Get Started

Install `pytest`

1. Run the following command in your command line:

```
pip install -U pytest
```

2. Check that you installed the correct version:

```
$ pytest --version
pytest 9.0.1
```

Create your first test

Create a new file called `test_sample.py`, containing a function, and a test:

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


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

The test

```
$ pytest
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
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:6: AssertionError
========================== short test summary info ==========================
FAILED test_sample.py::test_answer - assert 4 == 5
============================ 1 failed in 0.12s =============================
```

The `[100%]` refers to the overall progress of running all test cases. After it finishes, pytest then shows a failure report because `func(3)` does not return `5`.

> **Note**
>
> You can use the `assert` statement to verify test expectations. pytest's Advanced assertion introspection will intelligently report intermediate values of the assert expression so you can avoid the many names of JUnit legacy methods.

Run multiple tests

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

Assert that a certain exception is raised

Use the raises helper to assert that some code raises an exception:

```
# content of test_sysexit.py
import pytest


def f():
    raise SystemExit(1)


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

Execute the test function with “quiet” reporting mode:

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

> **Note**
>
> The `-q/--quiet` flag keeps the output brief in this and following examples.

See Assertions about approximate equality for specifying more details about the expected exception.

Group multiple tests in a class

Once you develop multiple tests, you may want to group them into a class. pytest makes it easy to create a class containing more than one test:

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

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

`pytest` discovers all tests following its Conventions for Python test discovery, so it finds both `test_` Skip to content . There is no need to subclass anything, but make sure to prefix your class with `Test` otherwise the class will be skipped. We can simply run the module by passing its filename:

```
$ pytest -q test_class.py
.F                                                                   [100%]
================================= FAILURES =================================
_____________________________ TestClass.test_two _____________________________

self = <test_class.TestClass object at 0xdeadbeef0001>

    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
========================= short test summary info ==========================
FAILED test_class.py::TestClass::test_two - AssertionError: assert False
1 failed, 1 passed in 0.12s
```

The first test passed and the second failed. You can easily see the intermediate values in the assertion to help you understand the reason for the failure.

Grouping tests in classes can be beneficial for the following reasons:

- Test organization
- Sharing fixtures for tests only in that particular class
- Applying marks at the class level and having them implicitly apply to all tests

Something to be aware of when grouping tests inside classes is that each test has a unique instance of the class. Having each test share the same class instance would be very detrimental to test isolation and would promote poor test practices. This is outlined below:

```
# content of test_class_demo.py
class TestClassDemoInstance:
    value = 0

    def test_one(self):
        self.value = 1
        assert self.value == 1

    def test_two(self):
        self.value == 1
```

```
$ pytest -k TestClassDemoInstance -q
.F                                                                   [100%]
================================= FAILURES =================================
_____________________ TestClassDemoInstance.test_two ______________________

self = <test_class_demo.TestClassDemoInstance object at 0xdeadbeef0002>

    def test_two(self):
>       assert self.value == 1
E       assert 0 == 1
E        +  where 0 = <test_class_demo.TestClassDemoInstance object at 0xdeadbeef0002>.value

test_class_demo.py:9: AssertionError
========================= short test summary info ==========================
FAILED test_class_demo.py::TestClassDemoInstance::test_two - assert 0 == 1
1 failed, 1 passed in 0.12s
```

Note that attributes added at class level are *class attributes*, so they will be shared between tests.

Compare floating-point values with pytest.approx

`pytest` also provides a number of utilities to make writing tests easier. For example, you can use `pytest.approx()` to compare floating-point values that may have small rounding errors:

```
# content of test_approx.py
import pytest


def test_sum():
    assert (0.1 + 0.2) == pytest.approx(0.3)
```

This avoids the need for manual tolerance checks or using `math.isclose` and works with scalars, lists, and NumPy arrays.

Request a unique temporary directory for functional tests

`pytest` provides Builtin fixtures/function arguments to request arbitrary resources, like a unique

 ry:

```
# content of test_tmp_path.py
def test_needsfiles(tmp_path):
    print(tmp_path)
    assert 0
```

List the name `tmp_path` in the test function signature and `pytest` will lookup and call a fixture factory to create the resource before performing the test function call. Before the test runs, `pytest` creates a unique-per-test-invocation temporary directory:

```
$ pytest -q test_tmp_path.py
F                                                                    [100%]
================================= FAILURES =================================
_____________________________ test_needsfiles ______________________________

tmp_path = PosixPath('PYTEST_TMPDIR/test_needsfiles0')

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

test_tmp_path.py:3: AssertionError
--------------------------- Captured stdout call ---------------------------
PYTEST_TMPDIR/test_needsfiles0
========================= short test summary info ==========================
FAILED test_tmp_path.py::test_needsfiles - assert 0
1 failed in 0.12s
```

More info on temporary directory handling is available at Temporary directories and files.

Find out what kind of builtin pytest fixtures exist with the command:

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

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

Continue reading

Check out additional pytest resources to help you customize tests for your unique workflow:

- “How to invoke pytest” for command line invocation examples
- “How to use pytest with an existing test suite” for working with preexisting tests
- “How to mark test functions with attributes” for information on the `pytest.mark` mechanism
- “Fixtures reference” for providing a functional baseline to your tests
- “Writing plugins” for managing and writing plugins
- “Good Integration Practices” for virtualenv and test layouts

---