before performing the test function call.

1. Create a file named `test_tmpdir.py` with the following content:

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

2. `pytest` creates a unique-per-test-invocation temporary directory before running the test.

```
$ pytest -q test_tmpdir.py
F
[100%]
================================= 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
```

To learn more about `tmpdir` handling, see Temporary directories and files. You can run the following command for more information about `builtin pytest fixtures`:

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

Note, this command omits fixtures with leading _ unless the -v option is added.

Next steps

For more information on `pytest` resources and how to customize tests for your unique workflows, refer the following links:

- command line invocation examples
- working with pre-existing tests
- information about pytest.mark mechanism
- providing a functional baseline to your tests
- managing and writing plugins
- Good Integration Practices for `virtualenv` and test layouts