



pytest

pytest Documentation

Release 6.3

holger krekel, trainer and consultant, <https://merlinux.eu/>

Oct 27, 2021

CONTENTS

1	Start here	3
1.1	Get Started	3
2	How-to guides	9
2.1	How to invoke pytest	9
2.2	How to write and report assertions in tests	12
2.3	How to use fixtures	17
2.4	How to mark test functions with attributes	45
2.5	How to parametrize fixtures and test functions	47
2.6	How to use temporary directories and files in tests	51
2.7	How to monkeypatch/mock modules and environments	53
2.8	How to run doctests	59
2.9	How to re-run failed tests and maintain state between test runs	64
2.10	How to manage logging	69
2.11	How to capture stdout/stderr output	73
2.12	How to capture warnings	75
2.13	How to use skip and xfail to deal with tests that cannot succeed	81
2.14	How to install and use plugins	87
2.15	Writing plugins	89
2.16	Writing hook functions	95
2.17	How to use pytest with an existing test suite	100
2.18	How to use unittest-based tests with pytest	101
2.19	How to run tests written for nose	104
2.20	How to implement xunit-style set-up	105
2.21	How to set up bash completion	107
3	Reference guides	109
3.1	Fixtures reference	109
3.2	Plugin List	126
3.3	Configuration	146
3.4	API Reference	149
4	Explanation	247
4.1	Anatomy of a test	247
4.2	About fixtures	248
4.3	Good Integration Practices	250
4.4	Flaky tests	254
4.5	pytest import mechanisms and <code>sys.path/PYTHONPATH</code>	257
5	Further topics	261

5.1	Examples and customization tricks	261
5.2	Backwards Compatibility Policy	323
5.3	History	324
5.4	Deprecations and Removals	325
5.5	Python 2.7 and 3.4 support	337
5.6	Contribution getting started	339
5.7	Development Guide	347
5.8	Sponsor	347
5.9	pytest for enterprise	347
5.10	License	348
5.11	Contact channels	349
5.12	History	349
5.13	Historical Notes	351
5.14	Talks and Tutorials	355

Index		357
--------------	--	------------

[Download latest version as PDF](#)

START HERE

1.1 Get Started

1.1.1 Install `pytest`

`pytest` requires: Python 3.6, 3.7, 3.8, 3.9, or PyPy3.

1. Run the following command in your command line:

```
pip install -U pytest
```

2. Check that you installed the correct version:

```
$ pytest --version
pytest 6.3.0.dev685+g581b021aa.d20210922
```

1.1.2 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-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_sample.py F [100%]

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

(continues on next page)

(continued from previous page)

```

    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](#).

1.1.3 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*.

1.1.4 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
.
1 passed in 0.12s
[100%]

```

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

1.1.5 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_` prefixed functions. 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
```

(continues on next page)

(continued from previous page)

```
def test_two(self):
    assert 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.

1.1.6 Request a unique temporary directory for functional tests

pytest provides Builtin fixtures/function arguments to request arbitrary resources, like a unique temporary directory:

```
# 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.

1.1.7 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 pre-existing 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

HOW-TO GUIDES

2.1 How to invoke pytest

See also:

Complete pytest command-line flag reference

In general, pytest is invoked with the command `pytest` (see below for *other ways to invoke pytest*). This will execute all tests in all files whose names follow the form `test_*.py` or `*_test.py` in the current directory and its subdirectories. More generally, pytest follows *standard test discovery rules*.

2.1.1 Specifying which tests to run

Pytest supports several ways to run and select tests from the command-line.

Run tests in a module

```
pytest test_mod.py
```

Run tests in a directory

```
pytest testing/
```

Run tests by keyword expressions

```
pytest -k "MyClass and not method"
```

This will run tests which contain names that match the given *string expression* (case-insensitive), which can include Python operators that use filenames, class names and function names as variables. The example above will run `TestMyClass.test_something` but not `TestMyClass.test_method_simple`.

Run tests by node ids

Each collected test is assigned a unique `nodeid` which consist of the module filename followed by specifiers like class names, function names and parameters from parametrization, separated by `::` characters.

To run a specific test within a module:

```
pytest test_mod.py::test_func
```

Another example specifying a test method in the command line:

```
pytest test_mod.py::TestClass::test_method
```

Run tests by marker expressions

```
pytest -m slow
```

Will run all tests which are decorated with the `@pytest.mark.slow` decorator.

For more information see [marks](#).

Run tests from packages

```
pytest --pyargs pkg.testing
```

This will import `pkg.testing` and use its filesystem location to find and run tests from.

2.1.2 Getting help on version, option names, environment variables

```
pytest --version # shows where pytest was imported from
pytest --fixtures # show available builtin function arguments
pytest -h | --help # show help on command line and config file options
```

2.1.3 Profiling test execution duration

Changed in version 6.0.

To get a list of the slowest 10 test durations over 1.0s long:

```
pytest --durations=10 --durations-min=1.0
```

By default, pytest will not show test durations that are too small (<0.005s) unless `-vv` is passed on the command-line.

2.1.4 Managing loading of plugins

Early loading plugins

You can early-load plugins (internal and external) explicitly in the command-line with the `-p` option:

```
pytest -p mypluginmodule
```

The option receives a `name` parameter, which can be:

- A full module dotted name, for example `myproject.plugins`. This dotted name must be importable.
- The entry-point name of a plugin. This is the name passed to `setuptools` when the plugin is registered. For example to early-load the `pytest-cov` plugin you can use:

```
pytest -p pytest_cov
```

Disabling plugins

To disable loading specific plugins at invocation time, use the `-p` option together with the prefix `no:`.

Example: to disable loading the plugin `doctest`, which is responsible for executing `doctest` tests from text files, invoke `pytest` like this:

```
pytest -p no:doctest
```

2.1.5 Other ways of calling pytest

Calling pytest through `python -m pytest`

You can invoke testing through the Python interpreter from the command line:

```
python -m pytest [...]
```

This is almost equivalent to invoking the command line script `pytest [...]` directly, except that calling via `python` will also add the current directory to `sys.path`.

Calling pytest from Python code

You can invoke `pytest` from Python code directly:

```
retcode = pytest.main()
```

this acts as if you would call “`pytest`” from the command line. It will not raise `SystemExit` but return the exit code instead. You can pass in options and arguments:

```
retcode = pytest.main(["-x", "mytestdir"])
```

You can specify additional plugins to `pytest.main()`:

```
# content of myinvoke.py
import pytest
import sys

class MyPlugin:
    def pytest_sessionfinish(self):
        print("*** test run reporting finishing")

if __name__ == "__main__":
    sys.exit(pytest.main(["-qq"], plugins=[MyPlugin()]))
```

Running it will show that `MyPlugin` was added and its hook was invoked:

```
$ python myinvoke.py
*** test run reporting finishing
```

Note: Calling `pytest.main()` will result in importing your tests and any modules that they import. Due to the caching mechanism of python’s import system, making subsequent calls to `pytest.main()` from the same process

will not reflect changes to those files between the calls. For this reason, making multiple calls to `pytest.main()` from the same process (in order to re-run tests, for example) is not recommended.

2.2 How to write and report assertions in tests

2.2.1 Asserting with the `assert` statement

pytest allows you to use the standard Python `assert` for verifying expectations and values in Python tests. For example, you can write the following:

```
# content of test_assert1.py
def f():
    return 3

def test_function():
    assert f() == 4
```

to assert that your function returns a certain value. If this assertion fails you will see the return value of the function call:

```
$ pytest test_assert1.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_assert1.py F [100%]

===== FAILURES =====
_____ test_function _____

    def test_function():
>     assert f() == 4
E       assert 3 == 4
E         + where 3 = f()

test_assert1.py:6: AssertionError
===== short test summary info =====
FAILED test_assert1.py::test_function - assert 3 == 4
===== 1 failed in 0.12s =====
```

pytest has support for showing the values of the most common subexpressions including calls, attributes, comparisons, and binary and unary operators. (See *Demo of Python failure reports with pytest*). This allows you to use the idiomatic python constructs without boilerplate code while not losing introspection information.

However, if you specify a message with the assertion like this:

```
assert a % 2 == 0, "value was odd, should be even"
```

then no assertion introspection takes places at all and the message will be simply shown in the traceback.

See *Assertion introspection details* for more information on assertion introspection.

2.2.2 Assertions about expected exceptions

In order to write assertions about raised exceptions, you can use `pytest.raises()` as a context manager like this:

```
import pytest

def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

and if you need to have access to the actual exception info you may use:

```
def test_recursion_depth():
    with pytest.raises(RuntimeError) as excinfo:

        def f():
            f()

        f()
    assert "maximum recursion" in str(excinfo.value)
```

`excinfo` is an `ExceptionInfo` instance, which is a wrapper around the actual exception raised. The main attributes of interest are `.type`, `.value` and `.traceback`.

You can pass a `match` keyword parameter to the context-manager to test that a regular expression matches on the string representation of an exception (similar to the `TestCase.assertRaisesRegex` method from `unittest`):

```
import pytest

def myfunc():
    raise ValueError("Exception 123 raised")

def test_match():
    with pytest.raises(ValueError, match=r".* 123 .*"):
        myfunc()
```

The `regex` parameter of the `match` method is matched with the `re.search` function, so in the above example `match='123'` would have worked as well.

There's an alternate form of the `pytest.raises()` function where you pass a function that will be executed with the given `*args` and `**kwargs` and assert that the given exception is raised:

```
pytest.raises(ExpectedException, func, *args, **kwargs)
```

The reporter will provide you with helpful output in case of failures such as *no exception* or *wrong exception*.

Note that it is also possible to specify a “raises” argument to `pytest.mark.xfail`, which checks that the test is failing in a more specific way than just having any exception raised:

```
@pytest.mark.xfail(raises=IndexError)
def test_f():
    f()
```

Using `pytest.raises()` is likely to be better for cases where you are testing exceptions your own code is deliberately raising, whereas using `@pytest.mark.xfail` with a check function is probably better for something like documenting unfixed bugs (where the test describes what “should” happen) or bugs in dependencies.

2.2.3 Assertions about expected warnings

You can check that code raises a particular warning using `pytest.warns`.

2.2.4 Making use of context-sensitive comparisons

pytest has rich support for providing context-sensitive information when it encounters comparisons. For example:

```
# content of test_assert2.py
def test_set_comparison():
    set1 = set("1308")
    set2 = set("8035")
    assert set1 == set2
```

if you run this module:

```
$ pytest test_assert2.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_assert2.py F [100%]

===== FAILURES =====
_____ test_set_comparison _____

    def test_set_comparison():
        set1 = set("1308")
        set2 = set("8035")
>       assert set1 == set2
E       AssertionError: assert {'0', '1', '3', '8'} == {'0', '3', '5', '8'}
E         Extra items in the left set:
E         '1'
E         Extra items in the right set:
E         '5'
E         Use -v to get the full diff

test_assert2.py:4: AssertionError
===== short test summary info =====
FAILED test_assert2.py::test_set_comparison - AssertionError: assert {'0'...
===== 1 failed in 0.12s =====
```

Special comparisons are done for a number of cases:

- comparing long strings: a context diff is shown
- comparing long sequences: first failing indices
- comparing dicts: different entries

See the [reporting demo](#) for many more examples.

2.2.5 Defining your own explanation for failed assertions

It is possible to add your own detailed explanations by implementing the `pytest_assertrepr_compare` hook.

pytest_assertrepr_compare (*config, op, left, right*)

Return explanation for comparisons in failing assert expressions.

Return None for no custom explanation, otherwise return a list of strings. The strings will be joined by newlines but any newlines *in* a string will be escaped. Note that all but the first line will be indented slightly, the intention is for the first line to be a summary.

Parameters

- **config** (`pytest.Config`) – The pytest config object.
- **op** (*str*) –
- **left** (*object*) –
- **right** (*object*) –

Return type Optional[List[str]]

As an example consider adding the following hook in a `conftest.py` file which provides an alternative explanation for `Foo` objects:

```
# content of conftest.py
from test_foocompare import Foo

def pytest_assertrepr_compare(op, left, right):
    if isinstance(left, Foo) and isinstance(right, Foo) and op == "==":
        return [
            "Comparing Foo instances:",
            "  vals: {} != {}".format(left.val, right.val),
        ]
```

now, given this test module:

```
# content of test_foocompare.py
class Foo:
    def __init__(self, val):
        self.val = val

    def __eq__(self, other):
        return self.val == other.val

def test_compare():
    f1 = Foo(1)
    f2 = Foo(2)
    assert f1 == f2
```

you can run the test module and get the custom output defined in the conftest file:

```
$ pytest -q test_foocompare.py
F [100%]
===== FAILURES =====
_____ test_compare _____

def test_compare():
```

(continues on next page)

(continued from previous page)

```
f1 = Foo(1)
f2 = Foo(2)
> assert f1 == f2
E     assert Comparing Foo instances:
E         vals: 1 != 2

test_foocompare.py:12: AssertionError
===== short test summary info =====
FAILED test_foocompare.py::test_compare - assert Comparing Foo instances:
1 failed in 0.12s
```

2.2.6 Assertion introspection details

Reporting details about a failing assertion is achieved by rewriting assert statements before they are run. Rewritten assert statements put introspection information into the assertion failure message. `pytest` only rewrites test modules directly discovered by its test collection process, so **asserts in supporting modules which are not themselves test modules will not be rewritten**.

You can manually enable assertion rewriting for an imported module by calling `register_assert_rewrite` before you import it (a good place to do that is in your root `conftest.py`).

For further information, Benjamin Peterson wrote up [Behind the scenes of pytest's new assertion rewriting](#).

Assertion rewriting caches files on disk

`pytest` will write back the rewritten modules to disk for caching. You can disable this behavior (for example to avoid leaving stale `.pyc` files around in projects that move files around a lot) by adding this to the top of your `conftest.py` file:

```
import sys

sys.dont_write_bytecode = True
```

Note that you still get the benefits of assertion introspection, the only change is that the `.pyc` files won't be cached on disk.

Additionally, rewriting will silently skip caching if it cannot write new `.pyc` files, i.e. in a read-only filesystem or a zipfile.

Disabling assert rewriting

`pytest` rewrites test modules on import by using an import hook to write new `pyc` files. Most of the time this works transparently. However, if you are working with the import machinery yourself, the import hook may interfere.

If this is the case you have two options:

- Disable rewriting for a specific module by adding the string `PYTEST_DONT_REWRITE` to its docstring.
- Disable rewriting for all modules by using `--assert=plain`.

2.3 How to use fixtures

See also:

About fixtures

See also:

Fixtures reference

2.3.1 “Requesting” fixtures

At a basic level, test functions request fixtures they require by declaring them as arguments.

When pytest goes to run a test, it looks at the parameters in that test function’s signature, and then searches for fixtures that have the same names as those parameters. Once pytest finds them, it runs those fixtures, captures what they returned (if anything), and passes those objects into the test function as arguments.

Quick example

```
import pytest

class Fruit:
    def __init__(self, name):
        self.name = name
        self.cubed = False

    def cube(self):
        self.cubed = True

class FruitSalad:
    def __init__(self, *fruit_bowl):
        self.fruit = fruit_bowl
        self._cube_fruit()

    def _cube_fruit(self):
        for fruit in self.fruit:
            fruit.cube()

# Arrange
@pytest.fixture
def fruit_bowl():
    return [Fruit("apple"), Fruit("banana")]

def test_fruit_salad(fruit_bowl):
    # Act
    fruit_salad = FruitSalad(*fruit_bowl)

    # Assert
    assert all(fruit.cubed for fruit in fruit_salad.fruit)
```

In this example, `test_fruit_salad` “requests” `fruit_bowl` (i.e. `def test_fruit_salad(fruit_bowl):`), and when pytest sees this, it will execute the `fruit_bowl` fixture function and pass the object it returns into `test_fruit_salad` as the `fruit_bowl` argument.

Here’s roughly what’s happening if we were to do it by hand:

```
def fruit_bowl():
    return [Fruit("apple"), Fruit("banana")]

def test_fruit_salad(fruit_bowl):
    # Act
    fruit_salad = FruitSalad(*fruit_bowl)

    # Assert
    assert all(fruit.cubed for fruit in fruit_salad.fruit)

# Arrange
bowl = fruit_bowl()
test_fruit_salad(fruit_bowl=bowl)
```

Fixtures can request other fixtures

One of pytest’s greatest strengths is its extremely flexible fixture system. It allows us to boil down complex requirements for tests into more simple and organized functions, where we only need to have each one describe the things they are dependent on. We’ll get more into this further down, but for now, here’s a quick example to demonstrate how fixtures can use other fixtures:

```
# contents of test_append.py
import pytest

# Arrange
@pytest.fixture
def first_entry():
    return "a"

# Arrange
@pytest.fixture
def order(first_entry):
    return [first_entry]

def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]
```

Notice that this is the same example from above, but very little changed. The fixtures in pytest **request** fixtures just like tests. All the same **requesting** rules apply to fixtures that do for tests. Here’s how this example would work if we did it by hand:

```

def first_entry():
    return "a"

def order(first_entry):
    return [first_entry]

def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]

entry = first_entry()
the_list = order(first_entry=entry)
test_string(order=the_list)

```

Fixtures are reusable

One of the things that makes pytest's fixture system so powerful, is that it gives us the ability to define a generic setup step that can be reused over and over, just like a normal function would be used. Two different tests can request the same fixture and have pytest give each test their own result from that fixture.

This is extremely useful for making sure tests aren't affected by each other. We can use this system to make sure each test gets its own fresh batch of data and is starting from a clean state so it can provide consistent, repeatable results.

Here's an example of how this can come in handy:

```

# contents of test_append.py
import pytest

# Arrange
@pytest.fixture
def first_entry():
    return "a"

# Arrange
@pytest.fixture
def order(first_entry):
    return [first_entry]

def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]

def test_int(order):

```

(continues on next page)

(continued from previous page)

```
# Act
order.append(2)

# Assert
assert order == ["a", 2]
```

Each test here is being given its own copy of that `list` object, which means the `order` fixture is getting executed twice (the same is true for the `first_entry` fixture). If we were to do this by hand as well, it would look something like this:

```
def first_entry():
    return "a"

def order(first_entry):
    return [first_entry]

def test_string(order):
    # Act
    order.append("b")

    # Assert
    assert order == ["a", "b"]

def test_int(order):
    # Act
    order.append(2)

    # Assert
    assert order == ["a", 2]

entry = first_entry()
the_list = order(first_entry=entry)
test_string(order=the_list)

entry = first_entry()
the_list = order(first_entry=entry)
test_int(order=the_list)
```

A test/fixture can request more than one fixture at a time

Tests and fixtures aren't limited to **requesting** a single fixture at a time. They can request as many as they like. Here's another quick example to demonstrate:

```
# contents of test_append.py
import pytest

# Arrange
@pytest.fixture
def first_entry():
    return "a"
```

(continues on next page)

(continued from previous page)

```

# Arrange
@pytest.fixture
def second_entry():
    return 2

# Arrange
@pytest.fixture
def order(first_entry, second_entry):
    return [first_entry, second_entry]

# Arrange
@pytest.fixture
def expected_list():
    return ["a", 2, 3.0]

def test_string(order, expected_list):
    # Act
    order.append(3.0)

    # Assert
    assert order == expected_list

```

Fixtures can be requested more than once per test (return values are cached)

Fixtures can also be **requested** more than once during the same test, and pytest won't execute them again for that test. This means we can **request** fixtures in multiple fixtures that are dependent on them (and even again in the test itself) without those fixtures being executed more than once.

```

# contents of test_append.py
import pytest

# Arrange
@pytest.fixture
def first_entry():
    return "a"

# Arrange
@pytest.fixture
def order():
    return []

# Act
@pytest.fixture
def append_first(order, first_entry):
    return order.append(first_entry)

```

(continues on next page)

(continued from previous page)

```
def test_string_only(append_first, order, first_entry):
    # Assert
    assert order == [first_entry]
```

If a **requested** fixture was executed once for every time it was **requested** during a test, then this test would fail because both `append_first` and `test_string_only` would see `order` as an empty list (i.e. `[]`), but since the return value of `order` was cached (along with any side effects executing it may have had) after the first time it was called, both the test and `append_first` were referencing the same object, and the test saw the effect `append_first` had on that object.

2.3.2 Autouse fixtures (fixtures you don't have to request)

Sometimes you may want to have a fixture (or even several) that you know all your tests will depend on. “Autouse” fixtures are a convenient way to make all tests automatically **request** them. This can cut out a lot of redundant **requests**, and can even provide more advanced fixture usage (more on that further down).

We can make a fixture an autouse fixture by passing in `autouse=True` to the fixture's decorator. Here's a simple example for how they can be used:

```
# contents of test_append.py
import pytest

@pytest.fixture
def first_entry():
    return "a"

@pytest.fixture
def order(first_entry):
    return []

@pytest.fixture(autouse=True)
def append_first(order, first_entry):
    return order.append(first_entry)

def test_string_only(order, first_entry):
    assert order == [first_entry]

def test_string_and_int(order, first_entry):
    order.append(2)
    assert order == [first_entry, 2]
```

In this example, the `append_first` fixture is an autouse fixture. Because it happens automatically, both tests are affected by it, even though neither test **requested** it. That doesn't mean they *can't* be **requested** though; just that it isn't *necessary*.

2.3.3 Scope: sharing fixtures across classes, modules, packages or session

Fixtures requiring network access depend on connectivity and are usually time-expensive to create. Extending the previous example, we can add a `scope="module"` parameter to the `@pytest.fixture` invocation to cause a `smtp_connection` fixture function, responsible to create a connection to a preexisting SMTP server, to only be invoked once per test *module* (the default is to invoke once per test *function*). Multiple test functions in a test module will thus each receive the same `smtp_connection` fixture instance, thus saving time. Possible values for `scope` are: `function`, `class`, `module`, `package` or `session`.

The next example puts the fixture function into a separate `conftest.py` file so that tests from multiple test modules in the directory can access the fixture function:

```
# content of conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module")
def smtp_connection():
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
```

```
# content of test_module.py

def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
    assert b"smtp.gmail.com" in msg
    assert 0 # for demo purposes

def test_noop(smtp_connection):
    response, msg = smtp_connection.noop()
    assert response == 250
    assert 0 # for demo purposes
```

Here, the `test_ehlo` needs the `smtp_connection` fixture value. `pytest` will discover and call the `@pytest.fixture` marked `smtp_connection` fixture function. Running the test looks like this:

```
$ pytest test_module.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items

test_module.py FF [100%]

===== FAILURES =====
_____ test_ehlo _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0001>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
        assert b"smtp.gmail.com" in msg
```

(continues on next page)

(continued from previous page)

```

>     assert 0 # for demo purposes
E     assert 0

test_module.py:7: AssertionError
_____ test_noop _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0001>

    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
        assert response == 250
>     assert 0 # for demo purposes
E     assert 0

test_module.py:13: AssertionError
===== short test summary info =====
FAILED test_module.py::test_ehlo - assert 0
FAILED test_module.py::test_noop - assert 0
===== 2 failed in 0.12s =====

```

You see the two `assert 0` failing and more importantly you can also see that the **exactly same** `smtp_connection` object was passed into the two test functions because pytest shows the incoming argument values in the traceback. As a result, the two test functions using `smtp_connection` run as quick as a single one because they reuse the same instance.

If you decide that you rather want to have a session-scoped `smtp_connection` instance, you can simply declare it:

```

@pytest.fixture(scope="session")
def smtp_connection():
    # the returned fixture value will be shared for
    # all tests requesting it
    ...

```

Fixture scopes

Fixtures are created when first requested by a test, and are destroyed based on their `scope`:

- `function`: the default scope, the fixture is destroyed at the end of the test.
- `class`: the fixture is destroyed during teardown of the last test in the class.
- `module`: the fixture is destroyed during teardown of the last test in the module.
- `package`: the fixture is destroyed during teardown of the last test in the package.
- `session`: the fixture is destroyed at the end of the test session.

Note: Pytest only caches one instance of a fixture at a time, which means that when using a parametrized fixture, pytest may invoke a fixture more than once in the given scope.

Dynamic scope

New in version 5.2.

In some cases, you might want to change the scope of the fixture without changing the code. To do that, pass a callable to `scope`. The callable must return a string with a valid scope and will be executed only once - during the fixture definition. It will be called with two keyword arguments - `fixture_name` as a string and `config` with a configuration object.

This can be especially useful when dealing with fixtures that need time for setup, like spawning a docker container. You can use the command-line argument to control the scope of the spawned containers for different environments. See the example below.

```
def determine_scope(fixture_name, config):
    if config.getoption("--keep-containers", None):
        return "session"
    return "function"

@pytest.fixture(scope=determine_scope)
def docker_container():
    yield spawn_container()
```

2.3.4 Teardown/Cleanup (AKA Fixture finalization)

When we run our tests, we'll want to make sure they clean up after themselves so they don't mess with any other tests (and also so that we don't leave behind a mountain of test data to bloat the system). Fixtures in pytest offer a very useful teardown system, which allows us to define the specific steps necessary for each fixture to clean up after itself.

This system can be leveraged in two ways.

1. yield fixtures (recommended)

"Yield" fixtures `yield` instead of `return`. With these fixtures, we can run some code and pass an object back to the requesting fixture/test, just like with the other fixtures. The only differences are:

1. `return` is swapped out for `yield`.
2. Any teardown code for that fixture is placed *after* the `yield`.

Once pytest figures out a linear order for the fixtures, it will run each one up until it returns or yields, and then move on to the next fixture in the list to do the same thing.

Once the test is finished, pytest will go back down the list of fixtures, but in the *reverse order*, taking each one that yielded, and running the code inside it that was *after* the `yield` statement.

As a simple example, consider this basic email module:

```
# content of emaillib.py
class MailAdminClient:
    def create_user(self):
        return MailUser()

    def delete_user(self, user):
        # do some cleanup
        pass
```

(continues on next page)

(continued from previous page)

```
class MailUser:
    def __init__(self):
        self.inbox = []

    def send_email(self, email, other):
        other.inbox.append(email)

    def clear_mailbox(self):
        self.inbox.clear()

class Email:
    def __init__(self, subject, body):
        self.subject = subject
        self.body = body
```

Let's say we want to test sending email from one user to another. We'll have to first make each user, then send the email from one user to the other, and finally assert that the other user received that message in their inbox. If we want to clean up after the test runs, we'll likely have to make sure the other user's mailbox is emptied before deleting that user, otherwise the system may complain.

Here's what that might look like:

```
# content of test_emaillib.py
import pytest

from emaillib import Email, MailAdminClient

@pytest.fixture
def mail_admin():
    return MailAdminClient()

@pytest.fixture
def sending_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    mail_admin.delete_user(user)

@pytest.fixture
def receiving_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    mail_admin.delete_user(user)

def test_email_received(sending_user, receiving_user):
    email = Email(subject="Hey!", body="How's it going?")
    sending_user.send_email(email, receiving_user)
    assert email in receiving_user.inbox
```

Because `receiving_user` is the last fixture to run during setup, it's the first to run during teardown.

There is a risk that even having the order right on the teardown side of things doesn't guarantee a safe cleanup. That's covered in a bit more detail in *Safe teardowns*.

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

Handling errors for yield fixture

If a yield fixture raises an exception before yielding, pytest won't try to run the teardown code after that yield fixture's `yield` statement. But, for every fixture that has already run successfully for that test, pytest will still attempt to tear them down as it normally would.

2. Adding finalizers directly

While yield fixtures are considered to be the cleaner and more straightforward option, there is another choice, and that is to add “finalizer” functions directly to the test's *request-context* object. It brings a similar result as yield fixtures, but requires a bit more verbosity.

In order to use this approach, we have to request the *request-context* object (just like we would request another fixture) in the fixture we need to add teardown code for, and then pass a callable, containing that teardown code, to its `addfinalizer` method.

We have to be careful though, because pytest will run that finalizer once it's been added, even if that fixture raises an exception after adding the finalizer. So to make sure we don't run the finalizer code when we wouldn't need to, we would only add the finalizer once the fixture would have done something that we'd need to teardown.

Here's how the previous example would look using the `addfinalizer` method:

```
# content of test_emaillib.py
import pytest

from emaillib import Email, MailAdminClient

@pytest.fixture
def mail_admin():
    return MailAdminClient()

@pytest.fixture
def sending_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    mail_admin.delete_user(user)

@pytest.fixture
def receiving_user(mail_admin, request):
    user = mail_admin.create_user()

    def delete_user():
        mail_admin.delete_user(user)

    request.addfinalizer(delete_user)
    return user
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture
def email(sending_user, receiving_user, request):
    _email = Email(subject="Hey!", body="How's it going?")
    sending_user.send_email(_email, receiving_user)

    def empty_mailbox():
        receiving_user.clear_mailbox()

    request.addfinalizer(empty_mailbox)
    return _email

def test_email_received(receiving_user, email):
    assert email in receiving_user.inbox
```

It's a bit longer than yield fixtures and a bit more complex, but it does offer some nuances for when you're in a pinch.

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

2.3.5 Safe teardowns

The fixture system of pytest is *very* powerful, but it's still being run by a computer, so it isn't able to figure out how to safely teardown everything we throw at it. If we aren't careful, an error in the wrong spot might leave stuff from our tests behind, and that can cause further issues pretty quickly.

For example, consider the following tests (based off of the mail example from above):

```
# content of test_emaillib.py
import pytest

from emaillib import Email, MailAdminClient

@pytest.fixture
def setup():
    mail_admin = MailAdminClient()
    sending_user = mail_admin.create_user()
    receiving_user = mail_admin.create_user()
    email = Email(subject="Hey!", body="How's it going?")
    sending_user.send_email(email, receiving_user)
    yield receiving_user, email
    receiving_user.clear_mailbox()
    mail_admin.delete_user(sending_user)
    mail_admin.delete_user(receiving_user)

def test_email_received(setup):
    receiving_user, email = setup
    assert email in receiving_user.inbox
```

This version is a lot more compact, but it's also harder to read, doesn't have a very descriptive fixture name, and none of the fixtures can be reused easily.

There's also a more serious issue, which is that if any of those steps in the setup raise an exception, none of the teardown code will run.

One option might be to go with the `addfinalizer` method instead of yield fixtures, but that might get pretty complex and difficult to maintain (and it wouldn't be compact anymore).

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

Safe fixture structure

The safest and simplest fixture structure requires limiting fixtures to only making one state-changing action each, and then bundling them together with their teardown code, as *the email examples above* showed.

The chance that a state-changing operation can fail but still modify state is negligible, as most of these operations tend to be `transaction`-based (at least at the level of testing where state could be left behind). So if we make sure that any successful state-changing action gets torn down by moving it to a separate fixture function and separating it from other, potentially failing state-changing actions, then our tests will stand the best chance at leaving the test environment the way they found it.

For an example, let's say we have a website with a login page, and we have access to an admin API where we can generate users. For our test, we want to:

1. Create a user through that admin API
2. Launch a browser using Selenium
3. Go to the login page of our site
4. Log in as the user we created
5. Assert that their name is in the header of the landing page

We wouldn't want to leave that user in the system, nor would we want to leave that browser session running, so we'll want to make sure the fixtures that create those things clean up after themselves.

Here's what that might look like:

Note: For this example, certain fixtures (i.e. `base_url` and `admin_credentials`) are implied to exist elsewhere. So for now, let's assume they exist, and we're just not looking at them.

```
from uuid import uuid4
from urllib.parse import urljoin

from selenium.webdriver import Chrome
import pytest

from src.utils.pages import LoginPage, LandingPage
from src.utils import AdminApiClient
from src.utils.data_types import User

@pytest.fixture
def admin_client(base_url, admin_credentials):
    return AdminApiClient(base_url, **admin_credentials)
```

(continues on next page)

```
@pytest.fixture
def user(admin_client):
    _user = User(name="Susan", username=f"testuser-{{uuid4()}}", password="P4$$word")
    admin_client.create_user(_user)
    yield _user
    admin_client.delete_user(_user)

@pytest.fixture
def driver():
    _driver = Chrome()
    yield _driver
    _driver.quit()

@pytest.fixture
def login(driver, base_url, user):
    driver.get(urljoin(base_url, "/login"))
    page = LoginPage(driver)
    page.login(user)

@pytest.fixture
def landing_page(driver, login):
    return LandingPage(driver)

def test_name_on_landing_page_after_login(landing_page, user):
    assert landing_page.header == f"Welcome, {user.name}!"
```

The way the dependencies are laid out means it's unclear if the `user` fixture would execute before the `driver` fixture. But that's ok, because those are atomic operations, and so it doesn't matter which one runs first because the sequence of events for the test is still *linearizable*. But what *does* matter is that, no matter which one runs first, if the one raises an exception while the other would not have, neither will have left anything behind. If `driver` executes before `user`, and `user` raises an exception, the driver will still quit, and the user was never made. And if `driver` was the one to raise the exception, then the driver would never have been started and the user would never have been made.

2.3.6 Running multiple `assert` statements safely

Sometimes you may want to run multiple asserts after doing all that setup, which makes sense as, in more complex systems, a single action can kick off multiple behaviors. `pytest` has a convenient way of handling this and it combines a bunch of what we've gone over so far.

All that's needed is stepping up to a larger scope, then having the **act** step defined as an autouse fixture, and finally, making sure all the fixtures are targeting that higher level scope.

Let's pull *an example from above*, and tweak it a bit. Let's say that in addition to checking for a welcome message in the header, we also want to check for a sign out button, and a link to the user's profile.

Let's take a look at how we can structure that so we can run multiple asserts without having to repeat all those steps again.

Note: For this example, certain fixtures (i.e. `base_url` and `admin_credentials`) are implied to exist elsewhere. So for now, let's assume they exist, and we're just not looking at them.

```

# contents of tests/end_to_end/test_login.py
from uuid import uuid4
from urllib.parse import urljoin

from selenium.webdriver import Chrome
import pytest

from src.utils.pages import LoginPage, LandingPage
from src.utils import AdminApiClient
from src.utils.data_types import User

@pytest.fixture(scope="class")
def admin_client(base_url, admin_credentials):
    return AdminApiClient(base_url, **admin_credentials)

@pytest.fixture(scope="class")
def user(admin_client):
    _user = User(name="Susan", username=f"testuser-{{uuid4()}}", password="P4$$word")
    admin_client.create_user(_user)
    yield _user
    admin_client.delete_user(_user)

@pytest.fixture(scope="class")
def driver():
    _driver = Chrome()
    yield _driver
    _driver.quit()

@pytest.fixture(scope="class")
def landing_page(driver, login):
    return LandingPage(driver)

class TestLandingPageSuccess:
    @pytest.fixture(scope="class", autouse=True)
    def login(self, driver, base_url, user):
        driver.get(urljoin(base_url, "/login"))
        page = LoginPage(driver)
        page.login(user)

    def test_name_in_header(self, landing_page, user):
        assert landing_page.header == f"Welcome, {user.name}!"

    def test_sign_out_button(self, landing_page):
        assert landing_page.sign_out_button.is_displayed()

    def test_profile_link(self, landing_page, user):
        profile_href = urljoin(base_url, f"/profile?id={user.profile_id}")
        assert landing_page.profile_link.get_attribute("href") == profile_href

```

Notice that the methods are only referencing `self` in the signature as a formality. No state is tied to the actual test class as it might be in the `unittest.TestCase` framework. Everything is managed by the pytest fixture system.

Each method only has to request the fixtures that it actually needs without worrying about order. This is because the `act`

fixture is an autouse fixture, and it made sure all the other fixtures executed before it. There's no more changes of state that need to take place, so the tests are free to make as many non-state-changing queries as they want without risking stepping on the toes of the other tests.

The `login` fixture is defined inside the class as well, because not every one of the other tests in the module will be expecting a successful login, and the `act` may need to be handled a little differently for another test class. For example, if we wanted to write another test scenario around submitting bad credentials, we could handle it by adding something like this to the test file:

```
class TestLandingPageBadCredentials:
    @pytest.fixture(scope="class")
    def faux_user(self, user):
        _user = deepcopy(user)
        _user.password = "badpass"
        return _user

    def test_raises_bad_credentials_exception(self, login_page, faux_user):
        with pytest.raises(BadCredentialsException):
            login_page.login(faux_user)
```

2.3.7 Fixtures can introspect the requesting test context

Fixture functions can accept the `request` object to introspect the “requesting” test function, class or module context. Further extending the previous `smtp_connection` fixture example, let's read an optional server URL from the test module which uses our fixture:

```
# content of conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module")
def smtp_connection(request):
    server = getattr(request.module, "smtpserver", "smtp.gmail.com")
    smtp_connection = smtplib.SMTP(server, 587, timeout=5)
    yield smtp_connection
    print("finalizing {} ({}).format(smtp_connection, server))
    smtp_connection.close()
```

We use the `request.module` attribute to optionally obtain an `smtpserver` attribute from the test module. If we just execute again, nothing much has changed:

```
$ pytest -s -q --tb=no test_module.py
FFfinalizing <smtplib.SMTP object at 0xdeadbeef0002> (smtp.gmail.com)

===== short test summary info =====
FAILED test_module.py::test_ehlo - assert 0
FAILED test_module.py::test_noop - assert 0
2 failed in 0.12s
```

Let's quickly create another test module that actually sets the server URL in its module namespace:

```
# content of test_anothersmtp.py

smtpserver = "mail.python.org" # will be read by smtp fixture
```

(continues on next page)

(continued from previous page)

```
def test_showhelo(smtp_connection):
    assert 0, smtp_connection.helo()
```

Running it:

```
$ pytest -qq --tb=short test_anothersmtp.py
F [100%]
===== FAILURES =====
_____ test_showhelo _____
test_anothersmtp.py:6: in test_showhelo
    assert 0, smtp_connection.helo()
E   AssertionError: (250, b'mail.python.org')
E   assert 0
----- Captured stdout teardown -----
finalizing <smtpplib.SMTP object at 0xdeadbeef0003> (mail.python.org)
===== short test summary info =====
FAILED test_anothersmtp.py::test_showhelo - AssertionError: (250, b'mail....
```

voila! The `smtp_connection` fixture function picked up our mail server name from the module namespace.

2.3.8 Using markers to pass data to fixtures

Using the `request` object, a fixture can also access markers which are applied to a test function. This can be useful to pass data into a fixture from a test:

```
import pytest

@pytest.fixture
def fixt(request):
    marker = request.node.get_closest_marker("fixt_data")
    if marker is None:
        # Handle missing marker in some way...
        data = None
    else:
        data = marker.args[0]

    # Do something with the data
    return data

@pytest.mark.fixt_data(42)
def test_fixt(fixt):
    assert fixt == 42
```

2.3.9 Factories as fixtures

The “factory as fixture” pattern can help in situations where the result of a fixture is needed multiple times in a single test. Instead of returning data directly, the fixture instead returns a function which generates the data. This function can then be called multiple times in the test.

Factories can have parameters as needed:

```
@pytest.fixture
def make_customer_record():
    def _make_customer_record(name):
        return {"name": name, "orders": []}

    return _make_customer_record

def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

If the data created by the factory requires managing, the fixture can take care of that:

```
@pytest.fixture
def make_customer_record():

    created_records = []

    def _make_customer_record(name):
        record = models.Customer(name=name, orders=[])
        created_records.append(record)
        return record

    yield _make_customer_record

    for record in created_records:
        record.destroy()

def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

2.3.10 Parametrizing fixtures

Fixture functions can be parametrized in which case they will be called multiple times, each time executing the set of dependent tests, i.e. the tests that depend on this fixture. Test functions usually do not need to be aware of their re-running. Fixture parametrization helps to write exhaustive functional tests for components which themselves can be configured in multiple ways.

Extending the previous example, we can flag the fixture to create two `smtp_connection` fixture instances which will cause all tests using the fixture to run twice. The fixture function gets access to each parameter through the special `request` object:

```
# content of conftest.py
import pytest
import smtplib

@pytest.fixture(scope="module", params=["smtp.gmail.com", "mail.python.org"])
def smtp_connection(request):
    smtp_connection = smtplib.SMTP(request.param, 587, timeout=5)
    yield smtp_connection
    print("finalizing {}".format(smtp_connection))
    smtp_connection.close()
```

The main change is the declaration of params with `@pytest.fixture`, a list of values for each of which the fixture function will execute and can access a value via `request.param`. No test function code needs to change. So let's just do another run:

```
$ pytest -q test_module.py
FFFFF [100%]
===== FAILURES =====
_____ test_ehlo[smtp.gmail.com] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0004>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
        assert b"smtp.gmail.com" in msg
>       assert 0 # for demo purposes
E       assert 0

test_module.py:7: AssertionError
_____ test_noop[smtp.gmail.com] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0004>

    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
        assert response == 250
>       assert 0 # for demo purposes
E       assert 0

test_module.py:13: AssertionError
_____ test_ehlo[mail.python.org] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0005>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
>       assert b"smtp.gmail.com" in msg
E       AssertionError: assert b'smtp.gmail.com' in b'mail.python.org\nPIPELINING\
↪nSIZE 5120000\nETRN\nSTARTTLS\nAUTH DIGEST-MD5 NTLM CRAM-MD5\nENHANCEDSTATUSCODES\
↪n8BITIME\nDSN\nSMTPUTF8\nCHUNKING'

test_module.py:6: AssertionError
----- Captured stdout setup -----
finalizing <smtplib.SMTP object at 0xdeadbeef0004>
```

(continues on next page)

(continued from previous page)

```

_____ test_noop[mail.python.org] _____
smtp_connection = <smtpplib.SMTP object at 0xdeadbeef0005>

    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
        assert response == 250
>         assert 0 # for demo purposes
E         assert 0

test_module.py:13: AssertionError
----- Captured stdout teardown -----
finalizing <smtpplib.SMTP object at 0xdeadbeef0005>
===== short test summary info =====
FAILED test_module.py::test_ehlo[smtp.gmail.com] - assert 0
FAILED test_module.py::test_noop[smtp.gmail.com] - assert 0
FAILED test_module.py::test_ehlo[mail.python.org] - AssertionError: asser...
FAILED test_module.py::test_noop[mail.python.org] - assert 0
4 failed in 0.12s

```

We see that our two test functions each ran twice, against the different `smtp_connection` instances. Note also, that with the `mail.python.org` connection the second test fails in `test_ehlo` because a different server string is expected than what arrived.

pytest will build a string that is the test ID for each fixture value in a parametrized fixture, e.g. `test_ehlo[smtp.gmail.com]` and `test_ehlo[mail.python.org]` in the above examples. These IDs can be used with `-k` to select specific cases to run, and they will also identify the specific case when one is failing. Running pytest with `--collect-only` will show the generated IDs.

Numbers, strings, booleans and `None` will have their usual string representation used in the test ID. For other objects, pytest will make a string based on the argument name. It is possible to customise the string used in a test ID for a certain fixture value by using the `ids` keyword argument:

```

# content of test_ids.py
import pytest

@pytest.fixture(params=[0, 1], ids=["spam", "ham"])
def a(request):
    return request.param

def test_a(a):
    pass

def idfn(fixture_value):
    if fixture_value == 0:
        return "eggs"
    else:
        return None

@pytest.fixture(params=[0, 1], ids=idfn)
def b(request):
    return request.param

```

(continues on next page)

(continued from previous page)

```
def test_b(b):
    pass
```

The above shows how `ids` can be either a list of strings to use or a function which will be called with the fixture value and then has to return a string to use. In the latter case if the function returns `None` then pytest's auto-generated ID will be used.

Running the above tests results in the following test IDs being used:

```
$ pytest --collect-only
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 11 items

<Module test_anothersmtp.py>
  <Function test_showhelo[smtp.gmail.com]>
  <Function test_showhelo[mail.python.org]>
<Module test_emaillib.py>
  <Function test_email_received>
<Module test_ids.py>
  <Function test_a[spam]>
  <Function test_a[ham]>
  <Function test_b[eggs]>
  <Function test_b[1]>
<Module test_module.py>
  <Function test_ehlo[smtp.gmail.com]>
  <Function test_noop[smtp.gmail.com]>
  <Function test_ehlo[mail.python.org]>
  <Function test_noop[mail.python.org]>

===== 11 tests collected in 0.12s =====
```

2.3.11 Using marks with parametrized fixtures

`pytest.param()` can be used to apply marks in values sets of parametrized fixtures in the same way that they can be used with `@pytest.mark.parametrize`.

Example:

```
# content of test_fixture_marks.py
import pytest

@pytest.fixture(params=[0, 1, pytest.param(2, marks=pytest.mark.skip)])
def data_set(request):
    return request.param

def test_data(data_set):
    pass
```

Running this test will *skip* the invocation of `data_set` with value 2:

```
$ pytest test_fixture_marks.py -v
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 3 items

test_fixture_marks.py::test_data[0] PASSED [ 33%]
test_fixture_marks.py::test_data[1] PASSED [ 66%]
test_fixture_marks.py::test_data[2] SKIPPED (unconditional skip) [100%]

===== 2 passed, 1 skipped in 0.12s =====
```

2.3.12 Modularity: using fixtures from a fixture function

In addition to using fixtures in test functions, fixture functions can use other fixtures themselves. This contributes to a modular design of your fixtures and allows re-use of framework-specific fixtures across many projects. As a simple example, we can extend the previous example and instantiate an object `app` where we stick the already defined `smtp_connection` resource into it:

```
# content of test_appsetup.py

import pytest

class App:
    def __init__(self, smtp_connection):
        self.smtp_connection = smtp_connection

@pytest.fixture(scope="module")
def app(smtp_connection):
    return App(smtp_connection)

def test_smtp_connection_exists(app):
    assert app.smtp_connection
```

Here we declare an `app` fixture which receives the previously defined `smtp_connection` fixture and instantiates an `App` object with it. Let's run it:

```
$ pytest -v test_appsetup.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 2 items

test_appsetup.py::test_smtp_connection_exists[smtp.gmail.com] PASSED [ 50%]
test_appsetup.py::test_smtp_connection_exists[mail.python.org] PASSED [100%]

===== 2 passed in 0.12s =====
```

Due to the parametrization of `smtp_connection`, the test will run twice with two different `App` instances and re-

spective smtp servers. There is no need for the `app` fixture to be aware of the `smtp_connection` parametrization because `pytest` will fully analyse the fixture dependency graph.

Note that the `app` fixture has a scope of `module` and uses a module-scoped `smtp_connection` fixture. The example would still work if `smtp_connection` was cached on a `session` scope: it is fine for fixtures to use “broader” scoped fixtures but not the other way round: A session-scoped fixture could not use a module-scoped one in a meaningful way.

2.3.13 Automatic grouping of tests by fixture instances

`pytest` minimizes the number of active fixtures during test runs. If you have a parametrized fixture, then all the tests using it will first execute with one instance and then finalizers are called before the next fixture instance is created. Among other things, this eases testing of applications which create and use global state.

The following example uses two parametrized fixtures, one of which is scoped on a per-module basis, and all the functions perform `print` calls to show the setup/teardown flow:

```
# content of test_module.py
import pytest

@pytest.fixture(scope="module", params=["mod1", "mod2"])
def modarg(request):
    param = request.param
    print("  SETUP modarg", param)
    yield param
    print("  TEARDOWN modarg", param)

@pytest.fixture(scope="function", params=[1, 2])
def otherarg(request):
    param = request.param
    print("  SETUP otherarg", param)
    yield param
    print("  TEARDOWN otherarg", param)

def test_0(otherarg):
    print("  RUN test0 with otherarg", otherarg)

def test_1(modarg):
    print("  RUN test1 with modarg", modarg)

def test_2(otherarg, modarg):
    print("  RUN test2 with otherarg {} and modarg {}".format(otherarg, modarg))
```

Let’s run the tests in verbose mode and with looking at the print-output:

```
$ pytest -v -s test_module.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 8 items
```

(continues on next page)

(continued from previous page)

```

test_module.py::test_0[1]  SETUP otherarg 1
    RUN test0 with otherarg 1
PASSED  TEARDOWN otherarg 1

test_module.py::test_0[2]  SETUP otherarg 2
    RUN test0 with otherarg 2
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod1]  SETUP modarg mod1
    RUN test1 with modarg mod1
PASSED
test_module.py::test_2[mod1-1]  SETUP otherarg 1
    RUN test2 with otherarg 1 and modarg mod1
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[mod1-2]  SETUP otherarg 2
    RUN test2 with otherarg 2 and modarg mod1
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod2]  TEARDOWN modarg mod1
    SETUP modarg mod2
    RUN test1 with modarg mod2
PASSED
test_module.py::test_2[mod2-1]  SETUP otherarg 1
    RUN test2 with otherarg 1 and modarg mod2
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[mod2-2]  SETUP otherarg 2
    RUN test2 with otherarg 2 and modarg mod2
PASSED  TEARDOWN otherarg 2
    TEARDOWN modarg mod2

===== 8 passed in 0.12s =====

```

You can see that the parametrized module-scoped `modarg` resource caused an ordering of test execution that lead to the fewest possible “active” resources. The finalizer for the `mod1` parametrized resource was executed before the `mod2` resource was setup.

In particular notice that `test_0` is completely independent and finishes first. Then `test_1` is executed with `mod1`, then `test_2` with `mod1`, then `test_1` with `mod2` and finally `test_2` with `mod2`.

The `otherarg` parametrized resource (having function scope) was set up before and teared down after every test that used it.

2.3.14 Use fixtures in classes and modules with `usefixtures`

Sometimes test functions do not directly need access to a fixture object. For example, tests may require to operate with an empty directory as the current working directory but otherwise do not care for the concrete directory. Here is how you can use the standard `tempfile` and `pytest` fixtures to achieve it. We separate the creation of the fixture into a `conftest.py` file:

```

# content of conftest.py

import os

```

(continues on next page)

(continued from previous page)

```
import tempfile

import pytest

@pytest.fixture
def cleandir():
    with tempfile.TemporaryDirectory() as newpath:
        old_cwd = os.getcwd()
        os.chdir(newpath)
        yield
        os.chdir(old_cwd)
```

and declare its use in a test module via a usefixtures marker:

```
# content of test_setenv.py
import os
import pytest

@pytest.mark.usefixtures("cleandir")
class TestDirectoryInit:
    def test_cwd_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
        with open("myfile", "w") as f:
            f.write("hello")

    def test_cwd_again_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
```

Due to the usefixtures marker, the cleandir fixture will be required for the execution of each test method, just as if you specified a “cleandir” function argument to each of them. Let’s run it to verify our fixture is activated and the tests pass:

```
$ pytest -q
..                                     [100%]
2 passed in 0.12s
```

You can specify multiple fixtures like this:

```
@pytest.mark.usefixtures("cleandir", "anotherfixture")
def test():
    ...
```

and you may specify fixture usage at the test module level using `pytestmark`:

```
pytestmark = pytest.mark.usefixtures("cleandir")
```

It is also possible to put fixtures required by all tests in your project into an ini-file:

```
# content of pytest.ini
[pytest]
usefixtures = cleandir
```

Warning: Note this mark has no effect in **fixture functions**. For example, this **will not work as expected**:

```
@pytest.mark.usefixtures("my_other_fixture")
@pytest.fixture
def my_fixture_that_sadly_wont_use_my_other_fixture():
    ...
```

Currently this will not generate any error or warning, but this is intended to be handled by #3664.

2.3.15 Overriding fixtures on various levels

In relatively large test suite, you most likely need to override a global or root fixture with a locally defined one, keeping the test code readable and maintainable.

Override a fixture on a folder (conftest) level

Given the tests file structure is:

```
tests/
  __init__.py

  conftest.py
    # content of tests/conftest.py
    import pytest

    @pytest.fixture
    def username():
        return 'username'

  test_something.py
    # content of tests/test_something.py
    def test_username(username):
        assert username == 'username'

  subfolder/
    __init__.py

    conftest.py
      # content of tests/subfolder/conftest.py
      import pytest

      @pytest.fixture
      def username(username):
          return 'overridden-' + username

    test_something.py
      # content of tests/subfolder/test_something.py
      def test_username(username):
          assert username == 'overridden-username'
```

As you can see, a fixture with the same name can be overridden for certain test folder level. Note that the base or super fixture can be accessed from the overriding fixture easily - used in the example above.

Override a fixture on a test module level

Given the tests file structure is:

```
tests/
  __init__.py

  conftest.py
    # content of tests/conftest.py
    import pytest

    @pytest.fixture
    def username():
        return 'username'

  test_something.py
    # content of tests/test_something.py
    import pytest

    @pytest.fixture
    def username(username):
        return 'overridden-' + username

    def test_username(username):
        assert username == 'overridden-username'

  test_something_else.py
    # content of tests/test_something_else.py
    import pytest

    @pytest.fixture
    def username(username):
        return 'overridden-else-' + username

    def test_username(username):
        assert username == 'overridden-else-username'
```

In the example above, a fixture with the same name can be overridden for certain test module.

Override a fixture with direct test parametrization

Given the tests file structure is:

```
tests/
  __init__.py

  conftest.py
    # content of tests/conftest.py
    import pytest

    @pytest.fixture
    def username():
        return 'username'

    @pytest.fixture
    def other_username(username):
        return 'other-' + username
```

(continues on next page)

(continued from previous page)

```

test_something.py
# content of tests/test_something.py
import pytest

@pytest.mark.parametrize('username', ['directly-overridden-username'])
def test_username(username):
    assert username == 'directly-overridden-username'

@pytest.mark.parametrize('username', ['directly-overridden-username-other'])
def test_username_other(other_username):
    assert other_username == 'other-directly-overridden-username-other'

```

In the example above, a fixture value is overridden by the test parameter value. Note that the value of the fixture can be overridden this way even if the test doesn't use it directly (doesn't mention it in the function prototype).

Override a parametrized fixture with non-parametrized one and vice versa

Given the tests file structure is:

```

tests/
  __init__.py

conftest.py
# content of tests/conftest.py
import pytest

@pytest.fixture(params=['one', 'two', 'three'])
def parametrized_username(request):
    return request.param

@pytest.fixture
def non_parametrized_username(request):
    return 'username'

test_something.py
# content of tests/test_something.py
import pytest

@pytest.fixture
def parametrized_username():
    return 'overridden-username'

@pytest.fixture(params=['one', 'two', 'three'])
def non_parametrized_username(request):
    return request.param

def test_username(parametrized_username):
    assert parametrized_username == 'overridden-username'

def test_parametrized_username(non_parametrized_username):
    assert non_parametrized_username in ['one', 'two', 'three']

test_something_else.py
# content of tests/test_something_else.py

```

(continues on next page)

(continued from previous page)

```
def test_username(parametrized_username):
    assert parametrized_username in ['one', 'two', 'three']

def test_username(non_parametrized_username):
    assert non_parametrized_username == 'username'
```

In the example above, a parametrized fixture is overridden with a non-parametrized version, and a non-parametrized fixture is overridden with a parametrized version for certain test module. The same applies for the test folder level obviously.

2.3.16 Using fixtures from other projects

Usually projects that provide pytest support will use *entry points*, so just installing those projects into an environment will make those fixtures available for use.

In case you want to use fixtures from a project that does not use entry points, you can define *pytest_plugins* in your top `conftest.py` file to register that module as a plugin.

Suppose you have some fixtures in `mylibrary.fixtures` and you want to reuse them into your `app/tests` directory.

All you need to do is to define *pytest_plugins* in `app/tests/conftest.py` pointing to that module.

```
pytest_plugins = "mylibrary.fixtures"
```

This effectively registers `mylibrary.fixtures` as a plugin, making all its fixtures and hooks available to tests in `app/tests`.

Note: Sometimes users will *import* fixtures from other projects for use, however this is not recommended: importing fixtures into a module will register them in pytest as *defined* in that module.

This has minor consequences, such as appearing multiple times in `pytest --help`, but it is not **recommended** because this behavior might change/stop working in future versions.

2.4 How to mark test functions with attributes

By using the `pytest.mark` helper you can easily set metadata on your test functions. You can find the full list of builtin markers in the *API Reference*. Or you can list all the markers, including builtin and custom, using the CLI - `pytest --markers`.

Here are some of the builtin markers:

- *usefixtures* - use fixtures on a test function or class
- *filterwarnings* - filter certain warnings of a test function
- *skip* - always skip a test function
- *skipif* - skip a test function if a certain condition is met
- *xfail* - produce an “expected failure” outcome if a certain condition is met
- *parametrize* - perform multiple calls to the same test function.

It's easy to create custom markers or to apply markers to whole test classes or modules. Those markers can be used by plugins, and also are commonly used to *select tests* on the command-line with the `-m` option.

See *Working with custom markers* for examples which also serve as documentation.

Note: Marks can only be applied to tests, having no effect on *fixtures*.

2.4.1 Registering marks

You can register custom marks in your `pytest.ini` file like this:

```
[pytest]
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    serial
```

or in your `pyproject.toml` file like this:

```
[tool.pytest.ini_options]
markers = [
    "slow: marks tests as slow (deselect with '-m \"not slow\"')",
    "serial",
]
```

Note that everything past the `:` after the mark name is an optional description.

Alternatively, you can register new markers programmatically in a `pytest_configure` hook:

```
def pytest_configure(config):
    config.addinvalue_line(
        "markers", "env(name): mark test to run only on named environment"
    )
```

Registered marks appear in pytest's help text and do not emit warnings (see the next section). It is recommended that third-party plugins always *register their markers*.

2.4.2 Raising errors on unknown marks

Unregistered marks applied with the `@pytest.mark.name_of_the_mark` decorator will always emit a warning in order to avoid silently doing something surprising due to mistyped names. As described in the previous section, you can disable the warning for custom marks by registering them in your `pytest.ini` file or using a custom `pytest_configure` hook.

When the `--strict-markers` command-line flag is passed, any unknown marks applied with the `@pytest.mark.name_of_the_mark` decorator will trigger an error. You can enforce this validation in your project by adding `--strict-markers` to `addopts`:

```
[pytest]
addopts = --strict-markers
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    serial
```

2.5 How to parametrize fixtures and test functions

pytest enables test parametrization at several levels:

- `pytest.fixture()` allows one to *parametrize fixture functions*.
- `@pytest.mark.parametrize` allows one to define multiple sets of arguments and fixtures at the test function or class.
- `pytest_generate_tests` allows one to define custom parametrization schemes or extensions.

2.5.1 @pytest.mark.parametrize: parametrizing test functions

The builtin `pytest.mark.parametrize` decorator enables parametrization of arguments for a test function. Here is a typical example of a test function that implements checking that a certain input leads to an expected output:

```
# content of test_expectation.py
import pytest

@pytest.mark.parametrize("test_input,expected", [
    ("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Here, the `@parametrize` decorator defines three different `(test_input,expected)` tuples so that the `test_eval` function will run three times using them in turn:

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 3 items

test_expectation.py ..F                                     [100%]

===== FAILURES =====
_____ test_eval[6*9-42] _____

test_input = '6*9', expected = 42

    @pytest.mark.parametrize("test_input,expected", [
    ("3+5", 8), ("2+4", 6), ("6*9", 4
    ↪42)])
    def test_eval(test_input, expected):
>     assert eval(test_input) == expected
E       AssertionError: assert 54 == 42
E         + where 54 = eval('6*9')

test_expectation.py:6: AssertionError
===== short test summary info =====
FAILED test_expectation.py::test_eval[6*9-42] - AssertionError: assert 54...
===== 1 failed, 2 passed in 0.12s =====
```

Note: Parameter values are passed as-is to tests (no copy whatsoever).

For example, if you pass a list or a dict as a parameter value, and the test case code mutates it, the mutations will be reflected in subsequent test case calls.

Note: pytest by default escapes any non-ascii characters used in unicode strings for the parametrization because it has several downsides. If however you would like to use unicode strings in parametrization and see them in the terminal as is (non-escaped), use this option in your `pytest.ini`:

```
[pytest]
disable_test_id_escaping_and_forfeit_all_rights_to_community_support = True
```

Keep in mind however that this might cause unwanted side effects and even bugs depending on the OS used and plugins currently installed, so use it at your own risk.

As designed in this example, only one pair of input/output values fails the simple test function. And as usual with test function arguments, you can see the `input` and `output` values in the traceback.

Note that you could also use the `parametrize` marker on a class or a module (see *How to mark test functions with attributes*) which would invoke several functions with the argument sets, for instance:

```
import pytest

@pytest.mark.parametrize("n,expected", [(1, 2), (3, 4)])
class TestClass:
    def test_simple_case(self, n, expected):
        assert n + 1 == expected

    def test_weird_simple_case(self, n, expected):
        assert (n * 1) + 1 == expected
```

To parametrize all tests in a module, you can assign to the `pytestmark` global variable:

```
import pytest

pytestmark = pytest.mark.parametrize("n,expected", [(1, 2), (3, 4)])

class TestClass:
    def test_simple_case(self, n, expected):
        assert n + 1 == expected

    def test_weird_simple_case(self, n, expected):
        assert (n * 1) + 1 == expected
```

It is also possible to mark individual test instances within `parametrize`, for example with the builtin `mark.xfail`:

```
# content of test_expectation.py
import pytest

@pytest.mark.parametrize(
    "test_input,expected",
    [("3+5", 8), ("2+4", 6), pytest.param("6*9", 42, marks=pytest.mark.xfail)],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Let's run this:

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 3 items

test_expectation.py ..x [100%]

===== 2 passed, 1 xfailed in 0.12s =====
```

The one parameter set which caused a failure previously now shows up as an “xfailed” (expected to fail) test.

In case the values provided to `parametrize` result in an empty list - for example, if they’re dynamically generated by some function - the behaviour of pytest is defined by the `empty_parameter_set_mark` option.

To get all combinations of multiple parametrized arguments you can stack `parametrize` decorators:

```
import pytest

@pytest.mark.parametrize("x", [0, 1])
@pytest.mark.parametrize("y", [2, 3])
def test_foo(x, y):
    pass
```

This will run the test with the arguments set to `x=0/y=2`, `x=1/y=2`, `x=0/y=3`, and `x=1/y=3` exhausting parameters in the order of the decorators.

2.5.2 Basic `pytest_generate_tests` example

Sometimes you may want to implement your own parametrization scheme or implement some dynamism for determining the parameters or scope of a fixture. For this, you can use the `pytest_generate_tests` hook which is called when collecting a test function. Through the passed in `metafunc` object you can inspect the requesting test context and, most importantly, you can call `metafunc.parametrize()` to cause parametrization.

For example, let’s say we want to run a test taking string inputs which we want to set via a new `pytest` command line option. Let’s first write a simple test accepting a `stringinput` fixture function argument:

```
# content of test_strings.py

def test_valid_string(stringinput):
    assert stringinput.isalpha()
```

Now we add a `conftest.py` file containing the addition of a command line option and the parametrization of our test function:

```
# content of conftest.py

def pytest_addoption(parser):
    parser.addoption(
        "--stringinput",
        action="append",
        default=[],
```

(continues on next page)

(continued from previous page)

```

        help="list of stringinputs to pass to test functions",
    )

def pytest_generate_tests(metafunc):
    if "stringinput" in metafunc.fixturenames:
        metafunc.parametrize("stringinput", metafunc.config.getoption("stringinput"))

```

If we now pass two stringinput values, our test will run twice:

```

$ pytest -q --stringinput="hello" --stringinput="world" test_strings.py
.. [100%]
2 passed in 0.12s

```

Let's also run with a stringinput that will lead to a failing test:

```

$ pytest -q --stringinput="!" test_strings.py
F [100%]
===== FAILURES =====
_____ test_valid_string[!] _____

stringinput = '!'

    def test_valid_string(stringinput):
>     assert stringinput.isalpha()
E     AssertionError: assert False
E     + where False = <built-in method isalpha of str object at 0xdeadbeef0001>()
E     +   where <built-in method isalpha of str object at 0xdeadbeef0001> = '!'.
->isalpha

test_strings.py:4: AssertionError
===== short test summary info =====
FAILED test_strings.py::test_valid_string[!] - AssertionError: assert False
1 failed in 0.12s

```

As expected our test function fails.

If you don't specify a stringinput it will be skipped because `metafunc.parametrize()` will be called with an empty parameter list:

```

$ pytest -q -rs test_strings.py
s [100%]
===== short test summary info =====
SKIPPED [1] test_strings.py: got empty parameter set ['stringinput'], function test_
->valid_string at /home/sweet/project/test_strings.py:2
1 skipped in 0.12s

```

Note that when calling `metafunc.parametrize` multiple times with different parameter sets, all parameter names across those sets cannot be duplicated, otherwise an error will be raised.

2.5.3 More examples

For further examples, you might want to look at *more parametrization examples*.

2.6 How to use temporary directories and files in tests

2.6.1 The `tmp_path` fixture

You can use the `tmp_path` fixture which will provide a temporary directory unique to the test invocation, created in the *base temporary directory*.

`tmp_path` is a `pathlib.Path` object. Here is an example test usage:

```
# content of test_tmp_path.py
CONTENT = "content"

def test_create_file(tmp_path):
    d = tmp_path / "sub"
    d.mkdir()
    p = d / "hello.txt"
    p.write_text(CONTENT)
    assert p.read_text() == CONTENT
    assert len(list(tmp_path.iterdir())) == 1
    assert 0
```

Running this would result in a passed test except for the last `assert 0` line which we use to look at values:

```
$ pytest test_tmp_path.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_tmp_path.py F [100%]

===== FAILURES =====
_____ test_create_file _____

tmp_path = PosixPath('PYTEST_TMPDIR/test_create_file0')

    def test_create_file(tmp_path):
        d = tmp_path / "sub"
        d.mkdir()
        p = d / "hello.txt"
        p.write_text(CONTENT)
        assert p.read_text() == CONTENT
        assert len(list(tmp_path.iterdir())) == 1
>       assert 0
E       assert 0

test_tmp_path.py:11: AssertionError
===== short test summary info =====
FAILED test_tmp_path.py::test_create_file - assert 0
===== 1 failed in 0.12s =====
```

2.6.2 The `tmp_path_factory` fixture

The `tmp_path_factory` is a session-scoped fixture which can be used to create arbitrary temporary directories from any other fixture or test.

For example, suppose your test suite needs a large image on disk, which is generated procedurally. Instead of computing the same image for each test that uses it into its own `tmp_path`, you can generate it once per-session to save time:

```
# contents of conftest.py
import pytest

@pytest.fixture(scope="session")
def image_file(tmp_path_factory):
    img = compute_expensive_image()
    fn = tmp_path_factory.mktemp("data") / "img.png"
    img.save(fn)
    return fn

# contents of test_image.py
def test_histogram(image_file):
    img = load_image(image_file)
    # compute and test histogram
```

See `tmp_path_factory` API for details.

2.6.3 The `tmpdir` and `tmpdir_factory` fixtures

The `tmpdir` and `tmpdir_factory` fixtures are similar to `tmp_path` and `tmp_path_factory`, but use/return legacy `py.path.local` objects rather than standard `pathlib.Path` objects. These days, prefer to use `tmp_path` and `tmp_path_factory`.

See `tmpdir` `tmpdir_factory` API for details.

2.6.4 The default base temporary directory

Temporary directories are by default created as sub-directories of the system temporary directory. The base name will be `pytest-NUM` where *NUM* will be incremented with each test run. Moreover, entries older than 3 temporary directories will be removed.

You can override the default temporary directory setting like this:

```
pytest --basetemp=mydir
```

Warning: The contents of `mydir` will be completely removed, so make sure to use a directory for that purpose only.

When distributing tests on the local machine using `pytest-xdist`, care is taken to automatically configure a `basetemp` directory for the sub processes such that all temporary data lands below a single per-test run `basetemp` directory.

2.7 How to monkeypatch/mock modules and environments

Sometimes tests need to invoke functionality which depends on global settings or which invokes code which cannot be easily tested such as network access. The `monkeypatch` fixture helps you to safely set/delete an attribute, dictionary item or environment variable, or to modify `sys.path` for importing.

The `monkeypatch` fixture provides these helper methods for safely patching and mocking functionality in tests:

```
monkeypatch.setattr(obj, name, value, raising=True)
monkeypatch.setattr("somemodule.obj.name", value, raising=True)
monkeypatch.delattr(obj, name, raising=True)
monkeypatch.setitem(mapping, name, value)
monkeypatch.delitem(obj, name, raising=True)
monkeypatch.setenv(name, value, prepend=None)
monkeypatch.delenv(name, raising=True)
monkeypatch.syspath_prepend(path)
monkeypatch.chdir(path)
```

All modifications will be undone after the requesting test function or fixture has finished. The `raising` parameter determines if a `KeyError` or `AttributeError` will be raised if the target of the set/deletion operation does not exist.

Consider the following scenarios:

1. Modifying the behavior of a function or the property of a class for a test e.g. there is an API call or database connection you will not make for a test but you know what the expected output should be. Use `monkeypatch.setattr` to patch the function or property with your desired testing behavior. This can include your own functions. Use `monkeypatch.delattr` to remove the function or property for the test.
2. Modifying the values of dictionaries e.g. you have a global configuration that you want to modify for certain test cases. Use `monkeypatch.setitem` to patch the dictionary for the test. `monkeypatch.delitem` can be used to remove items.
3. Modifying environment variables for a test e.g. to test program behavior if an environment variable is missing, or to set multiple values to a known variable. `monkeypatch.setenv` and `monkeypatch.delenv` can be used for these patches.
4. Use `monkeypatch.setenv("PATH", value, prepend=os.pathsep)` to modify `$PATH`, and `monkeypatch.chdir` to change the context of the current working directory during a test.
5. Use `monkeypatch.syspath_prepend` to modify `sys.path` which will also call `pkg_resources.fixup_namespace_packages` and `importlib.invalidate_caches()`.

See the [monkeypatch blog post](#) for some introduction material and a discussion of its motivation.

2.7.1 Simple example: monkeypatching functions

Consider a scenario where you are working with user directories. In the context of testing, you do not want your test to depend on the running user. `monkeypatch` can be used to patch functions dependent on the user to always return a specific value.

In this example, `monkeypatch.setattr` is used to patch `Path.home` so that the known testing path `Path("/abc")` is always used when the test is run. This removes any dependency on the running user for testing purposes. `monkeypatch.setattr` must be called before the function which will use the patched function is called. After the test function finishes the `Path.home` modification will be undone.

```
# contents of test_module.py with source code and the test
from pathlib import Path

def getssh():
    """Simple function to return expanded homedir ssh path."""
    return Path.home() / ".ssh"

def test_getssh(monkeypatch):
    # mocked return function to replace Path.home
    # always return '/abc'
    def mockreturn():
        return Path("/abc")

    # Application of the monkeypatch to replace Path.home
    # with the behavior of mockreturn defined above.
    monkeypatch.setattr(Path, "home", mockreturn)

    # Calling getssh() will use mockreturn in place of Path.home
    # for this test with the monkeypatch.
    x = getssh()
    assert x == Path("/abc/.ssh")
```

2.7.2 Monkeypatching returned objects: building mock classes

`monkeypatch.setattr` can be used in conjunction with classes to mock returned objects from functions instead of values. Imagine a simple function to take an API url and return the json response.

```
# contents of app.py, a simple API retrieval example
import requests

def get_json(url):
    """Takes a URL, and returns the JSON."""
    r = requests.get(url)
    return r.json()
```

We need to mock `r`, the returned response object for testing purposes. The mock of `r` needs a `.json()` method which returns a dictionary. This can be done in our test file by defining a class to represent `r`.

```
# contents of test_app.py, a simple test for our API retrieval
# import requests for the purposes of monkeypatching
import requests

# our app.py that includes the get_json() function
# this is the previous code block example
import app

# custom class to be the mock return value
# will override the requests.Response returned from requests.get
class MockResponse:

    # mock json() method always returns a specific testing dictionary
    @staticmethod
```

(continues on next page)

(continued from previous page)

```

def json():
    return {"mock_key": "mock_response"}

def test_get_json(monkeypatch):

    # Any arguments may be passed and mock_get() will always return our
    # mocked object, which only has the .json() method.
    def mock_get(*args, **kwargs):
        return MockResponse()

    # apply the monkeypatch for requests.get to mock_get
    monkeypatch.setattr(requests, "get", mock_get)

    # app.get_json, which contains requests.get, uses the monkeypatch
    result = app.get_json("https://fakeurl")
    assert result["mock_key"] == "mock_response"

```

monkeypatch applies the mock for `requests.get` with our `mock_get` function. The `mock_get` function returns an instance of the `MockResponse` class, which has a `json()` method defined to return a known testing dictionary and does not require any outside API connection.

You can build the `MockResponse` class with the appropriate degree of complexity for the scenario you are testing. For instance, it could include an `ok` property that always returns `True`, or return different values from the `json()` mocked method based on input strings.

This mock can be shared across tests using a fixture:

```

# contents of test_app.py, a simple test for our API retrieval
import pytest
import requests

# app.py that includes the get_json() function
import app

# custom class to be the mock return value of requests.get()
class MockResponse:
    @staticmethod
    def json():
        return {"mock_key": "mock_response"}

# monkeypatched requests.get moved to a fixture
@pytest.fixture
def mock_response(monkeypatch):
    """Requests.get() mocked to return {'mock_key': 'mock_response'}."""

    def mock_get(*args, **kwargs):
        return MockResponse()

    monkeypatch.setattr(requests, "get", mock_get)

# notice our test uses the custom fixture instead of monkeypatch directly
def test_get_json(mock_response):
    result = app.get_json("https://fakeurl")
    assert result["mock_key"] == "mock_response"

```

Furthermore, if the mock was designed to be applied to all tests, the fixture could be moved to a `conftest.py` file and use the `with autouse=True` option.

2.7.3 Global patch example: preventing “requests” from remote operations

If you want to prevent the “requests” library from performing http requests in all your tests, you can do:

```
# contents of conftest.py
import pytest

@pytest.fixture(autouse=True)
def no_requests(monkeypatch):
    """Remove requests.sessions.Session.request for all tests."""
    monkeypatch.delattr("requests.sessions.Session.request")
```

This `autouse` fixture will be executed for each test function and it will delete the method `request.session.Session.request` so that any attempts within tests to create http requests will fail.

Note: Be advised that it is not recommended to patch builtin functions such as `open`, `compile`, etc., because it might break pytest’s internals. If that’s unavoidable, passing `--tb=native`, `--assert=plain` and `--capture=no` might help although there’s no guarantee.

Note: Mind that patching `stdlib` functions and some third-party libraries used by pytest might break pytest itself, therefore in those cases it is recommended to use `MonkeyPatch.context()` to limit the patching to the block you want tested:

```
import functools

def test_partial(monkeypatch):
    with monkeypatch.context() as m:
        m.setattr(functools, "partial", 3)
        assert functools.partial == 3
```

See issue #3290 for details.

2.7.4 Monkeypatching environment variables

If you are working with environment variables you often need to safely change the values or delete them from the system for testing purposes. `monkeypatch` provides a mechanism to do this using the `setenv` and `delenv` method. Our example code to test:

```
# contents of our original code file e.g. code.py
import os

def get_os_user_lower():
    """Simple retrieval function.
    Returns lowercase USER or raises OSError."""
    username = os.getenv("USER")
```

(continues on next page)

(continued from previous page)

```

if username is None:
    raise OSError("USER environment is not set.")

return username.lower()

```

There are two potential paths. First, the `USER` environment variable is set to a value. Second, the `USER` environment variable does not exist. Using `monkeypatch` both paths can be safely tested without impacting the running environment:

```

# contents of our test file e.g. test_code.py
import pytest

def test_upper_to_lower(monkeypatch):
    """Set the USER env var to assert the behavior."""
    monkeypatch.setenv("USER", "TestingUser")
    assert get_os_user_lower() == "testinguser"

def test_raise_exception(monkeypatch):
    """Remove the USER env var and assert OSError is raised."""
    monkeypatch.delenv("USER", raising=False)

    with pytest.raises(OSError):
        _ = get_os_user_lower()

```

This behavior can be moved into fixture structures and shared across tests:

```

# contents of our test file e.g. test_code.py
import pytest

@pytest.fixture
def mock_env_user(monkeypatch):
    monkeypatch.setenv("USER", "TestingUser")

@pytest.fixture
def mock_env_missing(monkeypatch):
    monkeypatch.delenv("USER", raising=False)

# notice the tests reference the fixtures for mocks
def test_upper_to_lower(mock_env_user):
    assert get_os_user_lower() == "testinguser"

def test_raise_exception(mock_env_missing):
    with pytest.raises(OSError):
        _ = get_os_user_lower()

```

2.7.5 Monkeypatching dictionaries

`monkeypatch.setitem` can be used to safely set the values of dictionaries to specific values during tests. Take this simplified connection string example:

```
# contents of app.py to generate a simple connection string
DEFAULT_CONFIG = {"user": "user1", "database": "db1"}

def create_connection_string(config=None):
    """Creates a connection string from input or defaults."""
    config = config or DEFAULT_CONFIG
    return f"User Id={config['user']}; Location={config['database']};"
```

For testing purposes we can patch the `DEFAULT_CONFIG` dictionary to specific values.

```
# contents of test_app.py
# app.py with the connection string function (prior code block)
import app

def test_connection(monkeypatch):

    # Patch the values of DEFAULT_CONFIG to specific
    # testing values only for this test.
    monkeypatch.setitem(app.DEFAULT_CONFIG, "user", "test_user")
    monkeypatch.setitem(app.DEFAULT_CONFIG, "database", "test_db")

    # expected result based on the mocks
    expected = "User Id=test_user; Location=test_db;"

    # the test uses the monkeypatched dictionary settings
    result = app.create_connection_string()
    assert result == expected
```

You can use the `monkeypatch.delitem` to remove values.

```
# contents of test_app.py
import pytest

# app.py with the connection string function
import app

def test_missing_user(monkeypatch):

    # patch the DEFAULT_CONFIG to be missing the 'user' key
    monkeypatch.delitem(app.DEFAULT_CONFIG, "user", raising=False)

    # Key error expected because a config is not passed, and the
    # default is now missing the 'user' entry.
    with pytest.raises(KeyError):
        _ = app.create_connection_string()
```

The modularity of fixtures gives you the flexibility to define separate fixtures for each potential mock and reference them in the needed tests.

```

# contents of test_app.py
import pytest

# app.py with the connection string function
import app

# all of the mocks are moved into separated fixtures
@pytest.fixture
def mock_test_user(monkeypatch):
    """Set the DEFAULT_CONFIG user to test_user."""
    monkeypatch.setitem(app.DEFAULT_CONFIG, "user", "test_user")

@pytest.fixture
def mock_test_database(monkeypatch):
    """Set the DEFAULT_CONFIG database to test_db."""
    monkeypatch.setitem(app.DEFAULT_CONFIG, "database", "test_db")

@pytest.fixture
def mock_missing_default_user(monkeypatch):
    """Remove the user key from DEFAULT_CONFIG"""
    monkeypatch.delitem(app.DEFAULT_CONFIG, "user", raising=False)

# tests reference only the fixture mocks that are needed
def test_connection(mock_test_user, mock_test_database):

    expected = "User Id=test_user; Location=test_db;"

    result = app.create_connection_string()
    assert result == expected

def test_missing_user(mock_missing_default_user):

    with pytest.raises(KeyError):
        _ = app.create_connection_string()

```

2.7.6 API Reference

Consult the docs for the *MonkeyPatch* class.

2.8 How to run doctests

By default, all files matching the `test*.txt` pattern will be run through the python standard `doctest` module. You can change the pattern by issuing:

```
pytest --doctest-glob="*.rst"
```

on the command line. `--doctest-glob` can be given multiple times in the command-line.

If you then have a text file like this:

```
# content of test_example.txt

hello this is a doctest
>>> x = 3
>>> x
3
```

then you can just invoke `pytest` directly:

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_example.txt .                               [100%]

===== 1 passed in 0.12s =====
```

By default, `pytest` will collect `test*.txt` files looking for doctest directives, but you can pass additional globs using the `--doctest-glob` option (multi-allowed).

In addition to text files, you can also execute doctests directly from docstrings of your classes and functions, including from test modules:

```
# content of mymodule.py
def something():
    """a doctest in a docstring
    >>> something()
    42
    """
    return 42
```

```
$ pytest --doctest-modules
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items

mymodule.py .                                   [ 50%]
test_example.txt .                             [100%]

===== 2 passed in 0.12s =====
```

You can make these changes permanent in your project by putting them into a `pytest.ini` file like this:

```
# content of pytest.ini
[pytest]
addopts = --doctest-modules
```


2.8.1 Encoding

The default encoding is **UTF-8**, but you can specify the encoding that will be used for those doctest files using the `doctest_encoding` ini option:

```
# content of pytest.ini
[pytest]
doctest_encoding = latin1
```

2.8.2 Using ‘doctest’ options

Python’s standard `doctest` module provides some `options` to configure the strictness of doctest tests. In `pytest`, you can enable those flags using the configuration file.

For example, to make `pytest` ignore trailing whitespaces and ignore lengthy exception stack traces you can just write:

```
[pytest]
doctest_optionflags = NORMALIZE_WHITESPACE IGNORE_EXCEPTION_DETAIL
```

Alternatively, options can be enabled by an inline comment in the doc test itself:

```
>>> something_that_raises() # doctest: +IGNORE_EXCEPTION_DETAIL
Traceback (most recent call last):
ValueError: ...
```

`pytest` also introduces new options:

- `ALLOW_UNICODE`: when enabled, the `u` prefix is stripped from unicode strings in expected doctest output. This allows doctests to run in Python 2 and Python 3 unchanged.
- `ALLOW_BYTES`: similarly, the `b` prefix is stripped from byte strings in expected doctest output.
- `NUMBER`: when enabled, floating-point numbers only need to match as far as the precision you have written in the expected doctest output. For example, the following output would only need to match to 2 decimal places:

```
>>> math.pi
3.14
```

If you wrote `3.1416` then the actual output would need to match to 4 decimal places; and so on.

This avoids false positives caused by limited floating-point precision, like this:

```
Expected:
 0.233
Got:
 0.233000000000000001
```

`NUMBER` also supports lists of floating-point numbers – in fact, it matches floating-point numbers appearing anywhere in the output, even inside a string! This means that it may not be appropriate to enable globally in `doctest_optionflags` in your configuration file.

New in version 5.1.

2.8.3 Continue on failure

By default, pytest would report only the first failure for a given doctest. If you want to continue the test even when you have failures, do:

```
pytest --doctest-modules --doctest-continue-on-failure
```

2.8.4 Output format

You can change the diff output format on failure for your doctests by using one of standard doctest modules format in options (see `doctest.REPORT_UDIFF`, `doctest.REPORT_CDIF`, `doctest.REPORT_NDIFF`, `doctest.REPORT_ONLY_FIRST_FAILURE`):

```
pytest --doctest-modules --doctest-report none
pytest --doctest-modules --doctest-report udiff
pytest --doctest-modules --doctest-report cdiff
pytest --doctest-modules --doctest-report ndiff
pytest --doctest-modules --doctest-report only_first_failure
```

2.8.5 pytest-specific features

Some features are provided to make writing doctests easier or with better integration with your existing test suite. Keep in mind however that by using those features you will make your doctests incompatible with the standard `doctests` module.

Using fixtures

It is possible to use fixtures using the `getfixture` helper:

```
# content of example.rst
>>> tmp = getfixture('tmp_path')
>>> ...
>>>
```

Note that the fixture needs to be defined in a place visible by pytest, for example, a `conftest.py` file or plugin; normal python files containing docstrings are not normally scanned for fixtures unless explicitly configured by `python_files`.

Also, the `usefixtures` mark and fixtures marked as `autouse` are supported when executing text doctest files.

'doctest_namespace' fixture

The `doctest_namespace` fixture can be used to inject items into the namespace in which your doctests run. It is intended to be used within your own fixtures to provide the tests that use them with context.

`doctest_namespace` is a standard `dict` object into which you place the objects you want to appear in the doctest namespace:

```
# content of conftest.py
import numpy

@pytest.fixture(autouse=True)
```

(continues on next page)

(continued from previous page)

```
def add_np(doctest_namespace):
    doctest_namespace["np"] = numpy
```

which can then be used in your doctests directly:

```
# content of numpy.py
def arange():
    """
    >>> a = np.arange(10)
    >>> len(a)
    10
    """
    pass
```

Note that like the normal `confest.py`, the fixtures are discovered in the directory tree `confest` is in. Meaning that if you put your doctest with your source code, the relevant `confest.py` needs to be in the same directory tree. Fixtures will not be discovered in a sibling directory tree!

Skipping tests

For the same reasons one might want to skip normal tests, it is also possible to skip tests inside doctests.

To skip a single check inside a doctest you can use the standard `doctest.SKIP` directive:

```
def test_random(y):
    """
    >>> random.random() # doctest: +SKIP
    0.156231223

    >>> 1 + 1
    2
    """
```

This will skip the first check, but not the second.

pytest also allows using the standard pytest functions `pytest.skip()` and `pytest.xfail()` inside doctests, which might be useful because you can then skip/xfail tests based on external conditions:

```
>>> import sys, pytest
>>> if sys.platform.startswith('win'):
...     pytest.skip('this doctest does not work on Windows')
...
>>> import fcntl
>>> ...
```

However using those functions is discouraged because it reduces the readability of the docstring.

Note: `pytest.skip()` and `pytest.xfail()` behave differently depending if the doctests are in a Python file (in docstrings) or a text file containing doctests intermingled with text:

- Python modules (docstrings): the functions only act in that specific docstring, letting the other docstrings in the same module execute as normal.
- Text files: the functions will skip/xfail the checks for the rest of the entire file.

2.8.6 Alternatives

While the built-in pytest support provides a good set of functionalities for using doctests, if you use them extensively you might be interested in those external packages which add many more features, and include pytest integration:

- `pytest-doctestplus`: provides advanced doctest support and enables the testing of reStructuredText (“.rst”) files.
- `Sybil`: provides a way to test examples in your documentation by parsing them from the documentation source and evaluating the parsed examples as part of your normal test run.

2.9 How to re-run failed tests and maintain state between test runs

2.9.1 Usage

The plugin provides two command line options to rerun failures from the last `pytest` invocation:

- `--lf, --last-failed` - to only re-run the failures.
- `--ff, --failed-first` - to run the failures first and then the rest of the tests.

For cleanup (usually not needed), a `--cache-clear` option allows to remove all cross-session cache contents ahead of a test run.

Other plugins may access the `config.cache` object to set/get **json encodable** values between `pytest` invocations.

Note: This plugin is enabled by default, but can be disabled if needed: see *Deactivating / unregistering a plugin by name* (the internal name for this plugin is `cacheprovider`).

2.9.2 Rerunning only failures or failures first

First, let’s create 50 test invocation of which only 2 fail:

```
# content of test_50.py
import pytest

@pytest.mark.parametrize("i", range(50))
def test_num(i):
    if i in (17, 25):
        pytest.fail("bad luck")
```

If you run this for the first time you will see two failures:

```
$ pytest -q
.....F.....F..... [100%]
===== FAILURES =====
_____ test_num[17] _____

i = 17

@pytest.mark.parametrize("i", range(50))
def test_num(i):
    if i in (17, 25):
```

(continues on next page)

(continued from previous page)

```
>         pytest.fail("bad luck")
E         Failed: bad luck

test_50.py:7: Failed
_____ test_num[25] _____

i = 25

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>         pytest.fail("bad luck")
E         Failed: bad luck

test_50.py:7: Failed
===== short test summary info =====
FAILED test_50.py::test_num[17] - Failed: bad luck
FAILED test_50.py::test_num[25] - Failed: bad luck
2 failed, 48 passed in 0.12s
```

If you then run it with `--lf`:

```
$ pytest --lf
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items
run-last-failure: rerun previous 2 failures

test_50.py FF [100%]

===== FAILURES =====
_____ test_num[17] _____

i = 17

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>         pytest.fail("bad luck")
E         Failed: bad luck

test_50.py:7: Failed
_____ test_num[25] _____

i = 25

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>         pytest.fail("bad luck")
E         Failed: bad luck

test_50.py:7: Failed
===== short test summary info =====
FAILED test_50.py::test_num[17] - Failed: bad luck
```

(continues on next page)

(continued from previous page)

```
FAILED test_50.py::test_num[25] - Failed: bad luck
===== 2 failed in 0.12s =====
```

You have run only the two failing tests from the last run, while the 48 passing tests have not been run (“deselected”).

Now, if you run with the `--ff` option, all tests will be run but the first previous failures will be executed first (as can be seen from the series of FF and dots):

```
$ pytest --ff
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 50 items
run-last-failure: rerun previous 2 failures first

test_50.py FF..... [100%]

===== FAILURES =====
_____ test_num[17] _____

i = 17

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>         pytest.fail("bad luck")
E         Failed: bad luck

test_50.py:7: Failed
_____ test_num[25] _____

i = 25

    @pytest.mark.parametrize("i", range(50))
    def test_num(i):
        if i in (17, 25):
>         pytest.fail("bad luck")
E         Failed: bad luck

test_50.py:7: Failed
===== short test summary info =====
FAILED test_50.py::test_num[17] - Failed: bad luck
FAILED test_50.py::test_num[25] - Failed: bad luck
===== 2 failed, 48 passed in 0.12s =====
```

New `--nf`, `--new-first` options: run new tests first followed by the rest of the tests, in both cases tests are also sorted by the file modified time, with more recent files coming first.

2.9.3 Behavior when no tests failed in the last run

When no tests failed in the last run, or when no cached `lastfailed` data was found, `pytest` can be configured either to run all of the tests or no tests, using the `--last-failed-no-failures` option, which takes one of the following values:

```
pytest --last-failed --last-failed-no-failures all      # run all tests (default
↳behavior)
pytest --last-failed --last-failed-no-failures none    # run no tests and exit
```

2.9.4 The new `config.cache` object

Plugins or `conftest.py` support code can get a cached value using the `pytest config` object. Here is a basic example plugin which implements a *fixture* which re-uses previously created state across `pytest` invocations:

```
# content of test_caching.py
import pytest
import time

def expensive_computation():
    print("running expensive computation...")

@pytest.fixture
def mydata(request):
    val = request.config.cache.get("example/value", None)
    if val is None:
        expensive_computation()
        val = 42
        request.config.cache.set("example/value", val)
    return val

def test_function(mydata):
    assert mydata == 23
```

If you run this command for the first time, you can see the print statement:

```
$ pytest -q
F                                                                    [100%]
===== FAILURES =====
_____ test_function _____

mydata = 42

    def test_function(mydata):
>         assert mydata == 23
E         assert 42 == 23

test_caching.py:20: AssertionError
----- Captured stdout setup -----
running expensive computation...
===== short test summary info =====
FAILED test_caching.py::test_function - assert 42 == 23
1 failed in 0.12s
```

If you run it a second time, the value will be retrieved from the cache and nothing will be printed:

```
$ pytest -q
F [100%]
===== FAILURES =====
_____ test_function _____

mydata = 42

    def test_function(mydata):
>     assert mydata == 23
E     assert 42 == 23

test_caching.py:20: AssertionError
===== short test summary info =====
FAILED test_caching.py::test_function - assert 42 == 23
1 failed in 0.12s
```

See the *config.cache fixture* for more details.

2.9.5 Inspecting Cache content

You can always peek at the content of the cache using the `--cache-show` command line option:

```
$ pytest --cache-show
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
cachedir: /home/sweet/project/.pytest_cache
----- cache values for '*' -----
cache/lastfailed contains:
  {'test_caching.py::test_function': True}
cache/nodeids contains:
  ['test_caching.py::test_function']
cache/stepwise contains:
  []
example/value contains:
  42
===== no tests ran in 0.12s =====
```

`--cache-show` takes an optional argument to specify a glob pattern for filtering:

```
$ pytest --cache-show example/*
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
cachedir: /home/sweet/project/.pytest_cache
----- cache values for 'example/*' -----
example/value contains:
  42
===== no tests ran in 0.12s =====
```


2.9.6 Clearing Cache content

You can instruct pytest to clear all cache files and values by adding the `--cache-clear` option like this:

```
pytest --cache-clear
```

This is recommended for invocations from Continuous Integration servers where isolation and correctness is more important than speed.

2.9.7 Stepwise

As an alternative to `--lf -x`, especially for cases where you expect a large part of the test suite will fail, `--sw`, `--stepwise` allows you to fix them one at a time. The test suite will run until the first failure and then stop. At the next invocation, tests will continue from the last failing test and then run until the next failing test. You may use the `--stepwise-skip` option to ignore one failing test and stop the test execution on the second failing test instead. This is useful if you get stuck on a failing test and just want to ignore it until later. Providing `--stepwise-skip` will also enable `--stepwise` implicitly.

2.10 How to manage logging

pytest captures log messages of level `WARNING` or above automatically and displays them in their own section for each failed test in the same manner as captured stdout and stderr.

Running without options:

```
pytest
```

Shows failed tests like so:

```
----- Captured stdout call -----
test_reporting.py 26 WARNING text going to logger
----- Captured stdout call -----
text going to stdout
----- Captured stderr call -----
text going to stderr
===== 2 failed in 0.02 seconds =====
```

By default each captured log message shows the module, line number, log level and message.

If desired the log and date format can be specified to anything that the logging module supports by passing specific formatting options:

```
pytest --log-format="% (asctime)s % (levelname)s % (message)s" \
       --log-date-format="%Y-%m-%d %H:%M:%S"
```

Shows failed tests like so:

```
----- Captured stdout call -----
2010-04-10 14:48:44 WARNING text going to logger
----- Captured stdout call -----
text going to stdout
----- Captured stderr call -----
text going to stderr
===== 2 failed in 0.02 seconds =====
```

These options can also be customized through `pytest.ini` file:

```
[pytest]
log_format = %(asctime)s %(levelname)s %(message)s
log_date_format = %Y-%m-%d %H:%M:%S
```

Further it is possible to disable reporting of captured content (stdout, stderr and logs) on failed tests completely with:

```
pytest --show-capture=no
```

2.10.1 caplog fixture

Inside tests it is possible to change the log level for the captured log messages. This is supported by the `caplog` fixture:

```
def test_foo(caplog):
    caplog.set_level(logging.INFO)
    pass
```

By default the level is set on the root logger, however as a convenience it is also possible to set the log level of any logger:

```
def test_foo(caplog):
    caplog.set_level(logging.CRITICAL, logger="root.baz")
    pass
```

The log levels set are restored automatically at the end of the test.

It is also possible to use a context manager to temporarily change the log level inside a `with` block:

```
def test_bar(caplog):
    with caplog.at_level(logging.INFO):
        pass
```

Again, by default the level of the root logger is affected but the level of any logger can be changed instead with:

```
def test_bar(caplog):
    with caplog.at_level(logging.CRITICAL, logger="root.baz"):
        pass
```

Lastly all the logs sent to the logger during the test run are made available on the fixture in the form of both the `logging.LogRecord` instances and the final log text. This is useful for when you want to assert on the contents of a message:

```
def test_baz(caplog):
    func_under_test()
    for record in caplog.records:
        assert record.levelname != "CRITICAL"
    assert "wally" not in caplog.text
```

For all the available attributes of the log records see the `logging.LogRecord` class.

You can also resort to `record_tuples` if all you want to do is to ensure, that certain messages have been logged under a given logger name with a given severity and message:

```
def test_foo(caplog):
    logging.getLogger().info("boo %s", "arg")

    assert caplog.record_tuples == [("root", logging.INFO, "boo arg")]
```

You can call `caplog.clear()` to reset the captured log records in a test:

```
def test_something_with_clearing_records(caplog):
    some_method_that_creates_log_records()
    caplog.clear()
    your_test_method()
    assert ["Foo"] == [rec.message for rec in caplog.records]
```

The `caplog.records` attribute contains records from the current stage only, so inside the `setup` phase it contains only setup logs, same with the `call` and `teardown` phases.

To access logs from other stages, use the `caplog.get_records(when)` method. As an example, if you want to make sure that tests which use a certain fixture never log any warnings, you can inspect the records for the `setup` and `call` stages during `teardown` like so:

```
@pytest.fixture
def window(caplog):
    window = create_window()
    yield window
    for when in ("setup", "call"):
        messages = [
            x.message for x in caplog.get_records(when) if x.levelno == logging.
↳WARNING
        ]
        if messages:
            pytest.fail(
                "warning messages encountered during testing: {}".format(messages)
            )
```

The full API is available at `pytest.LogCaptureFixture`.

2.10.2 Live Logs

By setting the `log_cli` configuration option to `true`, `pytest` will output logging records as they are emitted directly into the console.

You can specify the logging level for which log records with equal or higher level are printed to the console by passing `--log-cli-level`. This setting accepts the logging level names as seen in python's documentation or an integer as the logging level num.

Additionally, you can also specify `--log-cli-format` and `--log-cli-date-format` which mirror and default to `--log-format` and `--log-date-format` if not provided, but are applied only to the console logging handler.

All of the CLI log options can also be set in the configuration INI file. The option names are:

- `log_cli_level`
- `log_cli_format`
- `log_cli_date_format`

If you need to record the whole test suite logging calls to a file, you can pass `--log-file=/path/to/log/file`. This log file is opened in write mode which means that it will be overwritten at each run tests session.

You can also specify the logging level for the log file by passing `--log-file-level`. This setting accepts the logging level names as seen in python's documentation(ie, uppercased level names) or an integer as the logging level num.

Additionally, you can also specify `--log-file-format` and `--log-file-date-format` which are equal to `--log-format` and `--log-date-format` but are applied to the log file logging handler.

All of the log file options can also be set in the configuration INI file. The option names are:

- `log_file`
- `log_file_level`
- `log_file_format`
- `log_file_date_format`

You can call `set_log_path()` to customize the `log_file` path dynamically. This functionality is considered **experimental**.

2.10.3 Customizing Colors

Log levels are colored if colored terminal output is enabled. Changing from default colors or putting color on custom log levels is supported through `add_color_level()`. Example:

```
@pytest.hookimpl
def pytest_configure(config):
    logging_plugin = config.pluginmanager.get_plugin("logging-plugin")

    # Change color on existing log level
    logging_plugin.log_cli_handler.formatter.add_color_level(logging.INFO, "cyan")

    # Add color to a custom log level (a custom log level `SPAM` is already set up)
    logging_plugin.log_cli_handler.formatter.add_color_level(logging.SPAM, "blue")
```

Warning: This feature and its API are considered **experimental** and might change between releases without a deprecation notice.

2.10.4 Release notes

This feature was introduced as a drop-in replacement for the `pytest-catchlog` plugin and they conflict with each other. The backward compatibility API with `pytest-capturelog` has been dropped when this feature was introduced, so if for that reason you still need `pytest-catchlog` you can disable the internal feature by adding to your `pytest.ini`:

```
[pytest]
addopts=-p no:logging
```

2.10.5 Incompatible changes in pytest 3.4

This feature was introduced in 3.3 and some **incompatible changes** have been made in 3.4 after community feedback:

- Log levels are no longer changed unless explicitly requested by the `log_level` configuration or `--log-level` command-line options. This allows users to configure logger objects themselves. Setting `log_level` will set the level that is captured globally so if a specific test requires a lower level than this, use the `caplog.set_level()` functionality otherwise that test will be prone to failure.
- *Live Logs* is now disabled by default and can be enabled setting the `log_cli` configuration option to `true`. When enabled, the verbosity is increased so logging for each test is visible.
- *Live Logs* are now sent to `sys.stdout` and no longer require the `-s` command-line option to work.

If you want to partially restore the logging behavior of version 3.3, you can add this options to your `ini` file:

```
[pytest]
log_cli=true
log_level=NOTSET
```

More details about the discussion that lead to this changes can be read in issue [#3013](#).

2.11 How to capture stdout/stderr output

2.11.1 Default stdout/stderr/stdin capturing behaviour

During test execution any output sent to `stdout` and `stderr` is captured. If a test or a setup method fails its according captured output will usually be shown along with the failure traceback. (this behavior can be configured by the `--show-capture` command-line option).

In addition, `stdin` is set to a “null” object which will fail on attempts to read from it because it is rarely desired to wait for interactive input when running automated tests.

By default capturing is done by intercepting writes to low level file descriptors. This allows to capture output from simple print statements as well as output from a subprocess started by a test.

2.11.2 Setting capturing methods or disabling capturing

There are three ways in which `pytest` can perform capturing:

- `fd` (file descriptor) level capturing (default): All writes going to the operating system file descriptors 1 and 2 will be captured.
- `sys` level capturing: Only writes to Python files `sys.stdout` and `sys.stderr` will be captured. No capturing of writes to filedescriptors is performed.
- `tee-sys` capturing: Python writes to `sys.stdout` and `sys.stderr` will be captured, however the writes will also be passed-through to the actual `sys.stdout` and `sys.stderr`. This allows output to be ‘live printed’ and captured for plugin use, such as `junitxml` (new in `pytest 5.4`).

You can influence output capturing mechanisms from the command line:

```
pytest -s                # disable all capturing
pytest --capture=sys     # replace sys.stdout/stderr with in-mem files
pytest --capture=fd     # also point filedescriptors 1 and 2 to temp file
pytest --capture=tee-sys # combines 'sys' and '-s', capturing sys.stdout/stderr
                        # and passing it along to the actual sys.stdout/stderr
```

2.11.3 Using print statements for debugging

One primary benefit of the default capturing of `stdout/stderr` output is that you can use print statements for debugging:

```
# content of test_module.py

def setup_function(function):
    print("setting up", function)
```

(continues on next page)

(continued from previous page)

```
def test_func1():
    assert True

def test_func2():
    assert False
```

and running this module will show you precisely the output of the failing function and hide the other one:

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items

test_module.py .F [100%]

===== FAILURES =====
_____ test_func2 _____

    def test_func2():
>     assert False
E     assert False

test_module.py:12: AssertionError
----- Captured stdout setup -----
setting up <function test_func2 at 0xdeadbeef0001>
===== short test summary info =====
FAILED test_module.py::test_func2 - assert False
===== 1 failed, 1 passed in 0.12s =====
```

2.11.4 Accessing captured output from a test function

The `capsys`, `capsysbinary`, `capfd`, and `capfdbinary` fixtures allow access to `stdout`/`stderr` output created during test execution. Here is an example test function that performs some output related checks:

```
def test_myoutput(capsys): # or use "capfd" for fd-level
    print("hello")
    sys.stderr.write("world\n")
    captured = capsys.readouterr()
    assert captured.out == "hello\n"
    assert captured.err == "world\n"
    print("next")
    captured = capsys.readouterr()
    assert captured.out == "next\n"
```

The `readouterr()` call snapshots the output so far - and capturing will be continued. After the test function finishes the original streams will be restored. Using `capsys` this way frees your test from having to care about setting/resetting output streams and also interacts well with pytest's own per-test capturing.

If you want to capture on filedescriptor level you can use the `capfd` fixture which offers the exact same interface but allows to also capture output from libraries or subprocesses that directly write to operating system level output streams (FD1 and FD2).

The return value from `readouterr` changed to a `namedtuple` with two attributes, `out` and `err`.

If the code under test writes non-textual data, you can capture this using the `capsysbinary` fixture which instead returns `bytes` from the `readouterr` method.

If the code under test writes non-textual data, you can capture this using the `capfdbinary` fixture which instead returns `bytes` from the `readouterr` method. The `capfdbinary` fixture operates on the `filedescriptor` level.

To temporarily disable capture within a test, both `capsys` and `capfd` have a `disabled()` method that can be used as a context manager, disabling capture inside the `with` block:

```
def test_disabling_capturing(capsys):
    print("this output is captured")
    with capsys.disabled():
        print("output not captured, going directly to sys.stdout")
    print("this output is also captured")
```

2.12 How to capture warnings

Starting from version 3.1, `pytest` now automatically catches warnings during test execution and displays them at the end of the session:

```
# content of test_show_warnings.py
import warnings

def api_v1():
    warnings.warn(UserWarning("api v1, should use functions from v2"))
    return 1

def test_one():
    assert api_v1() == 1
```

Running `pytest` now produces this output:

```
$ pytest test_show_warnings.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_show_warnings.py . [100%]

===== warnings summary =====
test_show_warnings.py::test_one
  /home/sweet/project/test_show_warnings.py:5: UserWarning: api v1, should use
  ↪functions from v2
    warnings.warn(UserWarning("api v1, should use functions from v2"))

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
===== 1 passed, 1 warning in 0.12s =====
```

The `-W` flag can be passed to control which warnings will be displayed or even turn them into errors:

```

$ pytest -q test_show_warnings.py -W error::UserWarning
F                                                                    [100%]
===== FAILURES =====
_____ test_one _____

    def test_one():
>         assert api_v1() == 1

test_show_warnings.py:10:
-----

    def api_v1():
>         warnings.warn(UserWarning("api v1, should use functions from v2"))
E         UserWarning: api v1, should use functions from v2

test_show_warnings.py:5: UserWarning
===== short test summary info =====
FAILED test_show_warnings.py::test_one - UserWarning: api v1, should use ...
1 failed in 0.12s

```

The same option can be set in the `pytest.ini` or `pyproject.toml` file using the `filterwarnings` ini option. For example, the configuration below will ignore all user warnings and specific deprecation warnings matching a regex, but will transform all other warnings into errors.

```

# pytest.ini
[pytest]
filterwarnings =
    error
    ignore::UserWarning
    ignore:function ham\(\) is deprecated:DeprecationWarning

```

```

# pyproject.toml
[tool.pytest.ini_options]
filterwarnings = [
    "error",
    "ignore::UserWarning",
    # note the use of single quote below to denote "raw" strings in TOML
    'ignore:function ham\(\) is deprecated:DeprecationWarning',
]

```

When a warning matches more than one option in the list, the action for the last matching option is performed.

Both `-W` command-line option and `filterwarnings` ini option are based on Python's own `-W` option and `warnings.simplefilter()`, so please refer to those sections in the Python documentation for other examples and advanced usage.

2.12.1 @pytest.mark.filterwarnings

You can use the `@pytest.mark.filterwarnings` to add warning filters to specific test items, allowing you to have finer control of which warnings should be captured at test, class or even module level:

```
import warnings

def api_v1():
    warnings.warn(UserWarning("api v1, should use functions from v2"))
    return 1

@pytest.mark.filterwarnings("ignore:api v1")
def test_one():
    assert api_v1() == 1
```

Filters applied using a mark take precedence over filters passed on the command line or configured by the `filterwarnings` ini option.

You may apply a filter to all tests of a class by using the `filterwarnings` mark as a class decorator or to all tests in a module by setting the `pytestmark` variable:

```
# turns all warnings into errors for this module
pytestmark = pytest.mark.filterwarnings("error")
```

Credits go to Florian Schulze for the reference implementation in the `pytest-warnings` plugin.

2.12.2 Disabling warnings summary

Although not recommended, you can use the `--disable-warnings` command-line option to suppress the warning summary entirely from the test run output.

2.12.3 Disabling warning capture entirely

This plugin is enabled by default but can be disabled entirely in your `pytest.ini` file with:

```
[pytest]
addopts = -p no:warnings
```

Or passing `-p no:warnings` in the command-line. This might be useful if your test suites handles warnings using an external system.

2.12.4 DeprecationWarning and PendingDeprecationWarning

By default `pytest` will display `DeprecationWarning` and `PendingDeprecationWarning` warnings from user code and third-party libraries, as recommended by [PEP 565](#). This helps users keep their code modern and avoid breakages when deprecated warnings are effectively removed.

Sometimes it is useful to hide some specific deprecation warnings that happen in code that you have no control over (such as third-party libraries), in which case you might use the warning filters options (ini or marks) to ignore those warnings.

For example:

```
[pytest]
filterwarnings =
    ignore::*U.*mode is deprecated:DeprecationWarning
```

This will ignore all warnings of type `DeprecationWarning` where the start of the message matches the regular expression `"*U.*mode is deprecated"`.

Note: If warnings are configured at the interpreter level, using the `PYTHONWARNINGS` environment variable or the `-W` command-line option, pytest will not configure any filters by default.

Also pytest doesn't follow [PEP 506](#) suggestion of resetting all warning filters because it might break test suites that configure warning filters themselves by calling `warnings.simplefilter()` (see [issue #2430](#) for an example of that).

2.12.5 Ensuring code triggers a deprecation warning

You can also use `pytest.deprecated_call()` for checking that a certain function call triggers a `DeprecationWarning` or `PendingDeprecationWarning`:

```
import pytest

def test_myfunction_deprecated():
    with pytest.deprecated_call():
        myfunction(17)
```

This test will fail if `myfunction` does not issue a deprecation warning when called with a `17` argument.

2.12.6 Asserting warnings with the warns function

You can check that code raises a particular warning using `pytest.warns()`, which works in a similar manner to `raises`:

```
import warnings
import pytest

def test_warning():
    with pytest.warns(UserWarning):
        warnings.warn("my warning", UserWarning)
```

The test will fail if the warning in question is not raised. The keyword argument `match` to assert that the exception matches a text or regex:

```
>>> with warns(UserWarning, match='must be 0 or None'):
...     warnings.warn("value must be 0 or None", UserWarning)

>>> with warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("value must be 42", UserWarning)

>>> with warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("this is not here", UserWarning)
```

(continues on next page)

(continued from previous page)

```
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...
```

You can also call `pytest.warns()` on a function or code string:

```
pytest.warns(expected_warning, func, *args, **kwargs)
pytest.warns(expected_warning, "func(*args, **kwargs)")
```

The function also returns a list of all raised warnings (as `warnings.WarningMessage` objects), which you can query for additional information:

```
with pytest.warns(RuntimeWarning) as record:
    warnings.warn("another warning", RuntimeWarning)

# check that only one warning was raised
assert len(record) == 1
# check that the message matches
assert record[0].message.args[0] == "another warning"
```

Alternatively, you can examine raised warnings in detail using the `recwarn` fixture (see below).

The `recwarn` fixture automatically ensures to reset the warnings filter at the end of the test, so no global state is leaked.

2.12.7 Recording warnings

You can record raised warnings either using `pytest.warns()` or with the `recwarn` fixture.

To record with `pytest.warns()` without asserting anything about the warnings, pass no arguments as the expected warning type and it will default to a generic `Warning`:

```
with pytest.warns() as record:
    warnings.warn("user", UserWarning)
    warnings.warn("runtime", RuntimeWarning)

assert len(record) == 2
assert str(record[0].message) == "user"
assert str(record[1].message) == "runtime"
```

The `recwarn` fixture will record warnings for the whole function:

```
import warnings

def test_hello(recwarn):
    warnings.warn("hello", UserWarning)
    assert len(recwarn) == 1
    w = recwarn.pop(UserWarning)
    assert isinstance(w.category, UserWarning)
    assert str(w.message) == "hello"
    assert w.filename
    assert w.lineno
```

Both `recwarn` and `pytest.warns()` return the same interface for recorded warnings: a `WarningsRecorder` instance. To view the recorded warnings, you can iterate over this instance, call `len` on it to get the number of recorded warnings, or index into it to get a particular recorded warning.

Full API: *WarningsRecorder*.

2.12.8 Custom failure messages

Recording warnings provides an opportunity to produce custom test failure messages for when no warnings are issued or other conditions are met.

```
def test():
    with pytest.warns(Warning) as record:
        f()
        if not record:
            pytest.fail("Expected a warning!")
```

If no warnings are issued when calling `f`, then `not record` will evaluate to `True`. You can then call `pytest.fail()` with a custom error message.

2.12.9 Internal pytest warnings

pytest may generate its own warnings in some situations, such as improper usage or deprecated features.

For example, pytest will emit a warning if it encounters a class that matches *python_classes* but also defines an `__init__` constructor, as this prevents the class from being instantiated:

```
# content of test_pytest_warnings.py
class Test:
    def __init__(self):
        pass

    def test_foo(self):
        assert 1 == 1
```

```
$ pytest test_pytest_warnings.py -q

===== warnings summary =====
test_pytest_warnings.py:1
  /home/sweet/project/test_pytest_warnings.py:1: PytestCollectionWarning: cannot_
↪ collect test class 'Test' because it has a __init__ constructor (from: test_pytest_
↪ warnings.py)
    class Test:

-- Docs: https://docs.pytest.org/en/stable/how-to/capture-warnings.html
1 warning in 0.12s
```

These warnings might be filtered using the same builtin mechanisms used to filter other types of warnings.

Please read our *Backwards Compatibility Policy* to learn how we proceed about deprecating and eventually removing features.

The full list of warnings is listed in *the reference documentation*.

2.13 How to use skip and xfail to deal with tests that cannot succeed

You can mark test functions that cannot be run on certain platforms or that you expect to fail so pytest can deal with them accordingly and present a summary of the test session, while keeping the test suite *green*.

A **skip** means that you expect your test to pass only if some conditions are met, otherwise pytest should skip running the test altogether. Common examples are skipping windows-only tests on non-windows platforms, or skipping tests that depend on an external resource which is not available at the moment (for example a database).

An **xfail** means that you expect a test to fail for some reason. A common example is a test for a feature not yet implemented, or a bug not yet fixed. When a test passes despite being expected to fail (marked with `pytest.mark.xfail`), it's an **xpass** and will be reported in the test summary.

pytest counts and lists *skip* and *xfail* tests separately. Detailed information about skipped/xfailed tests is not shown by default to avoid cluttering the output. You can use the `-r` option to see details corresponding to the “short” letters shown in the test progress:

```
pytest -rxXs # show extra info on xfailed, xpassed, and skipped tests
```

More details on the `-r` option can be found by running `pytest -h`.

(See *Builtin configuration file options*)

2.13.1 Skipping test functions

The simplest way to skip a test function is to mark it with the `skip` decorator which may be passed an optional `reason`:

```
@pytest.mark.skip(reason="no way of currently testing this")
def test_the_unknown():
    ...
```

Alternatively, it is also possible to skip imperatively during test execution or setup by calling the `pytest.skip(reason)` function:

```
def test_function():
    if not valid_config():
        pytest.skip("unsupported configuration")
```

The imperative method is useful when it is not possible to evaluate the skip condition during import time.

It is also possible to skip the whole module using `pytest.skip(reason, allow_module_level=True)` at the module level:

```
import sys
import pytest

if not sys.platform.startswith("win"):
    pytest.skip("skipping windows-only tests", allow_module_level=True)
```

Reference: `pytest.mark.skip`

skipif

If you wish to skip something conditionally then you can use `skipif` instead. Here is an example of marking a test function to be skipped when run on an interpreter earlier than Python3.6:

```
import sys

@pytest.mark.skipif(sys.version_info < (3, 7), reason="requires python3.7 or higher")
def test_function():
    ...
```

If the condition evaluates to `True` during collection, the test function will be skipped, with the specified reason appearing in the summary when using `-rs`.

You can share `skipif` markers between modules. Consider this test module:

```
# content of test_mymodule.py
import mymodule

minversion = pytest.mark.skipif(
    mymodule.__versioninfo__ < (1, 1), reason="at least mymodule-1.1 required"
)

@minversion
def test_function():
    ...
```

You can import the marker and reuse it in another test module:

```
# test_myothermodule.py
from test_mymodule import minversion

@minversion
def test_anotherfunction():
    ...
```

For larger test suites it's usually a good idea to have one file where you define the markers which you then consistently apply throughout your test suite.

Alternatively, you can use *condition strings* instead of booleans, but they can't be shared between modules easily so they are supported mainly for backward compatibility reasons.

Reference: `pytest.mark.skipif`

Skip all test functions of a class or module

You can use the `skipif` marker (as any other marker) on classes:

```
@pytest.mark.skipif(sys.platform == "win32", reason="does not run on windows")
class TestPosixCalls:
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

If the condition is `True`, this marker will produce a skip result for each of the test methods of that class.

If you want to skip all test functions of a module, you may use the `pytestmark` global:

```
# test_module.py
pytestmark = pytest.mark.skipif(...)
```

If multiple `skipif` decorators are applied to a test function, it will be skipped if any of the skip conditions is true.

Skipping files or directories

Sometimes you may need to skip an entire file or directory, for example if the tests rely on Python version-specific features or contain code that you do not wish pytest to run. In this case, you must exclude the files and directories from collection. Refer to [Customizing test collection](#) for more information.

Skipping on a missing import dependency

You can skip tests on a missing import by using `pytest.importorskip` at module level, within a test, or test setup function.

```
docutils = pytest.importorskip("docutils")
```

If `docutils` cannot be imported here, this will lead to a skip outcome of the test. You can also skip based on the version number of a library:

```
docutils = pytest.importorskip("docutils", minversion="0.3")
```

The version will be read from the specified module's `__version__` attribute.

Summary

Here's a quick guide on how to skip tests in a module in different situations:

1. Skip all tests in a module unconditionally:

```
pytestmark = pytest.mark.skip("all tests still WIP")
```

2. Skip all tests in a module based on some condition:

```
pytestmark = pytest.mark.skipif(sys.platform == "win32", reason="tests for_
↳linux only")
```

3. Skip all tests in a module if some import is missing:

```
pexpect = pytest.importorskip("pexpect")
```

2.13.2 XFail: mark test functions as expected to fail

You can use the `xfail` marker to indicate that you expect a test to fail:

```
@pytest.mark.xfail
def test_function():
    ...
```

This test will run but no traceback will be reported when it fails. Instead, terminal reporting will list it in the “expected to fail” (XFAIL) or “unexpectedly passing” (XPASS) sections.

Alternatively, you can also mark a test as XFAIL from within the test or its setup function imperatively:

```
def test_function():
    if not valid_config():
        pytest.xfail("failing configuration (but should work)")
```

```
def test_function2():
    import slow_module

    if slow_module.slow_function():
        pytest.xfail("slow_module taking too long")
```

These two examples illustrate situations where you don’t want to check for a condition at the module level, which is when a condition would otherwise be evaluated for marks.

This will make `test_function` XFAIL. Note that no other code is executed after the `pytest.xfail()` call, differently from the marker. That’s because it is implemented internally by raising a known exception.

Reference: [*pytest.mark.xfail*](#)

condition parameter

If a test is only expected to fail under a certain condition, you can pass that condition as the first parameter:

```
@pytest.mark.xfail(sys.platform == "win32", reason="bug in a 3rd party library")
def test_function():
    ...
```

Note that you have to pass a reason as well (see the parameter description at [*pytest.mark.xfail*](#)).

reason parameter

You can specify the motive of an expected failure with the `reason` parameter:

```
@pytest.mark.xfail(reason="known parser issue")
def test_function():
    ...
```

raises parameter

If you want to be more specific as to why the test is failing, you can specify a single exception, or a tuple of exceptions, in the `raises` argument.

```
@pytest.mark.xfail(raises=RuntimeError)
def test_function():
    ...
```

Then the test will be reported as a regular failure if it fails with an exception not mentioned in `raises`.

run parameter

If a test should be marked as `xfail` and reported as such but should not be even executed, use the `run` parameter as `False`:

```
@pytest.mark.xfail(run=False)
def test_function():
    ...
```

This is specially useful for xfailing tests that are crashing the interpreter and should be investigated later.

strict parameter

Both `XFAIL` and `XPASS` don't fail the test suite by default. You can change this by setting the `strict` keyword-only parameter to `True`:

```
@pytest.mark.xfail(strict=True)
def test_function():
    ...
```

This will make `XPASS` (“unexpectedly passing”) results from this test to fail the test suite.

You can change the default value of the `strict` parameter using the `xfail_strict` ini option:

```
[pytest]
xfail_strict=true
```

Ignoring xfail

By specifying on the commandline:

```
pytest --runxfail
```

you can force the running and reporting of an `xfail` marked test as if it weren't marked at all. This also causes `pytest.xfail()` to produce no effect.

Examples

Here is a simple test file with the several usages:

```
import pytest

xfail = pytest.mark.xfail

@xfail
def test_hello():
    assert 0

@xfail(run=False)
def test_hello2():
    assert 0
```

(continues on next page)

(continued from previous page)

```
@xfail("hasattr(os, 'sep')")
def test_hello3():
    assert 0

@xfail(reason="bug 110")
def test_hello4():
    assert 0

@xfail('pytest.__version__[0] != "17"')
def test_hello5():
    assert 0

def test_hello6():
    pytest.xfail("reason")

@xfail(raises=IndexError)
def test_hello7():
    x = []
    x[1] = 1
```

Running it with the report-on-xfail option gives this output:

```
! pytest -rx xfail_demo.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/example
collected 7 items

xfail_demo.py xxxxxxxx [100%]

===== short test summary info =====
XFAIL xfail_demo.py::test_hello
XFAIL xfail_demo.py::test_hello2
      reason: [NOTRUN]
XFAIL xfail_demo.py::test_hello3
      condition: hasattr(os, 'sep')
XFAIL xfail_demo.py::test_hello4
      bug 110
XFAIL xfail_demo.py::test_hello5
      condition: pytest.__version__[0] != "17"
XFAIL xfail_demo.py::test_hello6
      reason: reason
XFAIL xfail_demo.py::test_hello7
===== 7 xfailed in 0.12s =====
```

2.13.3 Skip/xfail with parametrize

It is possible to apply markers like skip and xfail to individual test instances when using parametrize:

```
import sys
import pytest

@pytest.mark.parametrize(
    ("n", "expected"),
    [
        (1, 2),
        pytest.param(1, 0, marks=pytest.mark.xfail),
        pytest.param(1, 3, marks=pytest.mark.xfail(reason="some bug")),
        (2, 3),
        (3, 4),
        (4, 5),
        pytest.param(
            10, 11, marks=pytest.mark.skipif(sys.version_info >= (3, 0), reason="py2k
↪")
        ),
    ],
)
def test_increment(n, expected):
    assert n + 1 == expected
```

2.14 How to install and use plugins

This section talks about installing and using third party plugins. For writing your own plugins, please refer to *Writing plugins*.

Installing a third party plugin can be easily done with pip:

```
pip install pytest-NAME
pip uninstall pytest-NAME
```

If a plugin is installed, `pytest` automatically finds and integrates it, there is no need to activate it.

Here is a little annotated list for some popular plugins:

- `pytest-django`: write tests for `django` apps, using `pytest` integration.
- `pytest-twisted`: write tests for `twisted` apps, starting a reactor and processing deferreds from test functions.
- `pytest-cov`: coverage reporting, compatible with distributed testing
- `pytest-xdist`: to distribute tests to CPUs and remote hosts, to run in boxed mode which allows to survive segmentation faults, to run in looponfailing mode, automatically re-running failing tests on file changes.
- `pytest-instafail`: to report failures while the test run is happening.
- `pytest-bdd`: to write tests using behaviour-driven testing.
- `pytest-timeout`: to timeout tests based on function marks or global definitions.
- `pytest-pep8`: a `--pep8` option to enable PEP8 compliance checking.
- `pytest-flakes`: check source code with `pyflakes`.
- `oejskit`: a plugin to run javascript unittests in live browsers.

To see a complete list of all plugins with their latest testing status against different pytest and Python versions, please visit [Plugin List](#).

You may also discover more plugins through a [pytest- pypi.org search](#).

2.14.1 Requiring/Loading plugins in a test module or conftest file

You can require plugins in a test module or a conftest file using `pytest_plugins`:

```
pytest_plugins = ("myapp.testsupport.myplugin",)
```

When the test module or conftest plugin is loaded the specified plugins will be loaded as well.

Note: Requiring plugins using a `pytest_plugins` variable in non-root `conftest.py` files is deprecated. See [full explanation](#) in the Writing plugins section.

Note: The name `pytest_plugins` is reserved and should not be used as a name for a custom plugin module.

2.14.2 Finding out which plugins are active

If you want to find out which plugins are active in your environment you can type:

```
pytest --trace-config
```

and will get an extended test header which shows activated plugins and their names. It will also print local plugins aka `conftest.py` files when they are loaded.

2.14.3 Deactivating / unregistering a plugin by name

You can prevent plugins from loading or unregister them:

```
pytest -p no:NAME
```

This means that any subsequent try to activate/load the named plugin will not work.

If you want to unconditionally disable a plugin for a project, you can add this option to your `pytest.ini` file:

```
[pytest]
addopts = -p no:NAME
```

Alternatively to disable it only in certain environments (for example in a CI server), you can set `PYTEST_ADDOPTS` environment variable to `-p no:name`.

See [Finding out which plugins are active](#) for how to obtain the name of a plugin.

2.15 Writing plugins

It is easy to implement *local conftest plugins* for your own project or *pip-installable plugins* that can be used throughout many projects, including third party projects. Please refer to *How to install and use plugins* if you only want to use but not write plugins.

A plugin contains one or multiple hook functions. *Writing hooks* explains the basics and details of how you can write a hook function yourself. `pytest` implements all aspects of configuration, collection, running and reporting by calling *well specified hooks* of the following plugins:

- *builtin plugins*: loaded from `pytest`'s internal `_pytest` directory.
- *external plugins*: modules discovered through *setuptools entry points*
- *conftest.py plugins*: modules auto-discovered in test directories

In principle, each hook call is a `1 : N` Python function call where `N` is the number of registered implementation functions for a given specification. All specifications and implementations follow the `pytest_` prefix naming convention, making them easy to distinguish and find.

2.15.1 Plugin discovery order at tool startup

`pytest` loads plugin modules at tool startup in the following way:

1. by scanning the command line for the `-p no:name` option and *blocking* that plugin from being loaded (even builtin plugins can be blocked this way). This happens before normal command-line parsing.
2. by loading all builtin plugins.
3. by scanning the command line for the `-p name` option and loading the specified plugin. This happens before normal command-line parsing.
4. by loading all plugins registered through *setuptools entry points*.
5. by loading all plugins specified through the `PYTEST_PLUGINS` environment variable.
6. by loading all `conftest.py` files as inferred by the command line invocation:
 - if no test paths are specified, use the current dir as a test path
 - if exists, load `conftest.py` and `test*/conftest.py` relative to the directory part of the first test path. After the `conftest.py` file is loaded, load all plugins specified in its `pytest_plugins` variable if present.

Note that `pytest` does not find `conftest.py` files in deeper nested sub directories at tool startup. It is usually a good idea to keep your `conftest.py` file in the top level test or project root directory.

7. by recursively loading all plugins specified by the `pytest_plugins` variable in `conftest.py` files.

2.15.2 conftest.py: local per-directory plugins

Local `conftest.py` plugins contain directory-specific hook implementations. Hook Session and test running activities will invoke all hooks defined in `conftest.py` files closer to the root of the filesystem. Example of implementing the `pytest_runtest_setup` hook so that is called for tests in the `a` sub directory but not for other directories:

```
a/conftest.py:
def pytest_runtest_setup(item):
    # called for running each test in 'a' directory
    print("setting up", item)
```

(continues on next page)

(continued from previous page)

```
a/test_sub.py:
    def test_sub():
        pass

test_flat.py:
    def test_flat():
        pass
```

Here is how you might run it:

```
pytest test_flat.py --capture=no # will not show "setting up"
pytest a/test_sub.py --capture=no # will show "setting up"
```

Note: If you have `conftest.py` files which do not reside in a python package directory (i.e. one containing an `__init__.py`) then “import `conftest`” can be ambiguous because there might be other `conftest.py` files as well on your `PYTHONPATH` or `sys.path`. It is thus good practice for projects to either put `conftest.py` under a package scope or to never import anything from a `conftest.py` file.

See also: [pytest import mechanisms and sys.path/PYTHONPATH](#).

Note: Some hooks should be implemented only in plugins or `conftest.py` files situated at the tests root directory due to how pytest discovers plugins during startup, see the documentation of each hook for details.

2.15.3 Writing your own plugin

If you want to write a plugin, there are many real-life examples you can copy from:

- a custom collection example plugin: [A basic example for specifying tests in Yaml files](#)
- builtin plugins which provide pytest’s own functionality
- many *external plugins* providing additional features

All of these plugins implement *hooks* and/or *fixtures* to extend and add functionality.

Note: Make sure to check out the excellent [cookiecutter-pytest-plugin](#) project, which is a [cookiecutter template](#) for authoring plugins.

The template provides an excellent starting point with a working plugin, tests running with `tox`, a comprehensive `README` file as well as a pre-configured entry-point.

Also consider [contributing your plugin to pytest-dev](#) once it has some happy users other than yourself.

2.15.4 Making your plugin installable by others

If you want to make your plugin externally available, you may define a so-called entry point for your distribution so that `pytest` finds your plugin module. Entry points are a feature that is provided by `setuptools`. `pytest` looks up the `pytest11` entrypoint to discover its plugins and you can thus make your plugin available by defining it in your `setuptools`-invocation:

```
# sample ./setup.py file
from setuptools import setup

setup(
    name="myproject",
    packages=["myproject"],
    # the following makes a plugin available to pytest
    entry_points={"pytest11": ["name_of_plugin = myproject.pluginmodule"]},
    # custom PyPI classifier for pytest plugins
    classifiers=["Framework :: Pytest"],
)
```

If a package is installed this way, `pytest` will load `myproject.pluginmodule` as a plugin which can define *hooks*.

Note: Make sure to include `Framework :: Pytest` in your list of `PyPI classifiers` to make it easy for users to find your plugin.

2.15.5 Assertion Rewriting

One of the main features of `pytest` is the use of plain `assert` statements and the detailed introspection of expressions upon assertion failures. This is provided by “assertion rewriting” which modifies the parsed AST before it gets compiled to bytecode. This is done via a [PEP 302](#) import hook which gets installed early on when `pytest` starts up and will perform this rewriting when modules get imported. However, since we do not want to test different bytecode from what you will run in production, this hook only rewrites test modules themselves (as defined by the `python_files` configuration option), and any modules which are part of plugins. Any other imported module will not be rewritten and normal assertion behaviour will happen.

If you have assertion helpers in other modules where you would need assertion rewriting to be enabled you need to ask `pytest` explicitly to rewrite this module before it gets imported.

register_assert_rewrite (*names)

Register one or more module names to be rewritten on import.

This function will make sure that this module or all modules inside the package will get their `assert` statements rewritten. Thus you should make sure to call this before the module is actually imported, usually in your `__init__.py` if you are a plugin using a package.

Raises `TypeError` – If the given module names are not strings.

Parameters `names` (*str*) –

Return type `None`

This is especially important when you write a `pytest` plugin which is created using a package. The import hook only treats `conftest.py` files and any modules which are listed in the `pytest11` entrypoint as plugins. As an example consider the following package:

```
pytest_foo/__init__.py
pytest_foo/plugin.py
pytest_foo/helper.py
```

With the following typical `setup.py` extract:

```
setup(..., entry_points={"pytest11": ["foo = pytest_foo.plugin"]}, ...)
```

In this case only `pytest_foo/plugin.py` will be rewritten. If the helper module also contains assert statements which need to be rewritten it needs to be marked as such, before it gets imported. This is easiest by marking it for rewriting inside the `__init__.py` module, which will always be imported first when a module inside a package is imported. This way `plugin.py` can still import `helper.py` normally. The contents of `pytest_foo/__init__.py` will then need to look like this:

```
import pytest

pytest.register_assert_rewrite("pytest_foo.helper")
```

2.15.6 Requiring/Loading plugins in a test module or conftest file

You can require plugins in a test module or a `conftest.py` file using `pytest_plugins`:

```
pytest_plugins = ["name1", "name2"]
```

When the test module or conftest plugin is loaded the specified plugins will be loaded as well. Any module can be blessed as a plugin, including internal application modules:

```
pytest_plugins = "myapp.testsupport.myplugin"
```

`pytest_plugins` are processed recursively, so note that in the example above if `myapp.testsupport.myplugin` also declares `pytest_plugins`, the contents of the variable will also be loaded as plugins, and so on.

Note: Requiring plugins using `pytest_plugins` variable in non-root `conftest.py` files is deprecated.

This is important because `conftest.py` files implement per-directory hook implementations, but once a plugin is imported, it will affect the entire directory tree. In order to avoid confusion, defining `pytest_plugins` in any `conftest.py` file which is not located in the tests root directory is deprecated, and will raise a warning.

This mechanism makes it easy to share fixtures within applications or even external applications without the need to create external plugins using the `setuptools`'s entry point technique.

Plugins imported by `pytest_plugins` will also automatically be marked for assertion rewriting (see `pytest.register_assert_rewrite()`). However for this to have any effect the module must not be imported already; if it was already imported at the time the `pytest_plugins` statement is processed, a warning will result and assertions inside the plugin will not be rewritten. To fix this you can either call `pytest.register_assert_rewrite()` yourself before the module is imported, or you can arrange the code to delay the importing until after the plugin is registered.

2.15.7 Accessing another plugin by name

If a plugin wants to collaborate with code from another plugin it can obtain a reference through the plugin manager like this:

```
plugin = config.pluginmanager.get_plugin("name_of_plugin")
```

If you want to look at the names of existing plugins, use the `--trace-config` option.

2.15.8 Registering custom markers

If your plugin uses any markers, you should register them so that they appear in pytest's help text and do not *cause spurious warnings*. For example, the following plugin would register `cool_marker` and `mark_with` for all users:

```
def pytest_configure(config):
    config.addinvalue_line("markers", "cool_marker: this one is for cool tests.")
    config.addinvalue_line(
        "markers", "mark_with(arg, arg2): this marker takes arguments."
    )
```

2.15.9 Testing plugins

pytest comes with a plugin named `pytester` that helps you write tests for your plugin code. The plugin is disabled by default, so you will have to enable it before you can use it.

You can do so by adding the following line to a `conftest.py` file in your testing directory:

```
# content of conftest.py

pytest_plugins = ["pytester"]
```

Alternatively you can invoke pytest with the `-p pytester` command line option.

This will allow you to use the `pytester` fixture for testing your plugin code.

Let's demonstrate what you can do with the plugin with an example. Imagine we developed a plugin that provides a fixture `hello` which yields a function and we can invoke this function with one optional parameter. It will return a string value of `Hello World!` if we do not supply a value or `Hello {value}!` if we do supply a string value.

```
import pytest

def pytest_addoption(parser):
    group = parser.getgroup("helloworld")
    group.addoption(
        "--name",
        action="store",
        dest="name",
        default="World",
        help='Default "name" for hello().',
    )

@pytest.fixture
def hello(request):
```

(continues on next page)

(continued from previous page)

```

name = request.config.getoption("name")

def _hello(name=None):
    if not name:
        name = request.config.getoption("name")
    return "Hello {name}!".format(name=name)

return _hello

```

Now the `pytester` fixture provides a convenient API for creating temporary `conftest.py` files and test files. It also allows us to run the tests and return a result object, with which we can assert the tests' outcomes.

```

def test_hello(pytester):
    """Make sure that our plugin works."""

    # create a temporary conftest.py file
    pytester.makeconftest(
        """
import pytest

@pytest.fixture(params=[
    "Brianna",
    "Andreas",
    "Floris",
])
def name(request):
    return request.param
        """
    )

    # create a temporary pytest test file
    pytester.makepyfile(
        """
def test_hello_default(hello):
    assert hello() == "Hello World!"

def test_hello_name(hello, name):
    assert hello(name) == "Hello {0}!".format(name)
        """
    )

    # run all tests with pytest
    result = pytester.runpytest()

    # check that all 4 tests passed
    result.assert_outcomes(passed=4)

```

Additionally it is possible to copy examples to the `pytester`'s isolated environment before running `pytest` on it. This way we can abstract the tested logic to separate files, which is especially useful for longer tests and/or longer `conftest.py` files.

Note that for `pytester.copy_example` to work we need to set `pytester_example_dir` in our `pytest.ini` to tell `pytest` where to look for example files.

```

# content of pytest.ini
[pytest]
pytester_example_dir = .

```

```
# content of test_example.py

def test_plugin(pytester):
    pytester.copy_example("test_example.py")
    pytester.runpytest("-k", "test_example")

def test_example():
    pass
```

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project, configfile: pytest.ini
collected 2 items

test_example.py .. [100%]

===== 2 passed in 0.12s =====
```

For more information about the result object that `runpytest()` returns, and the methods that it provides please check out the *RunResult* documentation.

2.16 Writing hook functions

2.16.1 hook function validation and execution

pytest calls hook functions from registered plugins for any given hook specification. Let’s look at a typical hook function for the `pytest_collection_modifyitems(session, config, items)` hook which pytest calls after collection of all test items is completed.

When we implement a `pytest_collection_modifyitems` function in our plugin pytest will during registration verify that you use argument names which match the specification and bail out if not.

Let’s look at a possible implementation:

```
def pytest_collection_modifyitems(config, items):
    # called after collection is completed
    # you can modify the `items` list
    ...
```

Here, pytest will pass in `config` (the pytest config object) and `items` (the list of collected test items) but will not pass in the `session` argument because we didn’t list it in the function signature. This dynamic “pruning” of arguments allows pytest to be “future-compatible”: we can introduce new hook named parameters without breaking the signatures of existing hook implementations. It is one of the reasons for the general long-lived compatibility of pytest plugins.

Note that hook functions other than `pytest_runttest_*` are not allowed to raise exceptions. Doing so will break the pytest run.

2.16.2 firstresult: stop at first non-None result

Most calls to `pytest` hooks result in a **list of results** which contains all non-None results of the called hook functions.

Some hook specifications use the `firstresult=True` option so that the hook call only executes until the first of `N` registered functions returns a non-None result which is then taken as result of the overall hook call. The remaining hook functions will not be called in this case.

2.16.3 hookwrapper: executing around other hooks

pytest plugins can implement hook wrappers which wrap the execution of other hook implementations. A hook wrapper is a generator function which yields exactly once. When pytest invokes hooks it first executes hook wrappers and passes the same arguments as to the regular hooks.

At the yield point of the hook wrapper pytest will execute the next hook implementations and return their result to the yield point in the form of a `Result` instance which encapsulates a result or exception info. The yield point itself will thus typically not raise exceptions (unless there are bugs).

Here is an example definition of a hook wrapper:

```
import pytest

@pytest.hookimpl(hookwrapper=True)
def pytest_pyfunc_call(pyfuncitem):
    do_something_before_next_hook_executes()

    outcome = yield
    # outcome.excinfo may be None or a (cls, val, tb) tuple

    res = outcome.get_result() # will raise if outcome was exception

    post_process_result(res)

    outcome.force_result(new_res) # to override the return value to the plugin system
```

Note that hook wrappers don't return results themselves, they merely perform tracing or other side effects around the actual hook implementations. If the result of the underlying hook is a mutable object, they may modify that result but it's probably better to avoid it.

For more information, consult the [pluggy documentation](#) about hookwrappers.

2.16.4 Hook function ordering / call example

For any given hook specification there may be more than one implementation and we thus generally view hook execution as a `1:N` function call where `N` is the number of registered functions. There are ways to influence if a hook implementation comes before or after others, i.e. the position in the `N`-sized list of functions:

```
# Plugin 1
@pytest.hookimpl(tryfirst=True)
def pytest_collection_modifyitems(items):
    # will execute as early as possible
    ...
```

(continues on next page)

(continued from previous page)

```
# Plugin 2
@pytest.hookimpl(trylast=True)
def pytest_collection_modifyitems(items):
    # will execute as late as possible
    ...

# Plugin 3
@pytest.hookimpl(hookwrapper=True)
def pytest_collection_modifyitems(items):
    # will execute even before the tryfirst one above!
    outcome = yield
    # will execute after all non-hookwrappers executed
```

Here is the order of execution:

1. Plugin3's `pytest_collection_modifyitems` called until the yield point because it is a hook wrapper.
2. Plugin1's `pytest_collection_modifyitems` is called because it is marked with `tryfirst=True`.
3. Plugin2's `pytest_collection_modifyitems` is called because it is marked with `trylast=True` (but even without this mark it would come after Plugin1).
4. Plugin3's `pytest_collection_modifyitems` then executing the code after the yield point. The yield receives a `Result` instance which encapsulates the result from calling the non-wrappers. Wrappers shall not modify the result.

It's possible to use `tryfirst` and `trylast` also in conjunction with `hookwrapper=True` in which case it will influence the ordering of hookwrappers among each other.

2.16.5 Declaring new hooks

Note: This is a quick overview on how to add new hooks and how they work in general, but a more complete overview can be found in [the pluggy documentation](#).

Plugins and `conftest.py` files may declare new hooks that can then be implemented by other plugins in order to alter behaviour or interact with the new plugin:

pytest_addhooks (*pluginmanager*)

Called at plugin registration time to allow adding new hooks via a call to `pluginmanager.add_hookspecs(module_or_class, prefix)`.

Parameters `pluginmanager` (`pytest.PytestPluginManager`) – The pytest plugin manager.

Return type `None`

Note: This hook is incompatible with `hookwrapper=True`.

Hooks are usually declared as do-nothing functions that contain only documentation describing when the hook will be called and what return values are expected. The names of the functions must start with `pytest_` otherwise pytest won't recognize them.

Here's an example. Let's assume this code is in the `sample_hook.py` module.

```
def pytest_my_hook(config):  
    """  
    Receives the pytest config and does things with it  
    """
```

To register the hooks with pytest they need to be structured in their own module or class. This class or module can then be passed to the `pluginmanager` using the `pytest_addhooks` function (which itself is a hook exposed by pytest).

```
def pytest_addhooks(pluginmanager):  
    """ This example assumes the hooks are grouped in the 'sample_hook' module. """  
    from my_app.tests import sample_hook  
  
    pluginmanager.add_hookspecs(sample_hook)
```

For a real world example, see `newhooks.py` from `xdist`.

Hooks may be called both from fixtures or from other hooks. In both cases, hooks are called through the `hook` object, available in the `config` object. Most hooks receive a `config` object directly, while fixtures may use the `pytest-config` fixture which provides the same object.

```
@pytest.fixture()  
def my_fixture(pytestconfig):  
    # call the hook called "pytest_my_hook"  
    # 'result' will be a list of return values from all registered functions.  
    result = pytestconfig.hook.pytest_my_hook(config=pytestconfig)
```

Note: Hooks receive parameters using only keyword arguments.

Now your hook is ready to be used. To register a function at the hook, other plugins or users must now simply define the function `pytest_my_hook` with the correct signature in their `confstest.py`.

Example:

```
def pytest_my_hook(config):  
    """  
    Print all active hooks to the screen.  
    """  
    print(config.hook)
```

2.16.6 Using hooks in `pytest_addoption`

Occasionally, it is necessary to change the way in which command line options are defined by one plugin based on hooks in another plugin. For example, a plugin may expose a command line option for which another plugin needs to define the default value. The `pluginmanager` can be used to install and use hooks to accomplish this. The plugin would define and add the hooks and use `pytest_addoption` as follows:

```
# contents of hooks.py  
  
# Use firstresult=True because we only want one plugin to define this  
# default value  
@hookspec(firstresult=True)  
def pytest_config_file_default_value():  
    """ Return the default value for the config file command line option. """
```

(continues on next page)

(continued from previous page)

```
# contents of myplugin.py

def pytest_addhooks(pluginmanager):
    """ This example assumes the hooks are grouped in the 'hooks' module. """
    from . import hooks

    pluginmanager.add_hookspecs(hooks)

def pytest_addoption(parser, pluginmanager):
    default_value = pluginmanager.hook.pytest_config_file_default_value()
    parser.addoption(
        "--config-file",
        help="Config file to use, defaults to %(default)s",
        default=default_value,
    )
```

The `conftest.py` that is using `myplugin` would simply define the hook as follows:

```
def pytest_config_file_default_value():
    return "config.yaml"
```

2.16.7 Optionally using hooks from 3rd party plugins

Using new hooks from plugins as explained above might be a little tricky because of the standard *validation mechanism*: if you depend on a plugin that is not installed, validation will fail and the error message will not make much sense to your users.

One approach is to defer the hook implementation to a new plugin instead of declaring the hook functions directly in your plugin module, for example:

```
# contents of myplugin.py

class DeferPlugin:
    """Simple plugin to defer pytest-xdist hook functions."""

    def pytest_testnodedown(self, node, error):
        """standard xdist hook function."""

def pytest_configure(config):
    if config.pluginmanager.hasplugin("xdist"):
        config.pluginmanager.register(DeferPlugin())
```

This has the added benefit of allowing you to conditionally install hooks depending on which plugins are installed.

2.16.8 Storing data on items across hook functions

Plugins often need to store data on *Items* in one hook implementation, and access it in another. One common solution is to just assign some private attribute directly on the item, but type-checkers like mypy frown upon this, and it may also cause conflicts with other plugins. So pytest offers a better way to do this, `_pytest.nodes.Node.stash`.

To use the “stash” in your plugins, first create “stash keys” somewhere at the top level of your plugin:

```
been_there_key = pytest.StashKey[bool]()
done_that_key = pytest.StashKey[str]()
```

then use the keys to stash your data at some point:

```
def pytest_runtest_setup(item: pytest.Item) -> None:
    item.stash[been_there_key] = True
    item.stash[done_that_key] = "no"
```

and retrieve them at another point:

```
def pytest_runtest_teardown(item: pytest.Item) -> None:
    if not item.stash[been_there_key]:
        print("Oh?")
    item.stash[done_that_key] = "yes!"
```

Stashes are available on all node types (like *Class*, *Session*) and also on *Config*, if needed.

2.17 How to use pytest with an existing test suite

Pytest can be used with most existing test suites, but its behavior differs from other test runners such as *nose* or Python’s default unittest framework.

Before using this section you will want to *install pytest*.

2.17.1 Running an existing test suite with pytest

Say you want to contribute to an existing repository somewhere. After pulling the code into your development space using some flavor of version control and (optionally) setting up a virtualenv you will want to run:

```
cd <repository>
pip install -e . # Environment dependent alternatives include
                # 'python setup.py develop' and 'conda develop'
```

in your project root. This will set up a symlink to your code in site-packages, allowing you to edit your code while your tests run against it as if it were installed.

Setting up your project in development mode lets you avoid having to reinstall every time you want to run your tests, and is less brittle than mucking about with `sys.path` to point your tests at local code.

Also consider using *tox*.

2.18 How to use unittest-based tests with pytest

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`;
- `setUpModule/tearDownModule`;

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

- `load_tests` protocol;
- subtests;

2.18.1 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;
- `maxfail`;
- `-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);

2.18.2 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.

2.18.3 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:
        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 de-couples 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-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items
```

(continues on next page)

(continued from previous page)

```

test_unittest_db.py FF [100%]
===== 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 0xdeadbeef0001>
E       assert 0

test_unittest_db.py:10: AssertionError
_____ MyTest.test_method2 _____

self = <test_unittest_db.MyTest testMethod=test_method2>

    def test_method2(self):
>       assert 0, self.db # fail for demo purposes
E       AssertionError: <conftest.db_class.<locals>.DummyDB object at 0xdeadbeef0001>
E       assert 0

test_unittest_db.py:13: AssertionError
===== short test summary info =====
FAILED test_unittest_db.py::MyTest::test_method1 - AssertionError: <conft...
FAILED test_unittest_db.py::MyTest::test_method2 - AssertionError: <conft...
===== 2 failed in 0.12s =====

```

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.

2.18.4 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 `tmp_path` fixture to delegate the creation of a per-test temporary directory:

```

# content of test_unittest_cleandir.py
import os
import pytest
import unittest

class MyTest(unittest.TestCase):
    @pytest.fixture(autouse=True)
    def initdir(self, tmp_path, monkeypatch):
        monkeypatch.chdir(tmp_path) # change to pytest-provided temporary directory
        tmp_path.joinpath("samplefile.ini").write_text("# testdata")

```

(continues on next page)

(continued from previous page)

```
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 0.12s [100%]
```

... 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.

Note: Due to architectural differences between the two frameworks, `setup` and `teardown` for `unittest`-based tests is performed during the `call` phase of testing instead of in `pytest`'s standard `setup` and `teardown` stages. This can be important to understand in some situations, particularly when reasoning about errors. For example, if a `unittest`-based suite exhibits errors during `setup`, `pytest` will report no errors during its `setup` phase and will instead raise the error during `call`.

2.19 How to run tests written for nose

`pytest` has basic support for running tests written for `nose`.

2.19.1 Usage

After *Install `pytest`* type:

```
python setup.py develop # make sure tests can import our package
pytest # instead of 'nosetests'
```

and you should be able to run your `nose` style tests and make use of `pytest`'s capabilities.

2.19.2 Supported nose Idioms

- setup and teardown at module/class/method level
- SkipTest exceptions and markers
- setup/teardown decorators
- `__test__` attribute on modules/classes/functions
- general usage of nose utilities

2.19.3 Unsupported idioms / known issues

- unittest-style `setUp`, `tearDown`, `setUpClass`, `tearDownClass` are recognized only on `unittest.TestCase` classes but not on plain classes. nose supports these methods also on plain classes but pytest deliberately does not. As nose and pytest already both support `setup_class`, `teardown_class`, `setup_method`, `teardown_method` it doesn't seem useful to duplicate the unittest-API like nose does. If you however rather think pytest should support the unittest-spelling on plain classes please post [to this issue](#).
- nose imports test modules with the same import path (e.g. `tests.test_mode`) but different file system paths (e.g. `tests/test_mode.py` and `other/tests/test_mode.py`) by extending `sys.path/import` semantics. pytest does not do that but there is discussion in [#268](#) for adding some support. Note that `nose2` choose to avoid this `sys.path/import` hackery.

If you place a `confest.py` file in the root directory of your project (as determined by pytest) pytest will run tests “nose style” against the code below that directory by adding it to your `sys.path` instead of running against your installed code.

You may find yourself wanting to do this if you ran `python setup.py install` to set up your project, as opposed to `python setup.py develop` or any of the package manager equivalents. Installing with `develop` in a virtual environment like `tox` is recommended over this pattern.

- nose-style doctests are not collected and executed correctly, also doctest fixtures don't work.
- no nose-configuration is recognized.
- `yield`-based methods are unsupported as of pytest 4.1.0. They are fundamentally incompatible with pytest because they don't support fixtures properly since collection and test execution are separated.

2.19.4 Migrating from nose to pytest

`nose2pytest` is a Python script and pytest plugin to help convert Nose-based tests into pytest-based tests. Specifically, the script transforms `nose.tools.assert_*` function calls into raw `assert` statements, while preserving format of original arguments as much as possible.

2.20 How to implement xunit-style set-up

This section describes a classic and popular way how you can implement fixtures (setup and teardown test state) on a per-module/class/function basis.

Note: While these setup/teardown methods are simple and familiar to those coming from a `unittest` or `nose` background, you may also consider using pytest's more powerful *fixture mechanism* which leverages the concept of dependency injection, allowing for a more modular and more scalable approach for managing test state, especially for larger projects

and for functional testing. You can mix both fixture mechanisms in the same file but test methods of `unittest.TestCase` subclasses cannot receive fixture arguments.

2.20.1 Module level setup/teardown

If you have multiple test functions and test classes in a single module you can optionally implement the following fixture methods which will usually be called once for all the functions:

```
def setup_module(module):
    """ setup any state specific to the execution of the given module."""

def teardown_module(module):
    """teardown any state that was previously setup with a setup_module
    method.
    """
```

As of pytest-3.0, the `module` parameter is optional.

2.20.2 Class level setup/teardown

Similarly, the following methods are called at class level before and after all test methods of the class are called:

```
@classmethod
def setup_class(cls):
    """setup any state specific to the execution of the given class (which
    usually contains tests).
    """

@classmethod
def teardown_class(cls):
    """teardown any state that was previously setup with a call to
    setup_class.
    """
```

2.20.3 Method and function level setup/teardown

Similarly, the following methods are called around each method invocation:

```
def setup_method(self, method):
    """setup any state tied to the execution of the given method in a
    class. setup_method is invoked for every test method of a class.
    """

def teardown_method(self, method):
    """teardown any state that was previously setup with a setup_method
    call.
    """
```

As of pytest-3.0, the `method` parameter is optional.

If you would rather define test functions directly at module level you can also use the following functions to implement fixtures:

```
def setup_function(function):
    """setup any state tied to the execution of the given function.
    Invoked for every test function in the module.
    """

def teardown_function(function):
    """teardown any state that was previously setup with a setup_function
    call.
    """
```

As of pytest-3.0, the function parameter is optional.

Remarks:

- It is possible for setup/teardown pairs to be invoked multiple times per testing process.
- teardown functions are not called if the corresponding setup function existed and failed/was skipped.
- Prior to pytest-4.2, xunit-style functions did not obey the scope rules of fixtures, so it was possible, for example, for a `setup_method` to be called before a session-scoped autouse fixture.

Now the xunit-style functions are integrated with the fixture mechanism and obey the proper scope rules of fixtures involved in the call.

2.21 How to set up bash completion

When using bash as your shell, `pytest` can use `argcomplete` (<https://argcomplete.readthedocs.io/>) for auto-completion. For this `argcomplete` needs to be installed **and** enabled.

Install `argcomplete` using:

```
sudo pip install 'argcomplete>=0.5.7'
```

For global activation of all `argcomplete` enabled python applications run:

```
sudo activate-global-python-argcomplete
```

For permanent (but not global) `pytest` activation, use:

```
register-python-argcomplete pytest >> ~/.bashrc
```

For one-time activation of `argcomplete` for `pytest` only, use:

```
eval "$(register-python-argcomplete pytest)"
```


REFERENCE GUIDES

3.1 Fixtures reference

See also:

About fixtures

See also:

How to use fixtures

3.1.1 Built-in fixtures

Fixtures are defined using the `@pytest.fixture` decorator. Pytest has several useful built-in fixtures:

`capfd` Capture, as text, output to file descriptors 1 and 2.

`capfdbinary` Capture, as bytes, output to file descriptors 1 and 2.

`caplog` Control logging and access log entries.

`capsys` Capture, as text, output to `sys.stdout` and `sys.stderr`.

`capsysbinary` Capture, as bytes, output to `sys.stdout` and `sys.stderr`.

`cache` Store and retrieve values across pytest runs.

`doctest_namespace` Provide a dict injected into the docstests namespace.

`monkeypatch` Temporarily modify classes, functions, dictionaries, `os.environ`, and other objects.

`pytestconfig` Access to configuration values, pluginmanager and plugin hooks.

`record_property` Add extra properties to the test.

`record_testsuite_property` Add extra properties to the test suite.

`recwarn` Record warnings emitted by test functions.

`request` Provide information on the executing test function.

`testdir` Provide a temporary test directory to aid in running, and testing, pytest plugins.

`tmp_path` Provide a `pathlib.Path` object to a temporary directory which is unique to each test function.

`tmp_path_factory` Make session-scoped temporary directories and return `pathlib.Path` objects.

`tmpdir` Provide a `py.path.local` object to a temporary directory which is unique to each test function; replaced by `tmp_path`.

`tmpdir_factory` Make session-scoped temporary directories and return `py.path.local` objects; replaced by `tmp_path_factory`.

3.1.2 Fixture availability

Fixture availability is determined from the perspective of the test. A fixture is only available for tests to request if they are in the scope that fixture is defined in. If a fixture is defined inside a class, it can only be requested by tests inside that class. But if a fixture is defined inside the global scope of the module, then every test in that module, even if it's defined inside a class, can request it.

Similarly, a test can also only be affected by an autouse fixture if that test is in the same scope that autouse fixture is defined in (see *Autouse fixtures are executed first within their scope*).

A fixture can also request any other fixture, no matter where it's defined, so long as the test requesting them can see all fixtures involved.

For example, here's a test file with a fixture (`outer`) that requests a fixture (`inner`) from a scope it wasn't defined in:

```
import pytest

@pytest.fixture
def order():
    return []

@pytest.fixture
def outer(order, inner):
    order.append("outer")

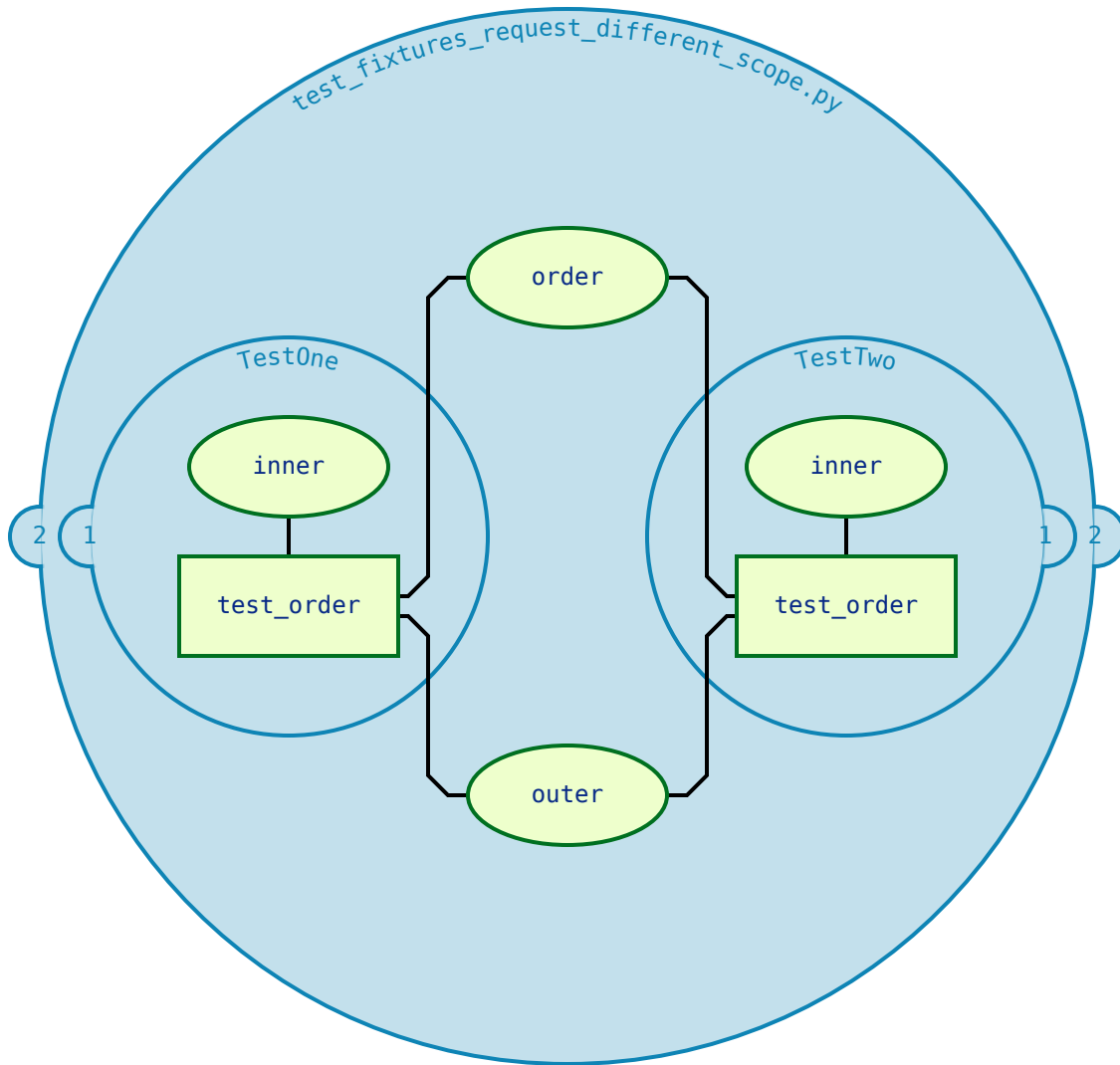
class TestOne:
    @pytest.fixture
    def inner(self, order):
        order.append("one")

    def test_order(self, order, outer):
        assert order == ["one", "outer"]

class TestTwo:
    @pytest.fixture
    def inner(self, order):
        order.append("two")

    def test_order(self, order, outer):
        assert order == ["two", "outer"]
```

From the tests' perspectives, they have no problem seeing each of the fixtures they're dependent on:



So when they run, `outer` will have no problem finding `inner`, because `pytest` searched from the tests' perspectives.

Note: The scope a fixture is defined in has no bearing on the order it will be instantiated in: the order is mandated by the logic described [here](#).

`conftest.py`: sharing fixtures across multiple files

The `conftest.py` file serves as a means of providing fixtures for an entire directory. Fixtures defined in a `conftest.py` can be used by any test in that package without needing to import them (`pytest` will automatically discover them).

You can have multiple nested directories/packages containing your tests, and each directory can have its own `conftest.py` with its own fixtures, adding on to the ones provided by the `conftest.py` files in parent directories.

For example, given a test file structure like this:

```
tests/
  __init__.py
```

(continues on next page)

```
confptest.py
# content of tests/confptest.py
import pytest

@pytest.fixture
def order():
    return []

@pytest.fixture
def top(order, innermost):
    order.append("top")

test_top.py
# content of tests/test_top.py
import pytest

@pytest.fixture
def innermost(order):
    order.append("innermost top")

def test_order(order, top):
    assert order == ["innermost top", "top"]

subpackage/
__init__.py

confptest.py
# content of tests/subpackage/confptest.py
import pytest

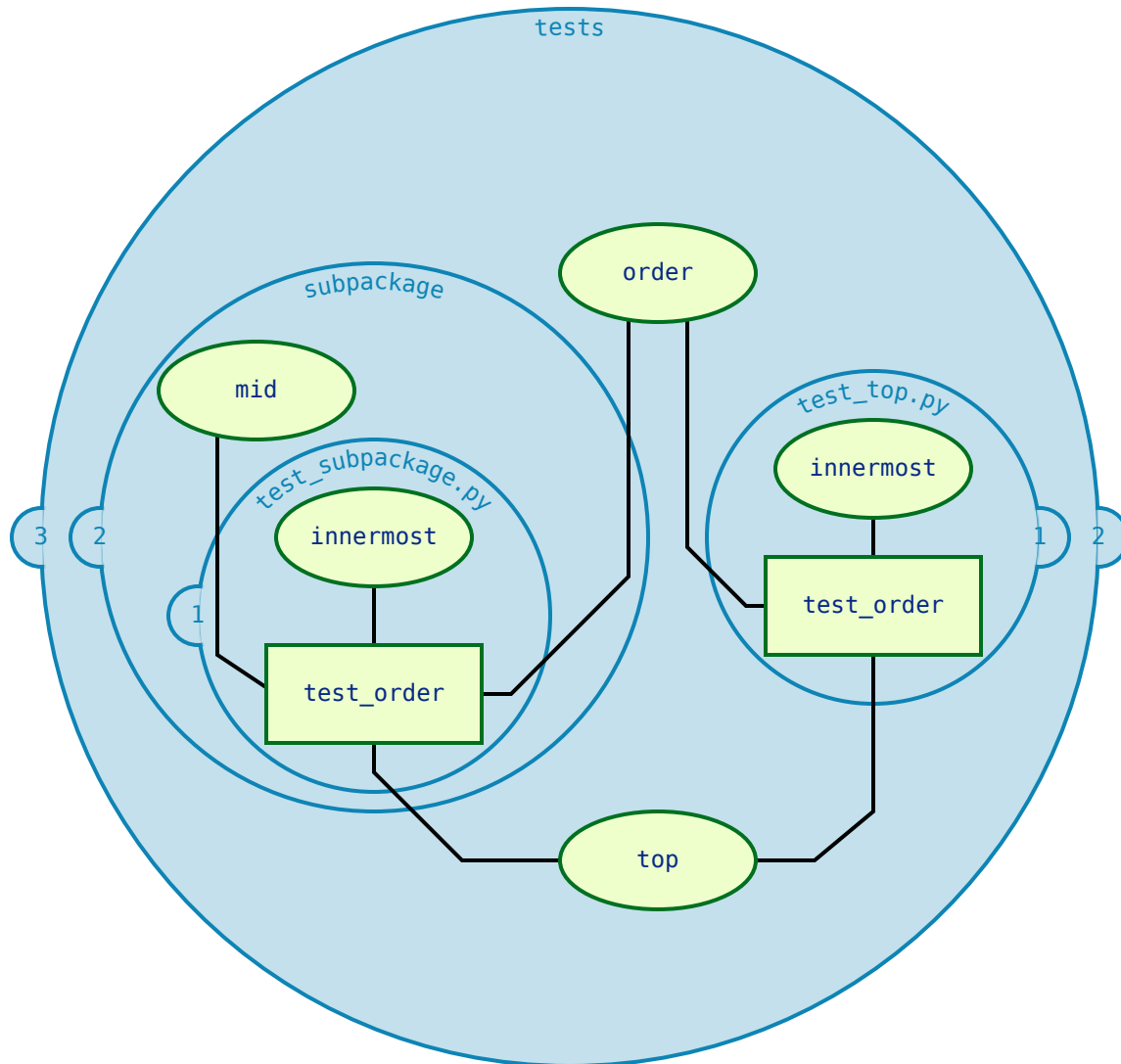
@pytest.fixture
def mid(order):
    order.append("mid subpackage")

test_subpackage.py
# content of tests/subpackage/test_subpackage.py
import pytest

@pytest.fixture
def innermost(order, mid):
    order.append("innermost subpackage")

def test_order(order, top):
    assert order == ["mid subpackage", "innermost subpackage", "top"]
```

The boundaries of the scopes can be visualized like this:



The directories become their own sort of scope where fixtures that are defined in a `conftest.py` file in that directory become available for that whole scope.

Tests are allowed to search upward (stepping outside a circle) for fixtures, but can never go down (stepping inside a circle) to continue their search. So `tests/subpackage/test_subpackage.py::test_order` would be able to find the `innermost` fixture defined in `tests/subpackage/test_subpackage.py`, but the one defined in `tests/test_top.py` would be unavailable to it because it would have to step down a level (step inside a circle) to find it.

The first fixture the test finds is the one that will be used, so *fixtures can be overridden* if you need to change or extend what one does for a particular scope.

You can also use the `conftest.py` file to implement *local per-directory plugins*.

Fixtures from third-party plugins

Fixtures don't have to be defined in this structure to be available for tests, though. They can also be provided by third-party plugins that are installed, and this is how many pytest plugins operate. As long as those plugins are installed, the fixtures they provide can be requested from anywhere in your test suite.

Because they're provided from outside the structure of your test suite, third-party plugins don't really provide a scope like `conftest.py` files and the directories in your test suite do. As a result, pytest will search for fixtures stepping out through scopes as explained previously, only reaching fixtures defined in plugins *last*.

For example, given the following file structure:

```
tests/
  __init__.py

  conftest.py
    # content of tests/conftest.py
    import pytest

    @pytest.fixture
    def order():
        return []

  subpackage/
    __init__.py

    conftest.py
      # content of tests/subpackage/conftest.py
      import pytest

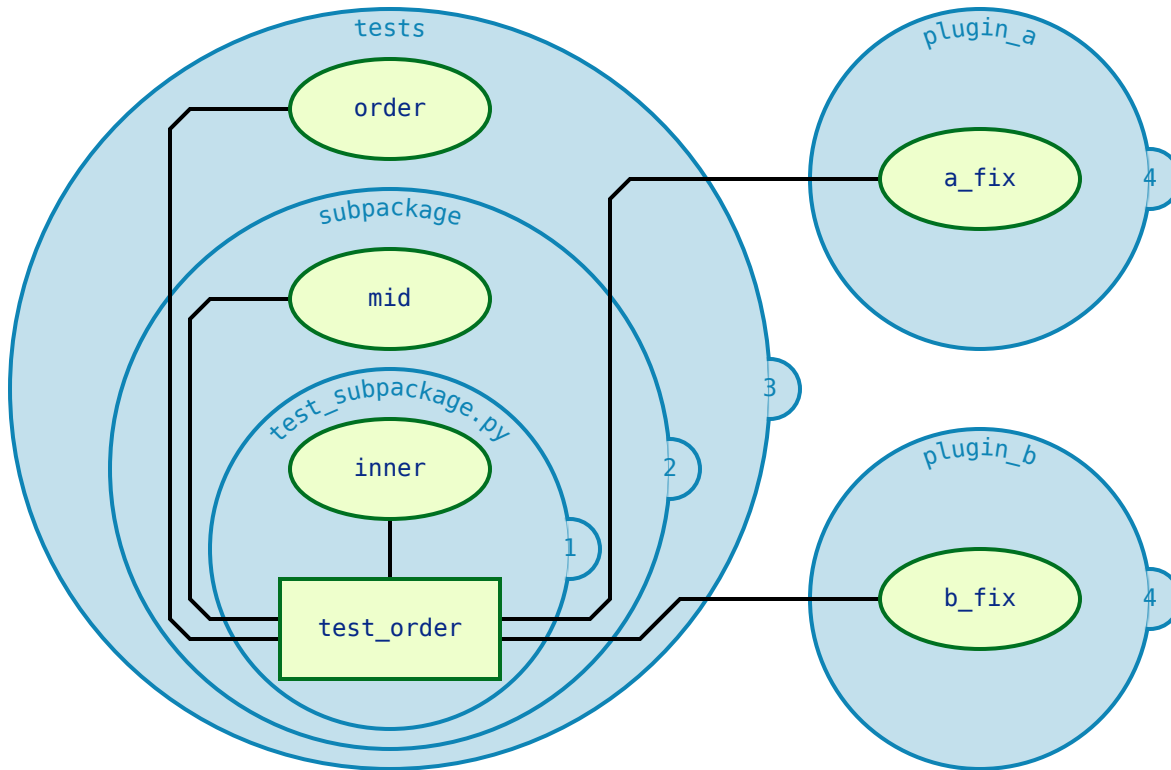
      @pytest.fixture(autouse=True)
      def mid(order, b_fix):
          order.append("mid subpackage")

    test_subpackage.py
      # content of tests/subpackage/test_subpackage.py
      import pytest

      @pytest.fixture
      def inner(order, mid, a_fix):
          order.append("inner subpackage")

      def test_order(order, inner):
          assert order == ["b_fix", "mid subpackage", "a_fix", "inner subpackage"]
↪ ]
```

If `plugin_a` is installed and provides the fixture `a_fix`, and `plugin_b` is installed and provides the fixture `b_fix`, then this is what the test's search for fixtures would look like:



pytest will only search for `a_fix` and `b_fix` in the plugins after searching for them first in the scopes inside `tests/`.

3.1.3 Fixture instantiation order

When pytest wants to execute a test, once it knows what fixtures will be executed, it has to figure out the order they'll be executed in. To do this, it considers 3 factors:

1. scope
2. dependencies
3. autouse

Names of fixtures or tests, where they're defined, the order they're defined in, and the order fixtures are requested in have no bearing on execution order beyond coincidence. While pytest will try to make sure coincidences like these stay consistent from run to run, it's not something that should be depended on. If you want to control the order, it's safest to rely on these 3 things and make sure dependencies are clearly established.

Higher-scoped fixtures are executed first

Within a function request for fixtures, those of higher-scopes (such as `session`) are executed before lower-scoped fixtures (such as `function` or `class`).

Here's an example:

```
import pytest

@pytest.fixture(scope="session")
def order():
```

(continues on next page)

```
    return []

@pytest.fixture
def func(order):
    order.append("function")

@pytest.fixture(scope="class")
def cls(order):
    order.append("class")

@pytest.fixture(scope="module")
def mod(order):
    order.append("module")

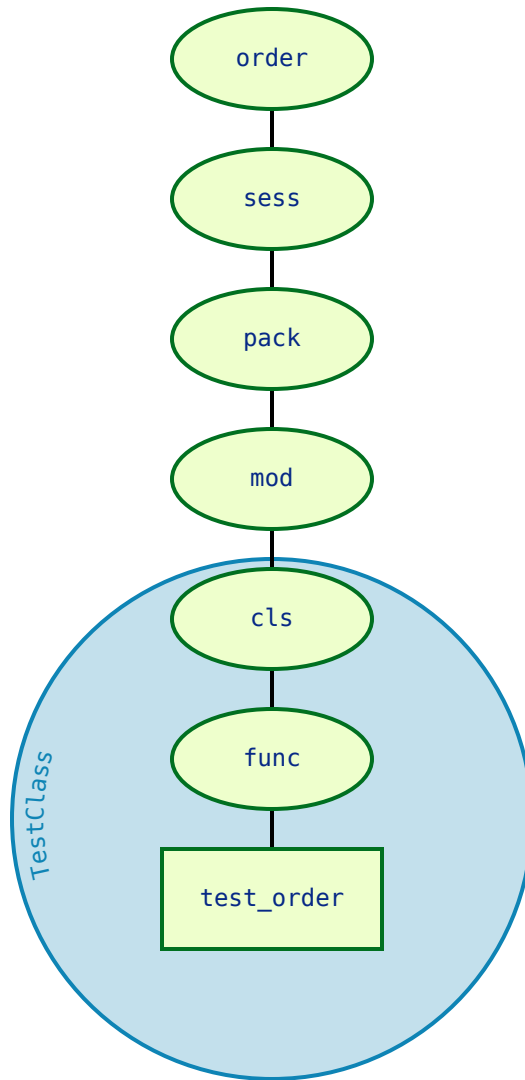
@pytest.fixture(scope="package")
def pack(order):
    order.append("package")

@pytest.fixture(scope="session")
def sess(order):
    order.append("session")

class TestClass:
    def test_order(self, func, cls, mod, pack, sess, order):
        assert order == ["session", "package", "module", "class", "function"]
```

The test will pass because the larger scoped fixtures are executing first.

The order breaks down to this:



Fixtures of the same order execute based on dependencies

When a fixture requests another fixture, the other fixture is executed first. So if fixture a requests fixture b, fixture b will execute first, because a depends on b and can't operate without it. Even if a doesn't need the result of b, it can still request b if it needs to make sure it is executed after b.

For example:

```

import pytest

@pytest.fixture
def order():
    return []

@pytest.fixture
def a(order):
    order.append("a")
  
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture
def b(a, order):
    order.append("b")

@pytest.fixture
def c(a, b, order):
    order.append("c")

@pytest.fixture
def d(c, b, order):
    order.append("d")

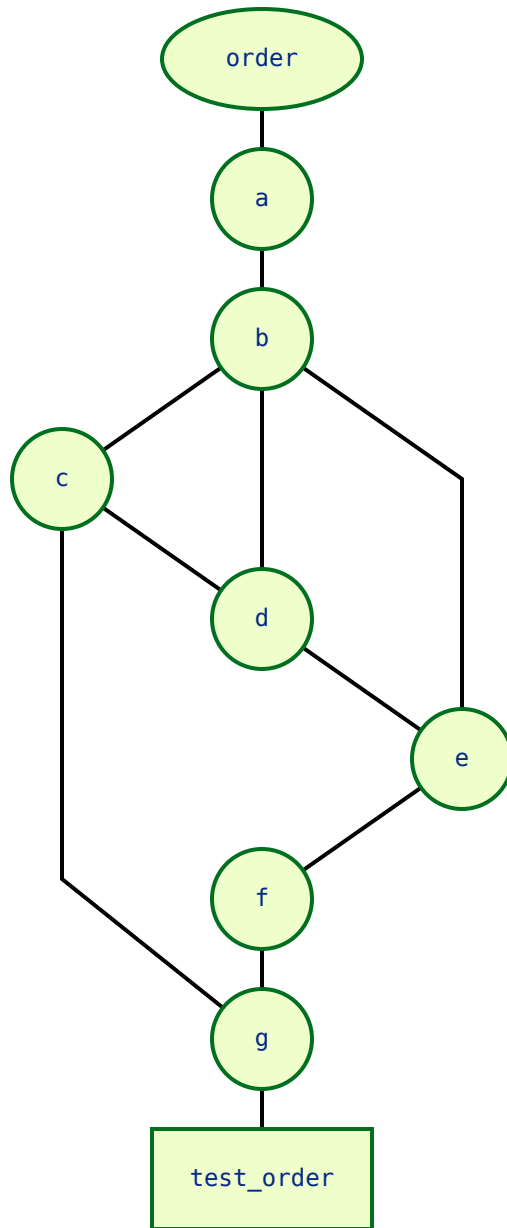
@pytest.fixture
def e(d, b, order):
    order.append("e")

@pytest.fixture
def f(e, order):
    order.append("f")

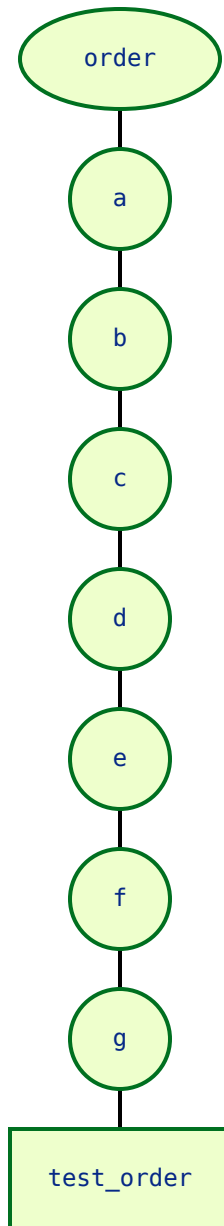
@pytest.fixture
def g(f, c, order):
    order.append("g")

def test_order(g, order):
    assert order == ["a", "b", "c", "d", "e", "f", "g"]
```

If we map out what depends on what, we get something that look like this:

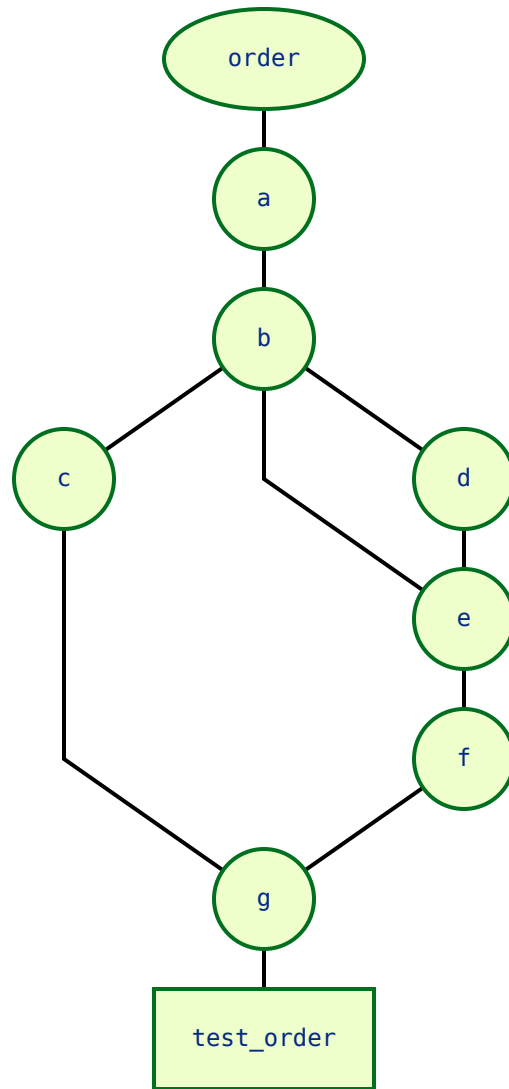


The rules provided by each fixture (as to what fixture(s) each one has to come after) are comprehensive enough that it can be flattened to this:



Enough information has to be provided through these requests in order for pytest to be able to figure out a clear, linear chain of dependencies, and as a result, an order of operations for a given test. If there's any ambiguity, and the order of operations can be interpreted more than one way, you should assume pytest could go with any one of those interpretations at any point.

For example, if `d` didn't request `c`, i.e. the graph would look like this:



Because nothing requested `c` other than `g`, and `g` also requests `f`, it's now unclear if `c` should go before/after `f`, `e`, or `d`. The only rules that were set for `c` is that it must execute after `b` and before `g`.

pytest doesn't know where `c` should go in the case, so it should be assumed that it could go anywhere between `g` and `b`.

This isn't necessarily bad, but it's something to keep in mind. If the order they execute in could affect the behavior a test is targeting, or could otherwise influence the result of a test, then the order should be defined explicitly in a way that allows pytest to linearize/"flatten" that order.

Autouse fixtures are executed first within their scope

Autouse fixtures are assumed to apply to every test that could reference them, so they are executed before other fixtures in that scope. Fixtures that are requested by autouse fixtures effectively become autouse fixtures themselves for the tests that the real autouse fixture applies to.

So if fixture `a` is autouse and fixture `b` is not, but fixture `a` requests fixture `b`, then fixture `b` will effectively be an autouse fixture as well, but only for the tests that `a` applies to.

In the last example, the graph became unclear if `d` didn't request `c`. But if `c` was autouse, then `b` and `a` would effectively also be autouse because `c` depends on them. As a result, they would all be shifted above non-autouse fixtures within that

scope.

So if the test file looked like this:

```
import pytest

@pytest.fixture
def order():
    return []

@pytest.fixture
def a(order):
    order.append("a")

@pytest.fixture
def b(a, order):
    order.append("b")

@pytest.fixture(autouse=True)
def c(b, order):
    order.append("c")

@pytest.fixture
def d(b, order):
    order.append("d")

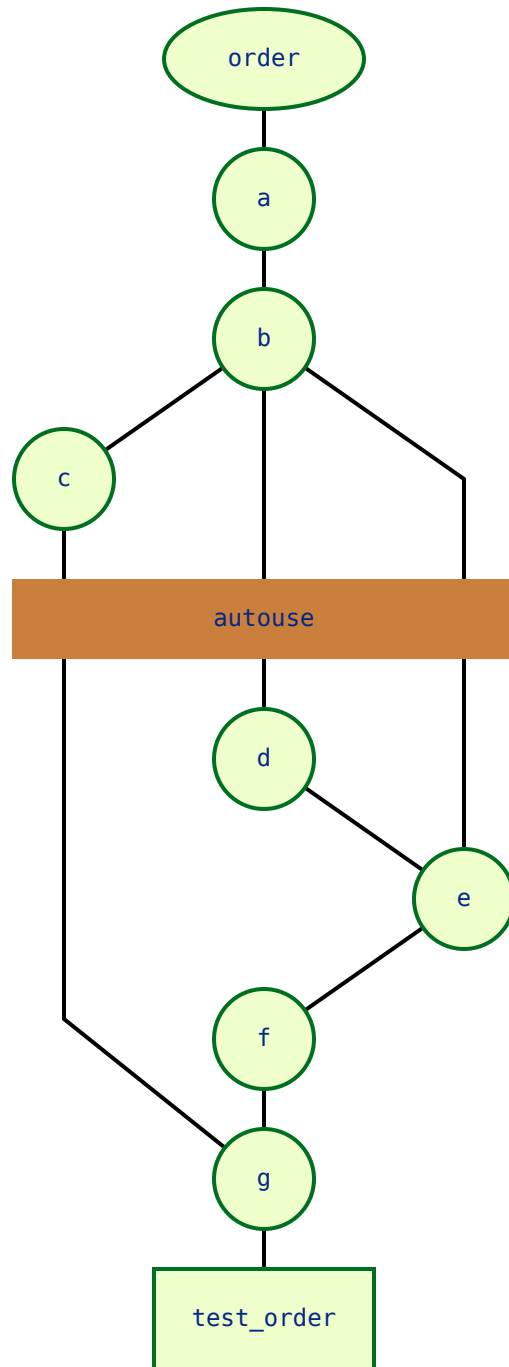
@pytest.fixture
def e(d, order):
    order.append("e")

@pytest.fixture
def f(e, order):
    order.append("f")

@pytest.fixture
def g(f, c, order):
    order.append("g")

def test_order_and_g(g, order):
    assert order == ["a", "b", "c", "d", "e", "f", "g"]
```

the graph would look like this:



Because `c` can now be put above `d` in the graph, pytest can once again linearize the graph to this:

In this example, `c` makes `b` and `a` effectively `autouse` fixtures as well.

Be careful with `autouse`, though, as an `autouse` fixture will automatically execute for every test that can reach it, even if they don't request it. For example, consider this file:

```
import pytest

@pytest.fixture(scope="class")
```

(continues on next page)

(continued from previous page)

```
def order():
    return []

@pytest.fixture(scope="class", autouse=True)
def c1(order):
    order.append("c1")

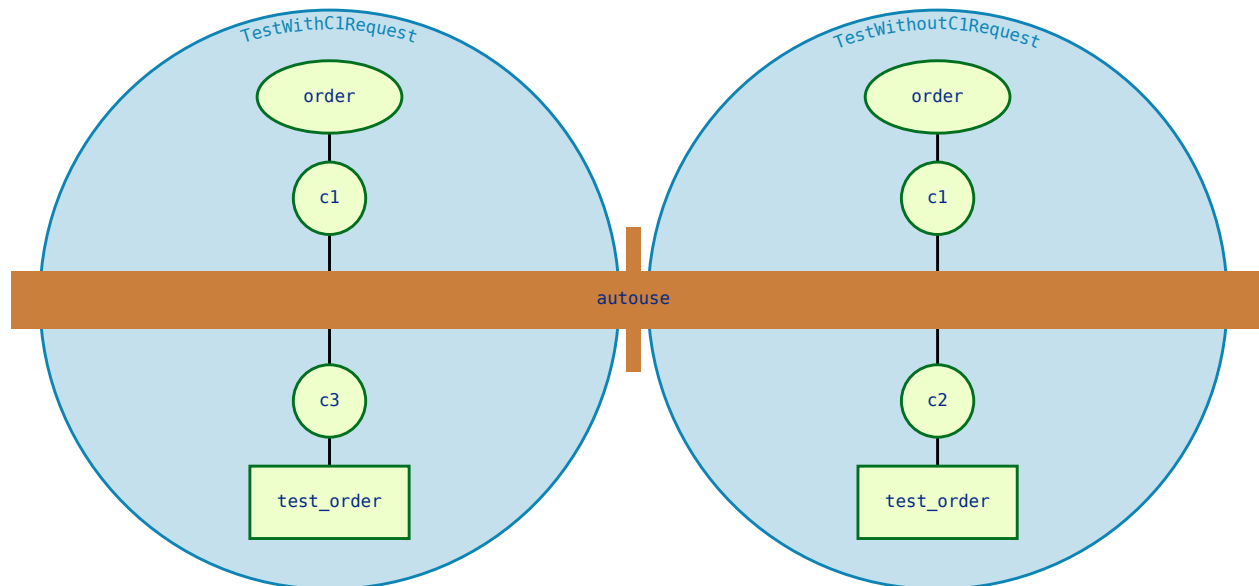
@pytest.fixture(scope="class")
def c2(order):
    order.append("c2")

@pytest.fixture(scope="class")
def c3(order, c1):
    order.append("c3")

class TestClassWithC1Request:
    def test_order(self, order, c1, c3):
        assert order == ["c1", "c3"]

class TestClassWithoutC1Request:
    def test_order(self, order, c2):
        assert order == ["c1", "c2"]
```

Even though nothing in `TestClassWithoutC1Request` is requesting `c1`, it still is executed for the tests inside it anyway:



But just because one `autouse` fixture requested a non-`autouse` fixture, that doesn't mean the non-`autouse` fixture becomes an `autouse` fixture for all contexts that it can apply to. It only effectively becomes an `autouse` fixture for the contexts the real `autouse` fixture (the one that requested the non-`autouse` fixture) can apply to.

For example, take a look at this test file:


```
import pytest

@pytest.fixture
def order():
    return []

@pytest.fixture
def c1(order):
    order.append("c1")

@pytest.fixture
def c2(order):
    order.append("c2")

class TestClassWithAutouse:
    @pytest.fixture(autouse=True)
    def c3(self, order, c2):
        order.append("c3")

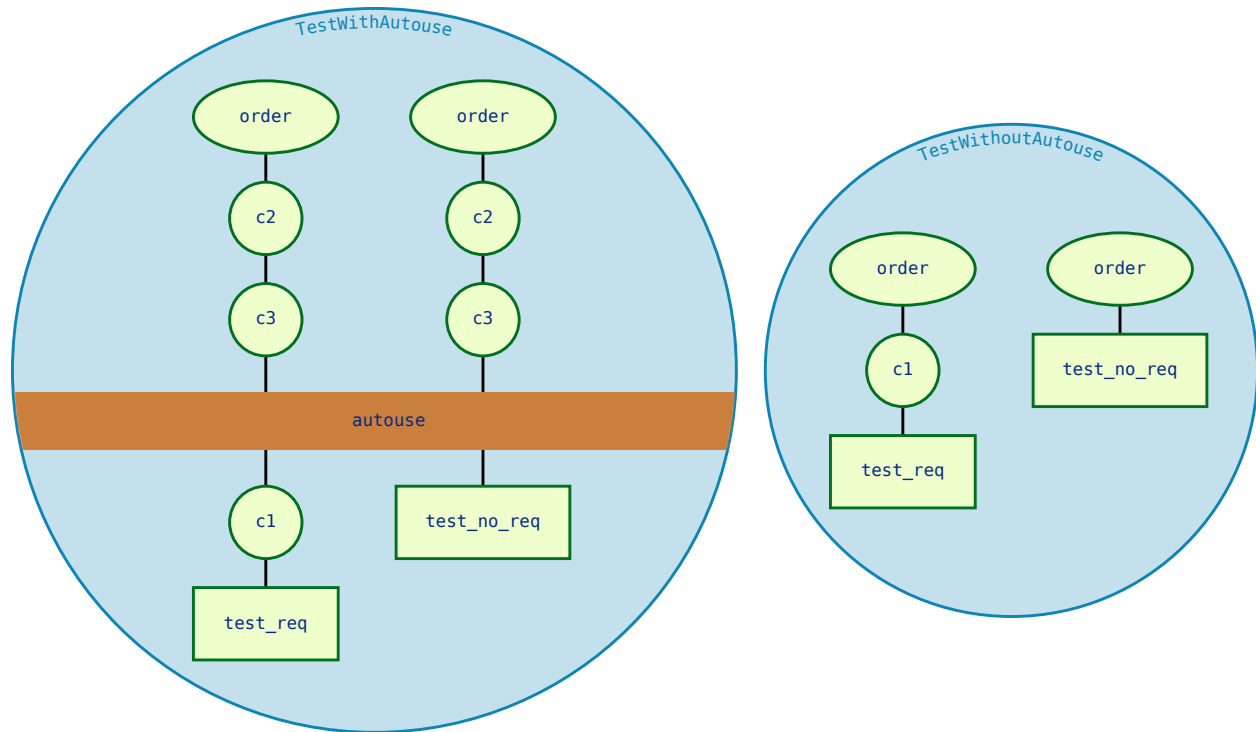
    def test_req(self, order, c1):
        assert order == ["c2", "c3", "c1"]

    def test_no_req(self, order):
        assert order == ["c2", "c3"]

class TestClassWithoutAutouse:
    def test_req(self, order, c1):
        assert order == ["c1"]

    def test_no_req(self, order):
        assert order == []
```

It would break down to something like this:



For `test_req` and `test_no_req` inside `TestClassWithAutouse`, `c3` effectively makes `c2` an `autouse` fixture, which is why `c2` and `c3` are executed for both tests, despite not being requested, and why `c2` and `c3` are executed before `c1` for `test_req`.

If this made `c2` an *actual* `autouse` fixture, then `c2` would also execute for the tests inside `TestClassWithoutAutouse`, since they can reference `c2` if they wanted to. But it doesn't, because from the perspective of the `TestClassWithoutAutouse` tests, `c2` isn't an `autouse` fixture, since they can't see `c3`.

3.2 Plugin List

PyPI projects that match “`pytest-*`” are considered plugins and are listed automatically. Packages classified as inactive are excluded. This list contains 942 plugins.

name	summary
pytest-accept	A pytest-plugin for updating doctest outputs
pytest-adaptavist	pytest plugin for generating test execution results within Jira Test Management (tm4j)
pytest-addons-test	用于测试pytest的插件
pytest-adf	Pytest plugin for writing Azure Data Factory integration tests
pytest-adf-azure-identity	Pytest plugin for writing Azure Data Factory integration tests
pytest-aggreport	pytest plugin for pytest-repeat that generate aggregate report of the same test cases with addit
pytest-aiio	Pytest plugin for testing async python code
pytest-aiofiles	pytest fixtures for writing aiofiles tests with pyfakefs
pytest-aiohttp	pytest plugin for aiohttp support
pytest-aiohttp-client	Pytest `client` fixture for the Aiohttp
pytest-aioresponses	py.test integration for aioresponses
pytest-aioworkers	A plugin to test aioworkers project with pytest
pytest-airflow	pytest support for airflow.

name	summary
pytest-alembic	A pytest plugin for verifying alembic migrations.
pytest-allclose	Pytest fixture extending Numpy's allclose function
pytest-allure-adaptor	Plugin for py.test to generate allure xml reports
pytest-allure-adaptor2	Plugin for py.test to generate allure xml reports
pytest-allure-dsl	pytest plugin to test case doc string dls instructions
pytest-alphamoon	Static code checks used at Alphamoon
pytest-android	This fixture provides a configured "driver" for Android Automated Testing, using uiautomato
pytest-anki	A pytest plugin for testing Anki add-ons
pytest-annotate	pytest-annotate: Generate PyAnnotate annotations from your pytest tests.
pytest-ansible	Plugin for py.test to simplify calling ansible modules from tests or fixtures
pytest-ansible-playbook	Pytest fixture which runs given ansible playbook file.
pytest-ansible-playbook-runner	Pytest fixture which runs given ansible playbook file.
pytest-antilru	Bust functools.lru_cache when running pytest to avoid test pollution
pytest-anyio	The pytest anyio plugin is built into anyio. You don't need this package.
pytest-anything	Pytest fixtures to assert anything and something
pytest-aoc	Downloads puzzle inputs for Advent of Code and synthesizes PyTest fixtures
pytest-api	PyTest-API Python Web Framework built for testing purposes.
pytest-apistellar	apistellar plugin for pytest.
pytest-appengine	AppEngine integration that works well with pytest-django
pytest-appium	Pytest plugin for appium
pytest-approvaltests	A plugin to use approvaltests with pytest
pytest-argus	pyest results colection plugin
pytest-arraydiff	pytest plugin to help with comparing array output from tests
pytest-asgi-server	Convenient ASGI client/server fixtures for Pytest
pytest-asptest	test Answer Set Programming programs
pytest-assertutil	pytest-assertutil
pytest-assert-utils	Useful assertion utilities for use with pytest
pytest-assume	A pytest plugin that allows multiple failures per test
pytest-ast-back-to-python	A plugin for pytest devs to view how assertion rewriting recodes the AST
pytest-astropy	Meta-package containing dependencies for testing
pytest-astropy-header	pytest plugin to add diagnostic information to the header of the test output
pytest-ast-transformer	
pytest-asyncio	Pytest support for asyncio.
pytest-asyncio-cooperative	Run all your asynchronous tests cooperatively.
pytest-asyncio-network-simulator	pytest-asyncio-network-simulator: Plugin for pytest for simulator the network in tests
pytest-async-mongodb	pytest plugin for async MongoDB
pytest-async-sqlalchemy	Database testing fixtures using the SQLAlchemy asyncio API
pytest-atomic	Skip rest of tests if previous test failed.
pytest-attrib	pytest plugin to select tests based on attributes similar to the nose-attrib plugin
pytest-austin	Austin plugin for pytest
pytest-autochecklog	automatically check condition and log all the checks
pytest-automation	pytest plugin for building a test suite, using YAML files to extend pytest parameterize functio
pytest-automock	Pytest plugin for automatical mocks creation
pytest-auto-parametrize	pytest plugin: avoid repeating arguments in parametrize
pytest-autotest	This fixture provides a configured "driver" for Android Automated Testing, using uiautomato
pytest-avoidance	Makes pytest skip tests that don not need rerunning
pytest-aws	pytest plugin for testing AWS resource configurations
pytest-aws-config	Protect your AWS credentials in unit tests
pytest-axe	pytest plugin for axe-selenium-python

name	summary
pytest-azurepipelines	Formatting PyTest output for Azure Pipelines UI
pytest-bandit	A bandit plugin for pytest
pytest-base-url	pytest plugin for URL based testing
pytest-bdd	BDD for pytest
pytest-bdd-splinter	Common steps for pytest bdd and splinter integration
pytest-bdd-web	A simple plugin to use with pytest
pytest-bdd-wrappers	
pytest-beakerlib	A pytest plugin that reports test results to the BeakerLib framework
pytest-beds	Fixtures for testing Google Appengine (GAE) apps
pytest-bench	Benchmark utility that plugs into pytest.
pytest-benchmark	A ``pytest`` fixture for benchmarking code. It will group the tests into rounds that are calibrated
pytest-bg-process	Pytest plugin to initialize background process
pytest-bigchaindb	A BigchainDB plugin for pytest.
pytest-bigquery-mock	Provides a mock fixture for python bigquery client
pytest-black	A pytest plugin to enable format checking with black
pytest-black-multiply	Allow '-black' on older Pythons
pytest-blame	A pytest plugin helps developers to debug by providing useful commits history.
pytest-blender	Blender Pytest plugin.
pytest-blink1	Pytest plugin to emit notifications via the Blink(1) RGB LED
pytest-blockage	Disable network requests during a test run.
pytest-blocker	pytest plugin to mark a test as blocker and skip all other tests
pytest-board	Local continuous test runner with pytest and watchdog.
pytest-bpdb	A py.test plug-in to enable drop to pdb debugger on test failure.
pytest-bravado	Pytest-bravado automatically generates from OpenAPI specification client fixtures.
pytest-breakword	Use breakword with pytest
pytest-breed-adapter	A simple plugin to connect with breed-server
pytest-briefcase	A pytest plugin for running tests on a Briefcase project.
pytest-browser	A pytest plugin for console based browser test selection just after the collection phase
pytest-browsermob-proxy	BrowserMob proxy plugin for py.test.
pytest-browserstack-local	``py.test`` plugin to run ``BrowserStackLocal`` in background.
pytest-bug	Pytest plugin for marking tests as a bug
pytest-bugtong-tag	pytest-bugtong-tag is a plugin for pytest
pytest-bugzilla	py.test bugzilla integration plugin
pytest-bugzilla-notifier	A plugin that allows you to execute create, update, and read information from BugZilla bugs
pytest-buildkite	Plugin for pytest that automatically publishes coverage and pytest report annotations to Buildkite
pytest-bwrap	Run your tests in Bubblewrap sandboxes
pytest-cache	pytest plugin with mechanisms for caching across test runs
pytest-cagoule	Pytest plugin to only run tests affected by changes
pytest-camel-collect	Enable CamelCase-aware pytest class collection
pytest-canonical-data	A plugin which allows to compare results with canonical results, based on previous runs
pytest-caprng	A plugin that replays pRNG state on failure.
pytest-capture-deprecatedwarnings	pytest plugin to capture all deprecatedwarnings and put them in one file
pytest-capturelogs	A sample Python project
pytest-cases	Separate test code from test cases in pytest.
pytest-cassandra	Cassandra CCM Test Fixtures for pytest
pytest-catchlog	py.test plugin to catch log messages. This is a fork of pytest-capturelog.
pytest-catch-server	Pytest plugin with server for catching HTTP requests.
pytest-celery	pytest-celery a shim pytest plugin to enable celery.contrib.pytest
pytest-chainmaker	pytest plugin for chainmaker

name	summary
pytest-chalice	A set of py.test fixtures for AWS Chalice
pytest-change-report	turn . into √, turn F into x
pytest-chdir	A pytest fixture for changing current working directory
pytest-checkdocs	check the README when running tests
pytest-checkipdb	plugin to check if there are ipdb debugs left
pytest-check-links	Check links in files
pytest-check-mk	pytest plugin to test Check_MK checks
pytest-circleci	py.test plugin for CircleCI
pytest-circleci-parallelized	Parallelize pytest across CircleCI workers.
pytest-ckan	Backport of CKAN 2.9 pytest plugin and fixtures to CAKN 2.8
pytest-clarify	A plugin providing an alternative, colourful diff output for failing assertions.
pytest-cldf	Easy quality control for CLDF datasets using pytest
pytest-click	Py.test plugin for Click
pytest-clld	
pytest-cloud	Distributed tests planner plugin for pytest testing framework.
pytest-cloudflare-worker	pytest plugin for testing cloudflare workers
pytest-cobra	PyTest plugin for testing Smart Contracts for Ethereum blockchain.
pytest-codeblocks	Test code blocks in your READMEs
pytest-codecheckers	pytest plugin to add source code sanity checks (pep8 and friends)
pytest-codecov	Pytest plugin for uploading pytest-cov results to codecov.io
pytest-codegen	Automatically create pytest test signatures
pytest-codestyle	pytest plugin to run pycodestyle
pytest-collect-formatter	Formatter for pytest collect output
pytest-collect-formatter2	Formatter for pytest collect output
pytest-colordots	Colorizes the progress indicators
pytest-commander	An interactive GUI test runner for PyTest
pytest-common-subject	pytest framework for testing different aspects of a common method
pytest-concurrent	Concurrently execute test cases with multithread, multiprocessing and gevent
pytest-config	Base configurations and utilities for developing your Python project test suite with pytest.
pytest-confluence-report	Package stands for pytest plugin to upload results into Confluence page.
pytest-console-scripts	Pytest plugin for testing console scripts
pytest-consul	pytest plugin with fixtures for testing consul aware apps
pytest-contextfixture	Define pytest fixtures as context managers.
pytest-contexts	A plugin to run tests written with the Contexts framework using pytest
pytest-cookies	The pytest plugin for your Cookiecutter templates. 🍪
pytest-couchdbkit	py.test extension for per-test couchdb databases using couchdbkit
pytest-count	count erros and send email
pytest-cov	Pytest plugin for measuring coverage.
pytest-cover	Pytest plugin for measuring coverage. Forked from `pytest-cov`.
pytest-coverage	
pytest-coverage-context	Coverage dynamic context support for PyTest, including sub-processes
pytest-cov-exclude	Pytest plugin for excluding tests based on coverage data
pytest-cpp	Use pytest's runner to discover and execute C++ tests
pytest-cram	Run cram tests with pytest.
pytest-crate	Manages CrateDB instances during your integration tests
pytest-cricri	A Cricri plugin for pytest.
pytest-crontab	add crontab task in crontab
pytest-csv	CSV output for pytest.
pytest-curio	Pytest support for curio.

name	summary
pytest-curl-report	pytest plugin to generate curl command line report
pytest-custom-concurrency	Custom grouping concurrence for pytest
pytest-custom-exit-code	Exit pytest test session with custom exit code in different scenarios
pytest-custom-nodeid	Custom grouping for pytest-xdist, rename test cases name and test cases nodeid, support allur
pytest-custom-report	Configure the symbols displayed for test outcomes
pytest-custom-scheduling	Custom grouping for pytest-xdist, rename test cases name and test cases nodeid, support allur
pytest-cython	A plugin for testing Cython extension modules
pytest-darker	A pytest plugin for checking of modified code using Darker
pytest-dash	pytest fixtures to run dash applications.
pytest-data	Useful functions for managing data for pytest fixtures
pytest-databricks	Pytest plugin for remote Databricks notebooks testing
pytest-datadir	pytest plugin for test data directories and files
pytest-datadir-mgr	Manager for test data providing downloads, caching of generated files, and a context for temp
pytest-datadir-ng	Fixtures for pytest allowing test functions/methods to easily retrieve test resources from the lo
pytest-data-file	Fixture “data” and “case_data” for test from yaml file
pytest-datafiles	py.test plugin to create a ‘tmpdir’ containing predefined files/directories.
pytest-datafixtures	Data fixtures for pytest made simple
pytest-data-from-files	pytest plugin to provide data from files loaded automatically
pytest-dataplugin	A pytest plugin for managing an archive of test data.
pytest-datarecorder	A py.test plugin recording and comparing test output.
pytest-datatest	A pytest plugin for test driven data-wrangling (this is the development version of datatest’s py
pytest-db	Session scope fixture “db” for mysql query or change
pytest-dbfixtures	Databases fixtures plugin for py.test.
pytest-dbt-adapter	A pytest plugin for testing dbt adapter plugins
pytest-dbus-notification	D-BUS notifications for pytest results.
pytest-deadfixtures	A simple plugin to list unused fixtures in pytest
pytest-deepcov	deepcov
pytest-defer	
pytest-demo-plugin	pytest示例插件
pytest-dependency	Manage dependencies of tests
pytest-depends	Tests that depend on other tests
pytest-deprecate	Mark tests as testing a deprecated feature with a warning note.
pytest-describe	Describe-style plugin for pytest
pytest-describe-it	plugin for rich text descriptions
pytest-devpi-server	DevPI server fixture for py.test
pytest-diamond	pytest plugin for diamond
pytest-dicom	pytest plugin to provide DICOM fixtures
pytest-dictsdiff	
pytest-diff	A simple plugin to use with pytest
pytest-disable	pytest plugin to disable a test and skip it from testrun
pytest-disable-plugin	Disable plugins per test
pytest-discord	A pytest plugin to notify test results to a Discord channel.
pytest-django	A Django plugin for pytest.
pytest-django-ahead	A Django plugin for pytest.
pytest-djangoapp	Nice pytest plugin to help you with Django pluggable application testing.
pytest-django-cache-xdist	A djangocachexdist plugin for pytest
pytest-django-casperjs	Integrate CasperJS with your django tests as a pytest fixture.
pytest-django-dotenv	Pytest plugin used to setup environment variables with django-dotenv
pytest-django-factories	Factories for your Django models that can be used as Pytest fixtures.

name	summary
pytest-django-gcir	A Django plugin for pytest.
pytest-django-haystack	Cleanup your Haystack indexes between tests
pytest-django-ifactory	A model instance factory for pytest-django
pytest-django-lite	The bare minimum to integrate py.test with Django.
pytest-django-liveserver-ssl	
pytest-django-model	A Simple Way to Test your Django Models
pytest-django-ordering	A pytest plugin for preserving the order in which Django runs tests.
pytest-django-queries	Generate performance reports from your django database performance tests.
pytest-djangorestframework	A djangorestframework plugin for pytest
pytest-django-rq	A pytest plugin to help writing unit test for django-rq
pytest-django-sqlcounts	py.test plugin for reporting the number of SQLs executed per django testcase.
pytest-django-testing-postgresql	Use a temporary PostgreSQL database with pytest-django
pytest-doc	A documentation plugin for py.test.
pytest-docgen	An RST Documentation Generator for pytest-based test suites
pytest-docker	Simple pytest fixtures for Docker and docker-compose based tests
pytest-docker-butla	
pytest-dockerc	Run, manage and stop Docker Compose project from Docker API
pytest-docker-compose	Manages Docker containers during your integration tests
pytest-docker-db	A plugin to use docker databases for pytests
pytest-docker-fixtures	pytest docker fixtures
pytest-docker-git-fixtures	Pytest fixtures for testing with git scm.
pytest-docker-pexpect	pytest plugin for writing functional tests with pexpect and docker
pytest-docker-postgresql	A simple plugin to use with pytest
pytest-docker-py	Easy to use, simple to extend, pytest plugin that minimally leverages docker-py.
pytest-docker-registry-fixtures	Pytest fixtures for testing with docker registries.
pytest-docker-tools	Docker integration tests for pytest
pytest-docs	Documentation tool for pytest
pytest-docstyle	pytest plugin to run pydocstyle
pytest-doctest-custom	A py.test plugin for customizing string representations of doctest results.
pytest-doctest-ellipsis-markers	Setup additional values for ELLIPSIS_MARKER for doctests
pytest-doctest-import	A simple pytest plugin to import names and add them to the doctest namespace.
pytest-doctestplus	Pytest plugin with advanced doctest features.
pytest-doctest-ufunc	A plugin to run doctests in docstrings of Numpy ufuncs
pytest-dolphin	Some extra stuff that we use ininternally
pytest-doorstop	A pytest plugin for adding test results into doorstop items.
pytest-dotenv	A py.test plugin that parses environment files before running tests
pytest-drf	A Django REST framework plugin for pytest.
pytest-drivings	Tool to allow webdriver automation to be ran locally or remotely
pytest-drop-dup-tests	A Pytest plugin to drop duplicated tests during collection
pytest-dummynt	A py.test plugin providing access to a dummynt.
pytest-dump2json	A pytest plugin for dumping test results to json.
pytest-duration-insights	
pytest-dynamicrerun	A pytest plugin to rerun tests dynamically based off of test outcome and output.
pytest-dynamodb	DynamoDB fixtures for pytest
pytest-easy-addoption	pytest-easy-addoption: Easy way to work with pytest addoption
pytest-easy-api	Simple API testing with pytest
pytest-easyMPI	Package that supports mpi tests in pytest
pytest-easyread	pytest plugin that makes terminal printouts of the reports easier to read
pytest-easy-server	Pytest plugin for easy testing against servers

name	summary
pytest-ec2	Pytest execution on EC2 instance
pytest-echo	pytest plugin with mechanisms for echoing environment variables, package version and gener
pytest-elasticsearch	Elasticsearch fixtures and fixture factories for Pytest.
pytest-elements	Tool to help automate user interfaces
pytest-elk-reporter	A simple plugin to use with pytest
pytest-email	Send execution result email
pytest-embedded	pytest embedded plugin
pytest-embedded-idf	pytest embedded plugin for esp-idf project
pytest-embedded-jtag	pytest embedded plugin for testing with jtag
pytest-embedded-qemu	pytest embedded plugin for qemu, not target chip
pytest-embedded-qemu-idf	pytest embedded plugin for esp-idf project by qemu, not target chip
pytest-embedded-serial	pytest embedded plugin for testing serial ports
pytest-embedded-serial-esp	pytest embedded plugin for testing espressif boards via serial ports
pytest-emoji	A pytest plugin that adds emojis to your test result report
pytest-emoji-output	Pytest plugin to represent test output with emoji support
pytest-enabler	Enable installed pytest plugins
pytest-enhancements	Improvements for pytest (rejected upstream)
pytest-env	py.test plugin that allows you to add environment variables.
pytest-envfiles	A py.test plugin that parses environment files before running tests
pytest-env-info	Push information about the running pytest into envvars
pytest-envraw	py.test plugin that allows you to add environment variables.
pytest-envvars	Pytest plugin to validate use of envvars on your tests
pytest-env-yaml	
pytest-eradicate	pytest plugin to check for commented out code
pytest-error-for-skips	Pytest plugin to treat skipped tests a test failure
pytest-eth	PyTest plugin for testing Smart Contracts for Ethereum Virtual Machine (EVM).
pytest-ethereum	pytest-ethereum: Pytest library for ethereum projects.
pytest-eucalyptus	Pytest Plugin for BDD
pytest-eventlet	Applies eventlet monkey-patch as a pytest plugin.
pytest-excel	pytest plugin for generating excel reports
pytest-exceptional	Better exceptions
pytest-exception-script	Walk your code through exception script to check it's resiliency to failures.
pytest-executable	pytest plugin for testing executables
pytest-expect	py.test plugin to store test expectations and mark tests based on them
pytest-expecter	Better testing with expecter and pytest.
pytest-expectr	This plugin is used to expect multiple assert using pytest framework.
pytest-explicit	A Pytest plugin to ignore certain marked tests by default
pytest-exploratory	Interactive console for pytest.
pytest-external-blockers	a special outcome for tests that are blocked for external reasons
pytest-extra-durations	A pytest plugin to get durations on a per-function basis and per module basis.
pytest-fabric	Provides test utilities to run fabric task tests by using docker containers
pytest-factory	Use factories for test setup with py.test
pytest-factoryboy	Factory Boy support for pytest.
pytest-factoryboy-fixtures	Generates pytest fixtures that allow the use of type hinting
pytest-factoryboy-state	Simple factoryboy random state management
pytest-failed-screenshot	Test case fails,take a screenshot,save it,attach it to the allure
pytest-failed-to-verify	A pytest plugin that helps better distinguishing real test failures from setup flakiness.
pytest-faker	Faker integration with the pytest framework.
pytest-falcon	Pytest helpers for Falcon.

name	summary
pytest-falcon-client	Pytest `client` fixture for the Falcon Framework
pytest-fantasy	Pytest plugin for Flask Fantasy Framework
pytest-fastapi	
pytest-fastest	Use SCM and coverage to run only needed tests
pytest-fast-first	Pytest plugin that runs fast tests first
pytest-faulthandler	py.test plugin that activates the fault handler module for tests (dummy package)
pytest-fauxfactory	Integration of fauxfactory into pytest.
pytest-figleaf	py.test figleaf coverage plugin
pytest-filecov	A pytest plugin to detect unused files
pytest-filedata	easily load data from files
pytest-filemarker	A pytest plugin that runs marked tests when files change.
pytest-filter-case	run test cases filter by mark
pytest-filter-subpackage	Pytest plugin for filtering based on sub-packages
pytest-find-dependencies	A pytest plugin to find dependencies between tests
pytest-finer-verdicts	A pytest plugin to treat non-assertion failures as test errors.
pytest-firefox	pytest plugin to manipulate firefox
pytest-fixture-config	Fixture configuration utils for py.test
pytest-fixture-maker	Pytest plugin to load fixtures from YAML files
pytest-fixture-marker	A pytest plugin to add markers based on fixtures used.
pytest-fixture-order	pytest plugin to control fixture evaluation order
pytest-fixtures	Common fixtures for pytest
pytest-fixture-tools	Plugin for pytest which provides tools for fixtures
pytest-fixture-typecheck	A pytest plugin to assert type annotations at runtime.
pytest-flake8	pytest plugin to check FLAKE8 requirements
pytest-flake8-path	A pytest fixture for testing flake8 plugins.
pytest-flakefinder	Runs tests multiple times to expose flakiness.
pytest-flakes	pytest plugin to check source code with pyflakes
pytest-flaptastic	Flaptastic py.test plugin
pytest-flask	A set of py.test fixtures to test Flask applications.
pytest-flask-sqlalchemy	A pytest plugin for preserving test isolation in Flask-SQLAlchemy using database transactions
pytest-flask-sqlalchemy-transactions	Run tests in transactions using pytest, Flask, and SQLAlchemy.
pytest-flyte	Pytest fixtures for simplifying Flyte integration testing
pytest-focus	A pytest plugin that alerts user of failed test cases with screen notifications
pytest-forcefail	py.test plugin to make the test failing regardless of pytest.mark.xfail
pytest-forward-compatability	A name to avoid typosquating pytest-foward-compatibility
pytest-forward-compatibility	A pytest plugin to shim pytest commandline options for fowards compatibility
pytest-freezegun	Wrap tests with fixtures in freeze_time
pytest-freeze-reqs	Check if requirement files are frozen
pytest-frozen-uuids	Deterministically frozen UUID's for your tests
pytest-func-cov	Pytest plugin for measuring function coverage
pytest-funparam	An alternative way to parametrize test cases
pytest-fxa	pytest plugin for Firefox Accounts
pytest-fxtest	
pytest-gc	The garbage collector plugin for py.test
pytest-gcov	Uses gcov to measure test coverage of a C library
pytest-gevent	Ensure that gevent is properly patched when invoking pytest
pytest-gherkin	A flexible framework for executing BDD gherkin tests
pytest-ghostinspector	For finding/executing Ghost Inspector tests
pytest-girder	A set of pytest fixtures for testing Girder applications.

name	summary
pytest-git	Git repository fixture for py.test
pytest-gitcov	Pytest plugin for reporting on coverage of the last git commit.
pytest-git-fixtures	Pytest fixtures for testing with git.
pytest-github	Plugin for py.test that associates tests with github issues using a marker.
pytest-github-actions-annotate-failures	pytest plugin to annotate failed tests with a workflow command for GitHub Actions
pytest-gitignore	py.test plugin to ignore the same files as git
pytest-gnupg-fixtures	Pytest fixtures for testing with gnupg.
pytest-golden	Plugin for pytest that offloads expected outputs to data files
pytest-graphql-schema	Get graphql schema as fixture for pytest
pytest-greendots	Green progress dots
pytest-growl	Growl notifications for pytest results.
pytest-grpc	pytest plugin for grpc
pytest-hammertime	Display “↘” instead of “.” for passed pytest tests.
pytest-harvest	Store data created during your pytest tests execution, and retrieve it at the end of the session,
pytest-helm-chart	A plugin to provide different types and configs of Kubernetes clusters that can be used for tes
pytest-helm-charts	A plugin to provide different types and configs of Kubernetes clusters that can be used for tes
pytest-helper	Functions to help in using the pytest testing framework
pytest-helpers	pytest helpers
pytest-helpers-namespace	Pytest Helpers Namespace Plugin
pytest-hidecaptured	Hide captured output
pytest-historic	Custom report to display pytest historical execution records
pytest-historic-hook	Custom listener to store execution results into MYSQL DB, which is used for pytest-historic
pytest-homeassistant	A pytest plugin for use with homeassistant custom components.
pytest-homeassistant-custom-component	Experimental package to automatically extract test plugins for Home Assistant custom compo
pytest-honors	Report on tests that honor constraints, and guard against regressions
pytest-hoverfly	Simplify working with Hoverfly from pytest
pytest-hoverfly-wrapper	Integrates the Hoverfly HTTP proxy into Pytest
pytest-hpfeeds	Helpers for testing hpfeeds in your python project
pytest-html	pytest plugin for generating HTML reports
pytest-html-lee	optimized pytest plugin for generating HTML reports
pytest-html-profiling	Pytest plugin for generating HTML reports with per-test profiling and optionally call graph vi
pytest-html-reporter	Generates a static html report based on pytest framework
pytest-html-thread	pytest plugin for generating HTML reports
pytest-http	Fixture “http” for http requests
pytest-httpbin	Easily test your HTTP library against a local copy of httpbin
pytest-http-mocker	Pytest plugin for http mocking (via https://github.com/vilus/mocker)
pytest-httpretty	A thin wrapper of HTTPretty for pytest
pytest-httpserver	pytest-httpserver is a httpserver for pytest
pytest-httpx	Send responses to httpx.
pytest-httpx-blockage	Disable httpx requests during a test run
pytest-hue	Visualise PyTest status via your Phillips Hue lights
pytest-hylang	Pytest plugin to allow running tests written in hylang
pytest-hypo-25	help hypo module for pytest
pytest-ibutsu	A plugin to sent pytest results to an Ibutsu server
pytest-icdiff	use icdiff for better error messages in pytest assertions
pytest-idapro	A pytest plugin for idapython. Allows a pytest setup to run tests outside and inside IDA in an
pytest-ignore-flaky	ignore failures from flaky tests (pytest plugin)
pytest-image-diff	
pytest-incremental	an incremental test runner (pytest plugin)

name	summary
pytest-influxdb	Plugin for influxdb and pytest integration.
pytest-info-collector	pytest plugin to collect information from tests
pytest-informative-node	display more node information.
pytest-infrastructure	pytest stack validation prior to testing executing
pytest-ini	Reuse pytest.ini to store env variables
pytest-inmanta	A py.test plugin providing fixtures to simplify inmanta modules testing.
pytest-inmanta-extensions	Inmanta tests package
pytest-Inomaly	A simple image diff plugin for pytest
pytest-insta	A practical snapshot testing plugin for pytest
pytest-instafail	pytest plugin to show failures instantly
pytest-instrument	pytest plugin to instrument tests
pytest-integration	Organizing pytests by integration or not
pytest-integration-mark	Automatic integration test marking and excluding plugin for pytest
pytest-interactive	A pytest plugin for console based interactive test selection just after the collection phase
pytest-intercept-remote	Pytest plugin for intercepting outgoing connection requests during pytest run.
pytest-invenio	Pytest fixtures for Invenio.
pytest-involve	Run tests covering a specific file or changeset
pytest-ipdb	A py.test plug-in to enable drop to ipdb debugger on test failure.
pytest-ipynb	THIS PROJECT IS ABANDONED
pytest-isort	py.test plugin to check import ordering using isort
pytest-it	Pytest plugin to display test reports as a plaintext spec, inspired by Rspec: https://github.com
pytest-iterassert	Nicer list and iterable assertion messages for pytest
pytest-jasmine	Run jasmine tests from your pytest test suite
pytest-jest	A custom jest-pytest oriented Pytest reporter
pytest-jira	py.test JIRA integration plugin, using markers
pytest-jira-xray	pytest plugin to integrate tests with JIRA XRAY
pytest-jobserver	Limit parallel tests with posix jobserver.
pytest-joke	Test failures are better served with humor.
pytest-json	Generate JSON test reports
pytest-jsonlint	UNKNOWN
pytest-json-report	A pytest plugin to report test results as JSON files
pytest-kafka	Zookeeper, Kafka server, and Kafka consumer fixtures for Pytest
pytest-kafkavents	A plugin to send pytest events to Kafka
pytest-kind	Kubernetes test support with KIND for pytest
pytest-kivy	Kivy GUI tests fixtures using pytest
pytest-knows	A pytest plugin that can automatically skip test case based on dependence info calculated by tra
pytest-konira	Run Konira DSL tests with py.test
pytest-krtech-common	pytest krtech common library
pytest-kwparametrize	Alternate syntax for @pytest.mark.parametrize with test cases as dictionaries and default valu
pytest-lambda	Define pytest fixtures with lambda functions.
pytest-lamp	
pytest-layab	Pytest fixtures for layab.
pytest-lazy-fixture	It helps to use fixtures in pytest.mark.parametrize
pytest-ldap	python-ldap fixtures for pytest
pytest-leaks	A pytest plugin to trace resource leaks.
pytest-level	Select tests of a given level or lower
pytest-libfaketime	A python-libfaketime plugin for pytest.
pytest-libiio	A pytest plugin to manage interfacing with libiio contexts
pytest-libnotify	Pytest plugin that shows notifications about the test run

name	summary
pytest-ligo	
pytest-lineno	A pytest plugin to show the line numbers of test functions
pytest-line-profiler	Profile code executed by pytest
pytest-lisa	Pytest plugin for organizing tests.
pytest-listener	A simple network listener
pytest-litf	A pytest plugin that stream output in LITF format
pytest-live	Live results for pytest
pytest-localftpserver	A PyTest plugin which provides an FTP fixture for your tests
pytest-localserver	py.test plugin to test server connections locally.
pytest-localstack	Pytest plugin for AWS integration tests
pytest-lockable	lockable resource plugin for pytest
pytest-locker	Used to lock object during testing. Essentially changing assertions from being hard coded to
pytest-log	print log
pytest-logbook	py.test plugin to capture logbook log messages
pytest-logdog	Pytest plugin to test logging
pytest-logfest	Pytest plugin providing three logger fixtures with basic or full writing to log files
pytest-logger	Plugin configuring handlers for loggers from Python logging module.
pytest-logging	Configures logging and allows tweaking the log level with a py.test flag
pytest-log-report	Package for creating a pytest test run report
pytest-manual-marker	pytest marker for marking manual tests
pytest-markdown	Test your markdown docs with pytest
pytest-marker-bugzilla	py.test bugzilla integration plugin, using markers
pytest-markers-presence	A simple plugin to detect missed pytest tags and markers”
pytest-markfiltration	UNKNOWN
pytest-mark-no-py3	pytest plugin and bowler codemod to help migrate tests to Python 3
pytest-marks	UNKNOWN
pytest-matcher	Match test output against patterns stored in files
pytest-match-skip	Skip matching marks. Matches partial marks using wildcards.
pytest-mat-report	this is report
pytest-matrix	Provide tools for generating tests from combinations of fixtures.
pytest-mccabe	pytest plugin to run the mccabe code complexity checker.
pytest-md	Plugin for generating Markdown reports for pytest results
pytest-md-report	A pytest plugin to make a test results report with Markdown table format.
pytest-memprof	Estimates memory consumption of test functions
pytest-menu	A pytest plugin for console based interactive test selection just after the collection phase
pytest-mercurial	pytest plugin to write integration tests for projects using Mercurial Python internals
pytest-messenger	Pytest to Slack reporting plugin
pytest-metadata	pytest plugin for test session metadata
pytest-metrics	Custom metrics report for pytest
pytest-mimesis	Mimesis integration with the pytest test runner
pytest-minecraft	A pytest plugin for running tests against Minecraft releases
pytest-missing-fixtures	Pytest plugin that creates missing fixtures
pytest-ml	Test your machine learning!
pytest-mocha	pytest plugin to display test execution output like a mochajs
pytest-mock	Thin-wrapper around the mock package for easier use with pytest
pytest-mock-api	A mock API server with configurable routes and responses available as a fixture.
pytest-mock-generator	A pytest fixture wrapper for https://pypi.org/project/mock-generator
pytest-mock-helper	Help you mock HTTP call and generate mock code
pytest-mockito	Base fixtures for mockito

name	summary
pytest-mockredis	An in-memory mock of a Redis server that runs in a separate thread. This is to be used for u
pytest-mock-resources	A pytest plugin for easily instantiating reproducible mock resources.
pytest-mock-server	Mock server plugin for pytest
pytest-mockservers	A set of fixtures to test your requests to HTTP/UDP servers
pytest-modifyjunit	Utility for adding additional properties to junit xml for IDM QE
pytest-modifyscope	pytest plugin to modify fixture scope
pytest-molecule	PyTest Molecule Plugin :: discover and run molecule tests
pytest-mongo	MongoDB process and client fixtures plugin for Pytest.
pytest-mongodb	pytest plugin for MongoDB fixtures
pytest-monitor	Pytest plugin for analyzing resource usage.
pytest-monkeyplus	pytest's monkeypatch subclass with extra functionalities
pytest-monkeytype	pytest-monkeytype: Generate Monkeytype annotations from your pytest tests.
pytest-moto	Fixtures for integration tests of AWS services,uses moto mocking library.
pytest-motor	A pytest plugin for motor, the non-blocking MongoDB driver.
pytest-mp	A test batcher for multiprocessed Pytest runs
pytest-mpi	pytest plugin to collect information from tests
pytest-mpl	pytest plugin to help with testing figures output from Matplotlib
pytest-mproc	low-startup-overhead, scalable, distributed-testing pytest plugin
pytest-multi-check	Pytest-плагин, реализует возможность мульти проверок и мягких проверок
pytest-multihost	Utility for writing multi-host tests for pytest
pytest-multilog	Multi-process logs handling and other helpers for pytest
pytest-multithreading	a pytest plugin for th and concurrent testing
pytest-mutagen	Add the mutation testing feature to pytest
pytest-mypy	Мypy static type checker plugin for Pytest
pytest-mypyd	Мypy static type checker plugin for Pytest
pytest-mypy-plugins	pytest plugin for writing tests for mypy plugins
pytest-mypy-plugins-shim	Substitute for "pytest-mypy-plugins" for Python implementations which aren't supported by m
pytest-mypy-testing	Pytest plugin to check mypy output.
pytest-mysql	MySQL process and client fixtures for pytest
pytest-needle	pytest plugin for visual testing websites using selenium
pytest-neo	pytest-neo is a plugin for pytest that shows tests like screen of Matrix.
pytest-network	A simple plugin to disable network on socket level.
pytest-never-sleep	pytest plugin helps to avoid adding tests without mock `time.sleep`
pytest-nginx	nginx fixture for pytest
pytest-nginx-iplweb	nginx fixture for pytest - iplweb temporary fork
pytest-ngrok	
pytest-ngsfixtures	pytest ngs fixtures
pytest-nice	A pytest plugin that alerts user of failed test cases with screen notifications
pytest-nice-parametrize	A small snippet for nicer PyTest's Parametrize
pytest-nlcov	Pytest plugin to get the coverage of the new lines (based on git diff) only
pytest-nocustom	Run all tests without custom markers
pytest-nodenv	Test-driven source code search for Python.
pytest-nogarbage	Ensure a test produces no garbage
pytest-notebook	A pytest plugin for testing Jupyter Notebooks
pytest-notice	Send pytest execution result email
pytest-notification	A pytest plugin for sending a desktop notification and playing a sound upon completion of tes
pytest-notifier	A pytest plugin to notify test result
pytest-notimplemented	Pytest markers for not implemented features and tests.
pytest-notion	A PyTest Reporter to send test runs to Notion.so

name	summary
pytest-nunit	A pytest plugin for generating NUnit3 test result XML output
pytest-ochrus	pytest results data-base and HTML reporter
pytest-odoo	py.test plugin to run Odoo tests
pytest-odoo-fixtures	Project description
pytest-oerp	pytest plugin to test OpenERP modules
pytest-ok	The ultimate pytest output plugin
pytest-only	Use @pytest.mark.only to run a single test
pytest-oot	Run object-oriented tests in a simple format
pytest-openfiles	Pytest plugin for detecting inadvertent open file handles
pytest-opentmi	pytest plugin for publish results to opentmi
pytest-operator	Fixtures for Operators
pytest-optional	include/exclude values of fixtures in pytest
pytest-optional-tests	Easy declaration of optional tests (i.e., that are not run by default)
pytest-orchestration	A pytest plugin for orchestrating tests
pytest-order	pytest plugin to run your tests in a specific order
pytest-ordering	pytest plugin to run your tests in a specific order
pytest-osxnotify	OS X notifications for py.test results.
pytest-pact	A simple plugin to use with pytest
pytest-parallel	a pytest plugin for parallel and concurrent testing
pytest-parallel-39	a pytest plugin for parallel and concurrent testing
pytest-param	pytest plugin to test all, first, last or random params
pytest-paramark	Configure pytest fixtures using a combination of "parametrize" and markers
pytest-parametrization	Simpler PyTest parametrization
pytest-parametrize-cases	A more user-friendly way to write parametrized tests.
pytest-parametrized	Pytest plugin for parametrizing tests with default iterables.
pytest-parawtf	Finally spell paramete?ri[sz]e correctly
pytest-pass	Check out https://github.com/elilutsky/pytest-pass
pytest-passrunner	Pytest plugin providing the 'run_on_pass' marker
pytest-paste-config	Allow setting the path to a paste config file
pytest-patches	A contextmanager pytest fixture for handling multiple mock patches
pytest-pdb	pytest plugin which adds pdb helper commands related to pytest.
pytest-peach	pytest plugin for fuzzing with Peach API Security
pytest-pep257	py.test plugin for pep257
pytest-pep8	pytest plugin to check PEP8 requirements
pytest-percent	Change the exit code of pytest test sessions when a required percent of tests pass.
pytest-perf	pytest-perf
pytest-performance	A simple plugin to ensure the execution of critical sections of code has not been impacted
pytest-persistence	Pytest tool for persistent objects
pytest-pgsql	Pytest plugins and helpers for tests using a Postgres database.
pytest-picked	Run the tests related to the changed files
pytest-pigeonhole	
pytest-pikachu	Show surprise when tests are passing
pytest-pilot	Slice in your test base thanks to powerful markers.
pytest-pings	🦊 The pytest plugin for Firefox Telemetry 🏠
pytest-pinned	A simple pytest plugin for pinning tests
pytest-pinpoint	A pytest plugin which runs SBFL algorithms to detect faults.
pytest-pipeline	Pytest plugin for functional testing of data analysispipelines
pytest-platform-markers	Markers for pytest to skip tests on specific platforms
pytest-play	pytest plugin that let you automate actions and assertions with test metrics reporting executing

name	summary
pytest-playbook	Pytest plugin for reading playbooks.
pytest-playwright	A pytest wrapper with fixtures for Playwright to automate web browsers
pytest-playwright-snapshot	A pytest wrapper for snapshot testing with playwright
pytest-plt	Fixtures for quickly making Matplotlib plots in tests
pytest-plugin-helpers	A plugin to help developing and testing other plugins
pytest-plus	PyTest Plus Plugin :: extends pytest functionality
pytest-pmisc	
pytest-pointers	Pytest plugin to define functions you test with special marks for better navigation and reports
pytest-polarion-cfme	pytest plugin for collecting test cases and recording test results
pytest-polarion-collect	pytest plugin for collecting polarion test cases data
pytest-polecat	Provides Polecat pytest fixtures
pytest-ponyorm	PonyORM in Pytest
pytest-poo	Visualize your crappy tests
pytest-poo-fail	Visualize your failed tests with poo
pytest-pop	A pytest plugin to help with testing pop projects
pytest-portion	Select a portion of the collected tests
pytest-postgres	Run PostgreSQL in Docker container in Pytest.
pytest-postgresql	Postgresql fixtures and fixture factories for Pytest.
pytest-power	pytest plugin with powerful fixtures
pytest-pride	Minitest-style test colors
pytest-print	pytest-print adds the printer fixture you can use to print messages to the user (directly to the p
pytest-profiling	Profiling plugin for py.test
pytest-progress	pytest plugin for instant test progress status
pytest-prometheus	Report test pass / failures to a Prometheus PushGateway
pytest-prosper	Test helpers for Prosper projects
pytest-pspec	A rspec format reporter for Python ptest
pytest-psqlgraph	pytest plugin for testing applications that use psqlgraph
pytest-ptera	Use ptera probes in tests
pytest-pudb	Pytest PuDB debugger integration
pytest-purkinje	py.test plugin for purkinje test runner
pytest-pycharm	Plugin for py.test to enter PyCharm debugger on uncaught exceptions
pytest-pycodestyle	pytest plugin to run pycodestyle
pytest-pydev	py.test plugin to connect to a remote debug server with PyDev or PyCharm.
pytest-pydocstyle	pytest plugin to run pydocstyle
pytest-pylint	pytest plugin to check source code with pylint
pytest-pypi	Easily test your HTTP library against a local copy of pypi
pytest-pypom-navigation	Core engine for cookiecutter-qa and pytest-play packages
pytest-pypeteer	A plugin to run pypeteer in pytest.
pytest-pyq	Pytest fixture “q” for pyq
pytest-pyramid	pytest_pyramid - provides fixtures for testing pyramid applications with pytest test suite
pytest-pyramid-server	Pyramid server fixture for py.test
pytest-pyright	Pytest plugin for type checking code with Pyright
pytest-pytestrail	Pytest plugin for interaction with TestRail
pytest-pythonpath	pytest plugin for adding to the PYTHONPATH from command line or configs.
pytest-pytorch	pytest plugin for a better developer experience when working with the PyTorch test suite
pytest-qasync	Pytest support for qasync.
pytest-qatouch	Pytest plugin for uploading test results to your QA Touch Testrun.
pytest-qgis	A pytest plugin for testing QGIS python plugins
pytest-qml	Run QML Tests with pytest

name	summary
pytest-qt	pytest support for PyQt and PySide applications
pytest-qt-app	QT app fixture for py.test
pytest-quarantine	A plugin for pytest to manage expected test failures
pytest-quickcheck	pytest plugin to generate random data inspired by QuickCheck
pytest-rabbitmq	RabbitMQ process and client fixtures for pytest
pytest-race	Race conditions tester for pytest
pytest-rage	pytest plugin to implement PEP712
pytest-railflow-testrail-reporter	Generate json reports along with specified metadata defined in test markers.
pytest-raises	An implementation of pytest.raises as a pytest.mark fixture
pytest-raisesregex	Simple pytest plugin to look for regex in Exceptions
pytest-raisin	Plugin enabling the use of exception instances with pytest.raises
pytest-random	py.test plugin to randomize tests
pytest-randomly	Pytest plugin to randomly order tests and control random.seed.
pytest-randomness	Pytest plugin about random seed management
pytest-random-num	Randomise the order in which pytest tests are run with some control over the randomness
pytest-random-order	Randomise the order in which pytest tests are run with some control over the randomness
pytest-readme	Test your README.md file
pytest-reana	Pytest fixtures for REANA.
pytest-recording	A pytest plugin that allows you recording of network interactions via VCR.py
pytest-recordings	Provides pytest plugins for reporting request/response traffic, screenshots, and more to Report
pytest-redis	Redis fixtures and fixture factories for Pytest.
pytest-redislite	Pytest plugin for testing code using Redis
pytest-redmine	Pytest plugin for redmine
pytest-ref	A plugin to store reference files to ease regression testing
pytest-reference-formatter	Conveniently run pytest with a dot-formatted test reference.
pytest-regressions	Easy to use fixtures to write regression tests.
pytest-regtest	pytest plugin for regression tests
pytest-relative-order	a pytest plugin that sorts tests using “before” and “after” markers
pytest-relaxed	Relaxed test discovery/organization for pytest
pytest-remfiles	Pytest plugin to create a temporary directory with remote files
pytest-remotedata	Pytest plugin for controlling remote data access.
pytest-remote-response	Pytest plugin for capturing and mocking connection requests.
pytest-remove-stale-bytecode	py.test plugin to remove stale byte code files.
pytest-reorder	Reorder tests depending on their paths and names.
pytest-repeat	pytest plugin for repeating tests
pytest-replay	Saves previous test runs and allow re-execute previous pytest runs to reproduce crashes or flak
pytest-repo-health	A pytest plugin to report on repository standards conformance
pytest-report	Creates json report that is compatible with atom.io’s linter message format
pytest-reporter	Generate Pytest reports with templates
pytest-reporter-html1	A basic HTML report template for Pytest
pytest-reportinfra	Pytest plugin for reportinfra
pytest-reporting	A plugin to report summarized results in a table format
pytest-reportlog	Replacement for the -resultlog option, focused in simplicity and extensibility
pytest-report-me	A pytest plugin to generate report.
pytest-report-parameters	pytest plugin for adding tests’ parameters to junit report
pytest-reportportal	Agent for Reporting results of tests to the Report Portal
pytest-reqs	pytest plugin to check pinned requirements
pytest-requests	A simple plugin to use with pytest
pytest-reraise	Make multi-threaded pytest test cases fail when they should

name	summary
pytest-rerun	Re-run only changed files in specified branch
pytest-rerunfailures	pytest plugin to re-run tests to eliminate flaky failures
pytest-resilient-circuits	Resilient Circuits fixtures for PyTest.
pytest-resource	Load resource fixture plugin to use with pytest
pytest-resource-path	Provides path for uniform access to test resources in isolated directory
pytest-responsemock	Simplified requests calls mocking for pytest
pytest-responses	py.test integration for responses
pytest-restrict	Pytest plugin to restrict the test types allowed
pytest-rethinkdb	A RethinkDB plugin for pytest.
pytest-reverse	Pytest plugin to reverse test order.
pytest-ringo	pytest plugin to test webapplications using the Ringo webframework
pytest-rng	Fixtures for seeding tests and making randomness reproducible
pytest-roast	pytest plugin for ROAST configuration override and fixtures
pytest-rocketchat	Pytest to Rocket.Chat reporting plugin
pytest-rottest	Pytest integration with rottest
pytest-rpc	Extend py.test for RPC OpenStack testing.
pytest-rst	Test code from RST documents with pytest
pytest-rt	pytest data collector plugin for Testgr
pytest-rts	Coverage-based regression test selection (RTS) plugin for pytest
pytest-run-changed	Pytest plugin that runs changed tests only
pytest-runfailed	implement a --failed option for pytest
pytest-runner	Invoke py.test as distutils command with dependency resolution
pytest-runtime-xfail	Call runtime_xfail() to mark running test as xfail.
pytest-salt	Pytest Salt Plugin
pytest-salt-containers	A Pytest plugin that builds and creates docker containers
pytest-salt-factories	Pytest Salt Plugin
pytest-salt-from-filenames	Simple PyTest Plugin For Salt's Test Suite Specifically
pytest-salt-runttests-bridge	Simple PyTest Plugin For Salt's Test Suite Specifically
pytest-sanic	a pytest plugin for Sanic
pytest-sanity	
pytest-sa-pg	
pytest-sbase	A complete web automation framework for end-to-end testing.
pytest-scenario	pytest plugin for test scenarios
pytest-schema	🔗 Validate return values against a schema-like object in testing
pytest-securestore	An encrypted password store for use within pytest cases
pytest-select	A pytest plugin which allows to (de-)select tests from a file.
pytest-selenium	pytest plugin for Selenium
pytest-seleniumbase	A complete web automation framework for end-to-end testing.
pytest-selenium-enhancer	pytest plugin for Selenium
pytest-selenium-pdiff	A pytest package implementing perceptualdiff for Selenium tests.
pytest-send-email	Send pytest execution result email
pytest-sentry	A pytest plugin to send testrun information to Sentry.io
pytest-server-fixtures	Extensible server fixtures for py.test
pytest-serverless	Automatically mocks resources from serverless.yml in pytest using moto.
pytest-services	Services plugin for pytest testing framework
pytest-session2file	pytest-session2file (aka: pytest-session_to_file for v0.1.0 - v0.1.2) is a py.test plugin for capturing
pytest-session-fixture-globalize	py.test plugin to make session fixtures behave as if written in conftest, even if it is written in s
pytest-session_to_file	pytest-session_to_file is a py.test plugin for capturing and saving to file the stdout of py.test.
pytest-sftpsserver	py.test plugin to locally test sftp server connections.

name	summary
pytest-shard	
pytest-shell	A pytest plugin for testing shell scripts and line-based processes
pytest-sheraf	Versatile ZODB abstraction layer - pytest fixtures
pytest-sherlock	pytest plugin help to find coupled tests
pytest-shortcuts	Expand command-line shortcuts listed in pytest configuration
pytest-shutil	A goodie-bag of unix shell and environment tools for py.test
pytest-simplehttpserver	Simple pytest fixture to spin up an HTTP server
pytest-simple-plugin	Simple pytest plugin
pytest-simple-settings	simple-settings plugin for pytest
pytest-single-file-logging	Allow for multiple processes to log to a single file
pytest-skip-markers	Pytest Salt Plugin
pytest-skipper	A plugin that selects only tests with changes in execution path
pytest-skippy	Automatically skip tests that don't need to run!
pytest-skip-slow	A pytest plugin to skip <code>@pytest.mark.slow</code> tests by default.
pytest-slack	Pytest to Slack reporting plugin
pytest-slow	A pytest plugin to skip <code>@pytest.mark.slow</code> tests by default.
pytest-smartcollect	A plugin for collecting tests that touch changed code
pytest-smartcov	Smart coverage plugin for pytest.
pytest-smtp	Send email with pytest execution result
pytest-snail	Plugin for adding a marker to slow running tests. 🐌
pytest-snapci	py.test plugin for Snap-CI
pytest-snapshot	A plugin for snapshot testing with pytest.
pytest-snmpserver	
pytest-socket	Pytest Plugin to disable socket calls during tests
pytest-soft-assertions	
pytest-solr	Solr process and client fixtures for py.test.
pytest-sorter	A simple plugin to first execute tests that historically failed more
pytest-sourceorder	Test-ordering plugin for pytest
pytest-spark	pytest plugin to run the tests with support of pyspark.
pytest-spawner	py.test plugin to spawn process and communicate with them.
pytest-spec	Library pytest-spec is a pytest plugin to display test execution output like a SPECIFICATION
pytest-sphinx	Doctest plugin for pytest with support for Sphinx-specific doctest-directives
pytest-spiratest	Exports unit tests as test runs in SpiraTest/Team/Plan
pytest-splinter	Splinter plugin for pytest testing framework
pytest-split	Pytest plugin for splitting test suite based on test execution time
pytest-splitio	Split.io SDK integration for e2e tests
pytest-split-tests	A Pytest plugin for running a subset of your tests by splitting them in to equally sized groups.
pytest-split-tests-tresorit	
pytest-splunk-addon	A Dynamic test tool for Splunk Apps and Add-ons
pytest-splunk-addon-ui-smartx	Library to support testing Splunk Add-on UX
pytest-splunk-env	pytest fixtures for interaction with Splunk Enterprise and Splunk Cloud
pytest-sqitch	sqitch for pytest
pytest-sqlalchemy	pytest plugin with sqlalchemy related fixtures
pytest-sql-bigquery	Yet another SQL-testing framework for BigQuery provided by pytest plugin
pytest-srcpaths	Add paths to sys.path
pytest-ssh	pytest plugin for ssh command run
pytest-start-from	Start pytest run from a given point
pytest-statsd	pytest plugin for reporting to graphite
pytest-stepfunctions	A small description

name	summary
pytest-steps	Create step-wise / incremental tests in pytest.
pytest-stepwise	Run a test suite one failing test at a time.
pytest-stoq	A plugin to pytest stoq
pytest-stress	A Pytest plugin that allows you to loop tests for a user defined amount of time.
pytest-structlog	Structured logging assertions
pytest-structmpd	provide structured temporary directory
pytest-stub	Stub packages, modules and attributes.
pytest-stubprocess	Provide stub implementations for subprocesses in Python tests
pytest-study	A pytest plugin to organize long run tests (named studies) without interfering the regular tests
pytest-subprocess	A plugin to fake subprocess for pytest
pytest-subtesthack	A hack to explicitly set up and tear down fixtures.
pytest-subtests	unittest subTest() support and subtests fixture
pytest-subunit	pytest-subunit is a plugin for py.test which outputs testresult in subunit format.
pytest-sugar	pytest-sugar is a plugin for pytest that changes the default look and feel of pytest (e.g. progress)
pytest-sugar-bugfix159	Workaround for https://github.com/Frozenball/pytest-sugar/issues/159
pytest-super-check	Pytest plugin to check your TestCase classes call super in setUp, tearDown, etc.
pytest-svn	SVN repository fixture for py.test
pytest-symbols	pytest-symbols is a pytest plugin that adds support for passing test environment symbols into p
pytest-taketest	Fixtures for ansible, testinfra and molecule
pytest-tap	Test Anything Protocol (TAP) reporting plugin for pytest
pytest-tape	easy assertion with expected results saved to yaml files
pytest-target	Pytest plugin for remote target orchestration.
pytest-tblineinfo	tblineinfo is a py.test plugin that insert the node id in the final py.test report when -tb=line op
pytest-teamcity-logblock	py.test plugin to introduce block structure in teamcity build log, if output is not captured
pytest-telegram	Pytest to Telegram reporting plugin
pytest-tempdir	Predictable and repeatable tempdir support.
pytest-terraform	A pytest plugin for using terraform fixtures
pytest-terraform-fixture	generate terraform resources to use with pytest
pytest-testbook	A plugin to run tests written in Jupyter notebook
pytest-testconfig	Test configuration plugin for pytest.
pytest-testdirectory	A py.test plugin providing temporary directories in unit tests.
pytest-testdox	A testdox format reporter for pytest
pytest-test-groups	A Pytest plugin for running a subset of your tests by splitting them in to equally sized groups.
pytest-testinfra	Test infrastructures
pytest-testlink-adaptor	pytest reporting plugin for testlink
pytest-testmon	selects tests affected by changed files and methods
pytest-testobject	Plugin to use TestObject Suites with Pytest
pytest-testrail	pytest plugin for creating TestRail runs and adding results
pytest-testrail2	A small example package
pytest-testrail-api	Плагин Pytest, для интеграции с TestRail
pytest-testrail-api-client	TestRail Api Python Client
pytest-testrail-appetize	pytest plugin for creating TestRail runs and adding results
pytest-testrail-client	pytest plugin for Testrail
pytest-testrail-e2e	pytest plugin for creating TestRail runs and adding results
pytest-testrail-ns	pytest plugin for creating TestRail runs and adding results
pytest-testrail-plugin	PyTest plugin for TestRail
pytest-testrail-reporter	
pytest-testreport	
pytest-testslide	TestSlide fixture for pytest

name	summary
pytest-test-this	Plugin for py.test to run relevant tests, based on naively checking if a test contains a reference
pytest-tesults	Tesults plugin for pytest
pytest-tezos	pytest-ligo
pytest-thawgun	Pytest plugin for time travel
pytest-threadleak	Detects thread leaks
pytest-tick	Ticking on tests
pytest-timeit	A pytest plugin to time test function runs
pytest-timeout	pytest plugin to abort hanging tests
pytest-timeouts	Linux-only Pytest plugin to control durations of various test case execution phases
pytest-timer	A timer plugin for pytest
pytest-timestamper	Pytest plugin to add a timestamp prefix to the pytest output
pytest-tipsi-django	
pytest-tipsi-testing	Better fixtures management. Various helpers
pytest-tldr	A pytest plugin that limits the output to just the things you need.
pytest-tm4j-reporter	Cloud Jira Test Management (TM4J) PyTest reporter plugin
pytest-tmreport	this is a vue-element ui report for pytest
pytest-todo	A small plugin for the pytest testing framework, marking TODO comments as failure
pytest-tomato	
pytest-toolbelt	This is just a collection of utilities for pytest, but don't really belong in pytest proper.
pytest-toolbox	Numerous useful plugins for pytest.
pytest-tornado	A py.test plugin providing fixtures and markers to simplify testing of asynchronous tornado ap
pytest-tornado5	A py.test plugin providing fixtures and markers to simplify testing of asynchronous tornado ap
pytest-tornado-yen3	A py.test plugin providing fixtures and markers to simplify testing of asynchronous tornado ap
pytest-tornasync	py.test plugin for testing Python 3.5+ Tornado code
pytest-track	
pytest-translations	Test your translation files.
pytest-travis-fold	Folds captured output sections in Travis CI build log
pytest-trello	Plugin for py.test that integrates trello using markers
pytest-trepan	Pytest plugin for trepan debugger.
pytest-trialtemp	py.test plugin for using the same <code>_trial_temp</code> working directory as trial
pytest-trio	Pytest plugin for trio
pytest-tspwplib	A simple plugin to use with tspwplib
pytest-tstcls	Test Class Base
pytest-twisted	A twisted plugin for pytest.
pytest-typhoon-xray	Typhoon HIL plugin for pytest
pytest-tytest	Typhoon HIL plugin for pytest
pytest-ubersmith	Easily mock calls to ubersmith at the <code>`requests`</code> level.
pytest-ui	Text User Interface for running python tests
pytest-unhandled-exception-exit-code	Plugin for py.test set a different exit code on uncaught exceptions
pytest-unittest-filter	A pytest plugin for filtering unittest-based test classes
pytest-unmarked	Run only unmarked tests
pytest-unordered	Test equality of unordered collections in pytest
pytest-upload-report	pytest-upload-report is a plugin for pytest that upload your test report for test results.
pytest-utils	Some helpers for pytest.
pytest-vagrant	A py.test plugin providing access to vagrant.
pytest-valgrind	
pytest-variables	pytest plugin for providing variables to tests/fixtures
pytest-variant	Variant support for Pytest
pytest-vcr	Plugin for managing VCR.py cassettes

name	summary
pytest-vcv-delete-on-fail	A pytest plugin that automates vcrpy cassettes deletion on test failure.
pytest-vcrpandas	Test from HTTP interactions to dataframe processed.
pytest-venv	py.test fixture for creating a virtual environment
pytest-ver	Pytest module with Verification Report
pytest-verbose-parametrize	More descriptive output for parametrized py.test tests
pytest-vimqf	A simple pytest plugin that will shrink pytest output when specified, to fit vim quickfix window
pytest-virtualenv	Virtualenv fixture for py.test
pytest-voluptuous	Pytest plugin for asserting data against voluptuous schema.
pytest-vscodedebug	A pytest plugin to easily enable debugging tests within Visual Studio Code
pytest-vts	pytest plugin for automatic recording of http stubbed tests
pytest-vw	pytest-vw makes your failing test cases succeed under CI tools scrutiny
pytest-vyper	Plugin for the vyper smart contract language.
pytest-wa-e2e-plugin	Pytest plugin for testing whatsapp bots with end to end tests
pytest-watch	Local continuous test runner with pytest and watchdog.
pytest-watcher	Continuously runs pytest on changes in *.py files
pytest-wdl	Pytest plugin for testing WDL workflows.
pytest-webdriver	Selenium webdriver fixture for py.test
pytest-wetest	Welian API Automation test framework pytest plugin
pytest-whirlwind	Testing Tornado.
pytest-wholenodeid	pytest addon for displaying the whole node id for failures
pytest-win32consoletitle	Pytest progress in console title (Win32 only)
pytest-winnotify	Windows tray notifications for py.test results.
pytest-workflow	A pytest plugin for configuring workflow/pipeline tests using YAML files
pytest-xdist	pytest xdist plugin for distributed testing and loop-on-failing modes
pytest-xdist-debug-for-graingert	pytest xdist plugin for distributed testing and loop-on-failing modes
pytest-xdist-forked	forked from pytest-xdist
pytest-xdist-tracker	pytest plugin helps to reproduce failures for particular xdist node
pytest-xfaillist	Maintain a xfaillist in an additional file to avoid merge-conflicts.
pytest-xfiles	Pytest fixtures providing data read from function, module or package related (x)files.
pytest-xlog	Extended logging for test and decorators
pytest-xpara	An extended parametrizing plugin of pytest.
pytest-xprocess	A pytest plugin for managing processes across test runs.
pytest-xray	
pytest-xrayjira	
pytest-xray-server	
pytest-xvfb	A pytest plugin to run Xvfb for tests.
pytest-yaml	This plugin is used to load yaml output to your test using pytest framework.
pytest-yamltree	Create or check file/directory trees described by YAML
pytest-yamlwsgi	Run tests against wsgi apps defined in yaml
pytest-yapf	Run yapf
pytest-yapf3	Validate your Python file format with yapf
pytest-yield	PyTest plugin to run tests concurrently, each `yield` switch context to other one
pytest-yuk	Display tests you are uneasy with, using 🙄/🙃 for pass/fail of tests marked with yuk.
pytest-zafira	A Zafira plugin for pytest
pytest-zap	OWASP ZAP plugin for py.test.
pytest-zebrunner	Pytest connector for Zebrunner reporting
pytest-zigzag	Extend py.test for RPC OpenStack testing.

3.3 Configuration

3.3.1 Command line options and configuration file settings

You can get help on command line options and values in INI-style configurations files by using the general help option:

```
pytest -h # prints options _and_ config file settings
```

This will display command line and configuration file settings which were registered by installed plugins.

3.3.2 Configuration file formats

Many *pytest settings* can be set in a *configuration file*, which by convention resides on the root of your repository or in your tests folder.

A quick example of the configuration files supported by pytest:

pytest.ini

`pytest.ini` files take precedence over other files, even when empty.

```
# pytest.ini
[pytest]
minversion = 6.0
addopts = -ra -q
testpaths =
    tests
    integration
```

pyproject.toml

New in version 6.0.

`pyproject.toml` are considered for configuration when they contain a `tool.pytest.ini_options` table.

```
# pyproject.toml
[tool.pytest.ini_options]
minversion = "6.0"
addopts = "-ra -q"
testpaths = [
    "tests",
    "integration",
]
```

Note: One might wonder why `[tool.pytest.ini_options]` instead of `[tool.pytest]` as is the case with other tools.

The reason is that the pytest team intends to fully utilize the rich TOML data format for configuration in the future, reserving the `[tool.pytest]` table for that. The `ini_options` table is being used, for now, as a bridge between the existing `.ini` configuration system and the future configuration format.

tox.ini

`tox.ini` files are the configuration files of the `tox` project, and can also be used to hold `pytest` configuration if they have a `[pytest]` section.

```
# tox.ini
[pytest]
minversion = 6.0
addopts = -ra -q
testpaths =
    tests
    integration
```

setup.cfg

`setup.cfg` files are general purpose configuration files, used originally by `distutils`, and can also be used to hold `pytest` configuration if they have a `[tool:pytest]` section.

```
# setup.cfg
[tool:pytest]
minversion = 6.0
addopts = -ra -q
testpaths =
    tests
    integration
```

Warning: Usage of `setup.cfg` is not recommended unless for very simple use cases. `.cfg` files use a different parser than `pytest.ini` and `tox.ini` which might cause hard to track down problems. When possible, it is recommended to use the latter files, or `pyproject.toml`, to hold your `pytest` configuration.

3.3.3 Initialization: determining rootdir and configfile

`pytest` determines a `rootdir` for each test run which depends on the command line arguments (specified test files, paths) and on the existence of configuration files. The determined `rootdir` and `configfile` are printed as part of the `pytest` header during startup.

Here's a summary what `pytest` uses `rootdir` for:

- Construct *nodeids* during collection; each test is assigned a unique *nodeid* which is rooted at the `rootdir` and takes into account the full path, class name, function name and parametrization (if any).
- Is used by plugins as a stable location to store project/test run specific information; for example, the internal *cache* plugin creates a `.pytest_cache` subdirectory in `rootdir` to store its cross-test run state.

`rootdir` is **NOT** used to modify `sys.path/PYTHONPATH` or influence how modules are imported. See *pytest import mechanisms and sys.path/PYTHONPATH* for more details.

The `--rootdir=path` command-line option can be used to force a specific directory. Note that contrary to other command-line options, `--rootdir` cannot be used with `addopts` inside `pytest.ini` because the `rootdir` is used to *find* `pytest.ini` already.

Finding the `rootdir`

Here is the algorithm which finds the `rootdir` from `args`:

- If `-c` is passed in the command-line, use that as configuration file, and its directory as `rootdir`.
- Determine the common ancestor directory for the specified `args` that are recognised as paths that exist in the file system. If no such paths are found, the common ancestor directory is set to the current working directory.
- Look for `pytest.ini`, `pyproject.toml`, `tox.ini`, and `setup.cfg` files in the ancestor directory and upwards. If one is matched, it becomes the `configfile` and its directory becomes the `rootdir`.
- If no configuration file was found, look for `setup.py` upwards from the common ancestor directory to determine the `rootdir`.
- If no `setup.py` was found, look for `pytest.ini`, `pyproject.toml`, `tox.ini`, and `setup.cfg` in each of the specified `args` and upwards. If one is matched, it becomes the `configfile` and its directory becomes the `rootdir`.
- If no `configfile` was found and no configuration argument is passed, use the already determined common ancestor as root directory. This allows the use of `pytest` in structures that are not part of a package and don't have any particular configuration file.

If no `args` are given, `pytest` collects test below the current working directory and also starts determining the `rootdir` from there.

Files will only be matched for configuration if:

- `pytest.ini`: will always match and take precedence, even if empty.
- `pyproject.toml`: contains a `[tool.pytest.ini_options]` table.
- `tox.ini`: contains a `[pytest]` section.
- `setup.cfg`: contains a `[tool:pytest]` section.

The files are considered in the order above. Options from multiple `configfiles` candidates are never merged - the first match wins.

The `Config` object (accessible via hooks or through the `pytestconfig` fixture) will subsequently carry these attributes:

- `config.rootpath`: the determined root directory, guaranteed to exist.
- `config.inipath`: the determined `configfile`, may be `None` (it is named `inipath` for historical reasons).

New in version 6.1: The `config.rootpath` and `config.inipath` properties. They are `pathlib.Path` versions of the older `config.rootdir` and `config.inifile`, which have type `py.path.local`, and still exist for backward compatibility.

The `rootdir` is used as a reference directory for constructing test addresses (“nodeids”) and can be used also by plugins for storing per-testrun information.

Example:

```
pytest path/to/testdir path/other/
```

will determine the common ancestor as `path` and then check for configuration files as follows:

```
# first look for pytest.ini files
path/pytest.ini
path/pyproject.toml # must contain a [tool.pytest.ini_options] table to match
path/tox.ini       # must contain [pytest] section to match
```

(continues on next page)

(continued from previous page)

```

path/setup.cfg      # must contain [tool:pytest] section to match
pytest.ini
... # all the way up to the root

# now look for setup.py
path/setup.py
setup.py
... # all the way up to the root

```

Warning: Custom pytest plugin commandline arguments may include a path, as in `pytest --log-output ../../test.log args`. Then `args` is mandatory, otherwise pytest uses the folder of `test.log` for `rootdir` determination (see also [issue 1435](#)). A dot `.` for referencing to the current working directory is also possible.

3.3.4 Builtin configuration file options

For the full list of options consult the [reference documentation](#).

3.3.5 Syntax highlighting theme customization

The syntax highlighting themes used by pytest can be customized using two environment variables:

- `PYTEST_THEME` sets a [pygment style](#) to use.
- `PYTEST_THEME_MODE` sets this style to *light* or *dark*.

3.4 API Reference

This page contains the full reference to pytest's API.

- *Functions*
 - `pytest.approx`
 - `pytest.fail`
 - `pytest.skip`
 - `pytest.importorskip`
 - `pytest.xfail`
 - `pytest.exit`
 - `pytest.main`
 - `pytest.param`
 - `pytest.raises`
 - `pytest.deprecated_call`
 - `pytest.register_assert_rewrite`

- *pytest.warns*
- *pytest.freeze_includes*
- *Marks*
 - *pytest.mark.filterwarnings*
 - *pytest.mark.parametrize*
 - *pytest.mark.skip*
 - *pytest.mark.skipif*
 - *pytest.mark.usefixtures*
 - *pytest.mark.xfail*
 - *pytest.__version__*
 - *pytest.version_tuple*
 - *Custom marks*
- *Fixtures*
 - *@pytest.fixture*
 - *config.cache*
 - *capsys*
 - *capsysbinary*
 - *capfd*
 - *capfdbinary*
 - *doctest_namespace*
 - *request*
 - *pytestconfig*
 - *record_property*
 - *record_testsuite_property*
 - *caplog*
 - *monkeypatch*
 - *pytester*
 - *testdir*
 - *recwarn*
 - *tmp_path*
 - *tmp_path_factory*
 - *tmpdir*
 - *tmpdir_factory*
- *Hooks*
 - *Bootstrapping hooks*
 - *Initialization hooks*

- *Collection hooks*
- *Test running (runtest) hooks*
- *Reporting hooks*
- *Debugging/Interaction hooks*
- *Objects*
 - *CallInfo*
 - *Class*
 - *Collector*
 - *CollectReport*
 - *Config*
 - *ExceptionInfo*
 - *ExitCode*
 - *File*
 - *FixtureDef*
 - *FSCollector*
 - *Function*
 - *FunctionDefinition*
 - *Item*
 - *MarkDecorator*
 - *MarkGenerator*
 - *Mark*
 - *Metafunc*
 - *Module*
 - *Node*
 - *Parser*
 - *OptionGroup*
 - *PytestPluginManager*
 - *Session*
 - *TestReport*
 - *_Result*
 - *Stash*
- *Global Variables*
- *Environment Variables*
- *Exceptions*
- *Warnings*

- [Configuration Options](#)
- [Command-line Flags](#)

3.4.1 Functions

pytest.approx

approx (*expected*, *rel=None*, *abs=None*, *nan_ok=False*)

Assert that two numbers (or two sets of numbers) are equal to each other within some tolerance.

Due to the [Floating Point Arithmetic: Issues and Limitations](#), numbers that we would intuitively expect to be equal are not always so:

```
>>> 0.1 + 0.2 == 0.3
False
```

This problem is commonly encountered when writing tests, e.g. when making sure that floating-point values are what you expect them to be. One way to deal with this problem is to assert that two floating-point numbers are equal to within some appropriate tolerance:

```
>>> abs((0.1 + 0.2) - 0.3) < 1e-6
True
```

However, comparisons like this are tedious to write and difficult to understand. Furthermore, absolute comparisons like the one above are usually discouraged because there's no tolerance that works well for all situations. $1e-6$ is good for numbers around 1, but too small for very big numbers and too big for very small ones. It's better to express the tolerance as a fraction of the expected value, but relative comparisons like that are even more difficult to write correctly and concisely.

The `approx` class performs floating-point comparisons using a syntax that's as intuitive as possible:

```
>>> from pytest import approx
>>> 0.1 + 0.2 == approx(0.3)
True
```

The same syntax also works for sequences of numbers:

```
>>> (0.1 + 0.2, 0.2 + 0.4) == approx((0.3, 0.6))
True
```

Dictionary *values*:

```
>>> {'a': 0.1 + 0.2, 'b': 0.2 + 0.4} == approx({'a': 0.3, 'b': 0.6})
True
```

numpy arrays:

```
>>> import numpy as np
>>> np.array([0.1, 0.2]) + np.array([0.2, 0.4]) == approx(np.array([0.3, 0.6]))
True
```

And for a numpy array against a scalar:

```
>>> import numpy as np
>>> np.array([0.1, 0.2]) + np.array([0.2, 0.1]) == approx(0.3)
True
```

By default, `approx` considers numbers within a relative tolerance of $1e-6$ (i.e. one part in a million) of its expected value to be equal. This treatment would lead to surprising results if the expected value was `0.0`, because nothing but `0.0` itself is relatively close to `0.0`. To handle this case less surprisingly, `approx` also considers numbers within an absolute tolerance of $1e-12$ of its expected value to be equal. Infinity and NaN are special cases. Infinity is only considered equal to itself, regardless of the relative tolerance. NaN is not considered equal to anything by default, but you can make it be equal to itself by setting the `nan_ok` argument to `True`. (This is meant to facilitate comparing arrays that use NaN to mean “no data”).

Both the relative and absolute tolerances can be changed by passing arguments to the `approx` constructor:

```
>>> 1.0001 == approx(1)
False
>>> 1.0001 == approx(1, rel=1e-3)
True
>>> 1.0001 == approx(1, abs=1e-3)
True
```

If you specify `abs` but not `rel`, the comparison will not consider the relative tolerance at all. In other words, two numbers that are within the default relative tolerance of $1e-6$ will still be considered unequal if they exceed the specified absolute tolerance. If you specify both `abs` and `rel`, the numbers will be considered equal if either tolerance is met:

```
>>> 1 + 1e-8 == approx(1)
True
>>> 1 + 1e-8 == approx(1, abs=1e-12)
False
>>> 1 + 1e-8 == approx(1, rel=1e-6, abs=1e-12)
True
```

You can also use `approx` to compare nonnumeric types, or dicts and sequences containing nonnumeric types, in which case it falls back to strict equality. This can be useful for comparing dicts and sequences that can contain optional values:

```
>>> {"required": 1.0000005, "optional": None} == approx({"required": 1, "optional": None})
True
>>> [None, 1.0000005] == approx([None, 1])
True
>>> ["foo", 1.0000005] == approx([None, 1])
False
```

If you’re thinking about using `approx`, then you might want to know how it compares to other good ways of comparing floating-point numbers. All of these algorithms are based on relative and absolute tolerances and should agree for the most part, but they do have meaningful differences:

- `math.isclose(a, b, rel_tol=1e-9, abs_tol=0.0)`: True if the relative tolerance is met w.r.t. either `a` or `b` or if the absolute tolerance is met. Because the relative tolerance is calculated w.r.t. both `a` and `b`, this test is symmetric (i.e. neither `a` nor `b` is a “reference value”). You have to specify an absolute tolerance if you want to compare to `0.0` because there is no tolerance by default. More information: `math.isclose()`.
- `numpy.isclose(a, b, rtol=1e-5, atol=1e-8)`: True if the difference between `a` and `b` is less than the sum of the relative tolerance w.r.t. `b` and the absolute tolerance. Because the relative tolerance

is only calculated w.r.t. `b`, this test is asymmetric and you can think of `b` as the reference value. Support for comparing sequences is provided by `numpy.allclose()`. More information: `numpy.isclose`.

- `unittest.TestCase.assertAlmostEqual(a, b)`: True if `a` and `b` are within an absolute tolerance of $1e-7$. No relative tolerance is considered, so this function is not appropriate for very large or very small numbers. Also, it's only available in subclasses of `unittest.TestCase` and it's ugly because it doesn't follow PEP8. More information: `unittest.TestCase.assertAlmostEqual()`.
- `a == pytest.approx(b, rel=1e-6, abs=1e-12)`: True if the relative tolerance is met w.r.t. `b` or if the absolute tolerance is met. Because the relative tolerance is only calculated w.r.t. `b`, this test is asymmetric and you can think of `b` as the reference value. In the special case that you explicitly specify an absolute tolerance but not a relative tolerance, only the absolute tolerance is considered.

Note: `approx` can handle numpy arrays, but we recommend the specialised test helpers in [Test Support \(numpy.testing\)](#) if you need support for comparisons, NaNs, or ULP-based tolerances.

Warning: Changed in version 3.2.

In order to avoid inconsistent behavior, `TypeError` is raised for `>`, `>=`, `<` and `<=` comparisons. The example below illustrates the problem:

```
assert approx(0.1) > 0.1 + 1e-10 # calls approx(0.1).__gt__(0.1 + 1e-10)
assert 0.1 + 1e-10 > approx(0.1) # calls approx(0.1).__lt__(0.1 + 1e-10)
```

In the second example one expects `approx(0.1).__le__(0.1 + 1e-10)` to be called. But instead, `approx(0.1).__lt__(0.1 + 1e-10)` is used to comparison. This is because the call hierarchy of rich comparisons follows a fixed behavior. More information: `object.__ge__()`

Changed in version 3.7.1: `approx` raises `TypeError` when it encounters a dict value or sequence element of nonnumeric type.

Changed in version 6.1.0: `approx` falls back to strict equality for nonnumeric types instead of raising `TypeError`.

Parameters `nan_ok` (*bool*) –

Return type `_pytest.python_api.ApproxBase`

pytest.fail

Tutorial: *How to use skip and xfail to deal with tests that cannot succeed*

fail (*msg*="", *pytrace*=True)

Explicitly fail an executing test with the given message.

Parameters

- **msg** (*str*) – The message to show the user as reason for the failure.
- **pytrace** (*bool*) – If False, `msg` represents the full failure information and no python traceback will be reported.

Return type `NoReturn`

pytest.skip

skip (*msg* [, *allow_module_level=False*])

Skip an executing test with the given message.

This function should be called only during testing (setup, call or teardown) or during collection by using the `allow_module_level` flag. This function can be called in doctests as well.

Parameters

- **allow_module_level** (*bool*) – Allows this function to be called at module level, skipping the rest of the module. Defaults to False.
- **msg** (*str*) –

Return type NoReturn

Note: It is better to use the `pytest.mark.skipif` marker when possible to declare a test to be skipped under certain conditions like mismatching platforms or dependencies. Similarly, use the `# doctest: +SKIP` directive (see `doctest.SKIP`) to skip a doctest statically.

pytest.importorskip

importorskip (*modname*, *minversion=None*, *reason=None*)

Import and return the requested module *modname*, or skip the current test if the module cannot be imported.

Parameters

- **modname** (*str*) – The name of the module to import.
- **minversion** (*str*) – If given, the imported module's `__version__` attribute must be at least this minimal version, otherwise the test is still skipped.
- **reason** (*str*) – If given, this reason is shown as the message when the module cannot be imported.

Returns The imported module. This should be assigned to its canonical name.

Return type Any

Example:

```
docutils = pytest.importorskip("docutils")
```

pytest.xfail

xfail (*reason=""*)

Imperatively xfail an executing test or setup function with the given reason.

This function should be called only during testing (setup, call or teardown).

Note: It is better to use the `pytest.mark.xfail` marker when possible to declare a test to be xfailed under certain conditions like known bugs or missing features.

Parameters **reason** (*str*) –

Return type NoReturn

pytest.exit

exit (*msg*, *returncode=None*)

Exit testing process.

Parameters

- **msg** (*str*) – Message to display upon exit.
- **returncode** (*int*) – Return code to be used when exiting pytest.

Return type NoReturn

pytest.main

main (*args=None*, *plugins=None*)

Perform an in-process test run.

Parameters

- **args** (*Optional[Union[List[str], os.PathLike[str]]]*) – List of command line arguments.
- **plugins** (*Optional[Sequence[Union[str, object]]]*) – List of plugin objects to be auto-registered during initialization.

Returns An exit code.

Return type Union[int, [_pytest.config.ExitCode](#)]

pytest.param

param (**values*, [*id*], [*marks*])

Specify a parameter in [pytest.mark.parametrize](#) calls or [parametrized fixtures](#).

```
@pytest.mark.parametrize(
    "test_input, expected",
    [
        ("3+5", 8),
        pytest.param("6*9", 42, marks=pytest.mark.xfail),
    ],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Parameters

- **values** (*object*) – Variable args of the values of the parameter set, in order.
- **marks** (*Union[_pytest.mark.structures.MarkDecorator, Collection[Union[_pytest.mark.structures.MarkDecorator, _pytest.mark.structures.Mark]]]*) – A single mark or a list of marks to be applied to this parameter set.
- **id** (*Optional[str]*) – The id to attribute to this parameter set.

- `marks` –
- `id` –

Return type `_pytest.mark.structures.ParameterSet`

pytest.raises

Tutorial: *Assertions about expected exceptions.*

with raises (*expected_exception*: `Union[Type[_pytest.python_api.E], Tuple[Type[_pytest.python_api.E], ...]]`, *, *match*: `Optional[Union[str, Pattern[str]]] = '.'`) → `RaisesContext[E]` **as excinfo**

with raises (*expected_exception*: `Union[Type[_pytest.python_api.E], Tuple[Type[_pytest.python_api.E], ...]]`, *func*: `Callable[[...], Any]`, **args*: `Any`, ***kwargs*: `Any`) → `_pytest._code.code.ExceptionInfo[_pytest.python_api.E]` **as excinfo**

Assert that a code block/function call raises `expected_exception` or raise a failure exception otherwise.

Parameters match – If specified, a string containing a regular expression, or a regular expression object, that is tested against the string representation of the exception using `re.search()`. To match a literal string that may contain special characters, the pattern can first be escaped with `re.escape()`.

(This is only used when `pytest.raises()` is used as a context manager, and passed through to the function otherwise. When using `pytest.raises()` as a function, you can use: `pytest.raises(Exc, func, match="passed on").match("my pattern")`.)

Use `pytest.raises` as a context manager, which will capture the exception of the given type:

```
>>> import pytest
>>> with pytest.raises(ZeroDivisionError):
...     1/0
```

If the code block does not raise the expected exception (`ZeroDivisionError` in the example above), or no exception at all, the check will fail instead.

You can also use the keyword argument `match` to assert that the exception matches a text or regex:

```
>>> with pytest.raises(ValueError, match='must be 0 or None'):
...     raise ValueError("value must be 0 or None")

>>> with pytest.raises(ValueError, match=r'must be \d+$'):
...     raise ValueError("value must be 42")
```

The context manager produces an `ExceptionInfo` object which can be used to inspect the details of the captured exception:

```
>>> with pytest.raises(ValueError) as exc_info:
...     raise ValueError("value must be 42")
>>> assert exc_info.type is ValueError
>>> assert exc_info.value.args[0] == "value must be 42"
```

Note: When using `pytest.raises` as a context manager, it's worthwhile to note that normal context manager rules apply and that the exception raised *must* be the final line in the scope of the context manager. Lines of code after that, within the scope of the context manager will not be executed. For example:

```
>>> value = 15
>>> with pytest.raises(ValueError) as exc_info:
...     if value > 10:
...         raise ValueError("value must be <= 10")
...     assert exc_info.type is ValueError # this will not execute
```

Instead, the following approach must be taken (note the difference in scope):

```
>>> with pytest.raises(ValueError) as exc_info:
...     if value > 10:
...         raise ValueError("value must be <= 10")
...
>>> assert exc_info.type is ValueError
```

Using with `pytest.mark.parametrize`

When using `pytest.mark.parametrize` it is possible to parametrize tests such that some runs raise an exception and others do not.

See [Parametrizing conditional raising](#) for an example.

Legacy form

It is possible to specify a callable by passing a to-be-called lambda:

```
>>> raises(ZeroDivisionError, lambda: 1/0)
<ExceptionInfo ...>
```

or you can specify an arbitrary callable with arguments:

```
>>> def f(x): return 1/x
...
>>> raises(ZeroDivisionError, f, 0)
<ExceptionInfo ...>
>>> raises(ZeroDivisionError, f, x=0)
<ExceptionInfo ...>
```

The form above is fully supported but discouraged for new code because the context manager form is regarded as more readable and less error-prone.

Note: Similar to caught exception objects in Python, explicitly clearing local references to returned `ExceptionInfo` objects can help the Python interpreter speed up its garbage collection.

Clearing those references breaks a reference cycle (`ExceptionInfo` → caught exception → frame stack raising the exception → current frame stack → local variables → `ExceptionInfo`) which makes Python keep all objects referenced from that cycle (including all local variables in the current frame) alive until the next cyclic garbage collection run. More detailed information can be found in the official Python documentation for [the try statement](#).

pytest.deprecated_call

Tutorial: *Ensuring code triggers a deprecation warning.*

with deprecated_call (*, match: Optional[Union[str, Pattern[str]]] = '...') → WarningsRecorder
with deprecated_call (func: Callable[[...], _pytest.recwarn.T], *args: Any, **kwargs: Any) →
 _pytest.recwarn.T

Assert that code produces a DeprecationWarning or PendingDeprecationWarning.

This function can be used as a context manager:

```
>>> import warnings
>>> def api_call_v2():
...     warnings.warn('use v3 of this api', DeprecationWarning)
...     return 200

>>> import pytest
>>> with pytest.deprecated_call():
...     assert api_call_v2() == 200
```

It can also be used by passing a function and *args and **kwargs, in which case it will ensure calling func(*args, **kwargs) produces one of the warnings types above. The return value is the return value of the function.

In the context manager form you may use the keyword argument match to assert that the warning matches a text or regex.

The context manager produces a list of warnings.WarningMessage objects, one for each warning raised.

pytest.register_assert_rewrite

Tutorial: *Assertion Rewriting.*

register_assert_rewrite (*names)

Register one or more module names to be rewritten on import.

This function will make sure that this module or all modules inside the package will get their assert statements rewritten. Thus you should make sure to call this before the module is actually imported, usually in your __init__.py if you are a plugin using a package.

Raises **TypeError** – If the given module names are not strings.

Parameters **names** (str) –

Return type None

pytest.warns

Tutorial: *Asserting warnings with the warns function*

with warns (expected_warning: Union[Type[Warning], Tuple[Type[Warning], ...]] = <class 'Warning'>, *, match: Optional[Union[str, Pattern[str]]] = None) → WarningsChecker

with warns (expected_warning: Union[Type[Warning], Tuple[Type[Warning], ...]], func: Callable[[...], _pytest.recwarn.T], *args: Any, **kwargs: Any) → _pytest.recwarn.T

Assert that code raises a particular class of warning.

Specifically, the parameter expected_warning can be a warning class or sequence of warning classes, and the inside the with block must issue a warning of that class or classes.

This helper produces a list of `warnings.WarningMessage` objects, one for each warning raised.

This function can be used as a context manager, or any of the other ways `pytest.raises()` can be used:

```
>>> import pytest
>>> with pytest.warns(RuntimeWarning):
...     warnings.warn("my warning", RuntimeWarning)
```

In the context manager form you may use the keyword argument `match` to assert that the warning matches a text or regex:

```
>>> with pytest.warns(UserWarning, match='must be 0 or None'):
...     warnings.warn("value must be 0 or None", UserWarning)

>>> with pytest.warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("value must be 42", UserWarning)

>>> with pytest.warns(UserWarning, match=r'must be \d+$'):
...     warnings.warn("this is not here", UserWarning)
Traceback (most recent call last):
...
Failed: DID NOT WARN. No warnings of type ...UserWarning... was emitted...
```

pytest.freeze_includes

Tutorial: *Freezing pytest.*

freeze_includes()

Return a list of module names used by pytest that should be included by `cx_freeze`.

Return type List[str]

3.4.2 Marks

Marks can be used apply meta data to *test functions* (but not fixtures), which can then be accessed by fixtures or plugins.

pytest.mark.filterwarnings

Tutorial: *@pytest.mark.filterwarnings.*

Add warning filters to marked test items.

`pytest.mark.filterwarnings` (*filter*)

Parameters **filter** (*str*) – A *warning specification string*, which is composed of contents of the tuple (action, message, category, module, lineno) as specified in [The Warnings Filter](#) section of the Python documentation, separated by " : ". Optional fields can be omitted. Module names passed for filtering are not regex-escaped.

For example:

```
@pytest.mark.filterwarnings("ignore:.*usage will be deprecated.
↳*:DeprecationWarning")
def test_foo():
    ...
```

pytest.mark.parametrize

How to parametrize fixtures and test functions.

This mark has the same signature as `pytest.Metafunc.parametrize()`; see there.

pytest.mark.skip

Skipping test functions.

Unconditionally skip a test function.

`pytest.mark.skip(reason=None)`

Parameters `reason` (*str*) – Reason why the test function is being skipped.

pytest.mark.skipif

Skipping test functions.

Skip a test function if a condition is `True`.

`pytest.mark.skipif(condition, *, reason=None)`

Parameters

- **condition** (*bool* or *str*) – `True/False` if the condition should be skipped or a *condition string*.
- **reason** (*str*) – Reason why the test function is being skipped.

pytest.mark.usefixtures

Tutorial: *Use fixtures in classes and modules with usefixtures.*

Mark a test function as using the given fixture names.

`pytest.mark.usefixtures(*names)`

Parameters `args` – The names of the fixture to use, as strings.

Note: When using `usefixtures` in hooks, it can only load fixtures when applied to a test function before test setup (for example in the `pytest_collection_modifyitems` hook).

Also note that this mark has no effect when applied to **fixtures**.

pytest.mark.xfail

Tutorial: *XFail: mark test functions as expected to fail.*

Marks a test function as *expected to fail*.

`pytest.mark.xfail` (*condition=None, *, reason=None, raises=None, run=True, strict=False*)

Parameters

- **condition** (*bool* or *str*) – Condition for marking the test function as xfail (`True/False` or a *condition string*). If a *bool*, you also have to specify *reason* (see *condition string*).
- **reason** (*str*) – Reason why the test function is marked as xfail.
- **raises** (*Type[Exception]*) – Exception subclass expected to be raised by the test function; other exceptions will fail the test.
- **run** (*bool*) – If the test function should actually be executed. If `False`, the function will always xfail and will not be executed (useful if a function is segfaulting).
- **strict** (*bool*) –
 - If `False` (the default) the function will be shown in the terminal output as `xfailed` if it fails and as `xpass` if it passes. In both cases this will not cause the test suite to fail as a whole. This is particularly useful to mark *flaky* tests (tests that fail at random) to be tackled later.
 - If `True`, the function will be shown in the terminal output as `xfailed` if it fails, but if it unexpectedly passes then it will **fail** the test suite. This is particularly useful to mark functions that are always failing and there should be a clear indication if they unexpectedly start to pass (for example a new release of a library fixes a known bug).

pytest.__version__

The current pytest version, as a string:

```
>>> import pytest
>>> pytest.__version__
'7.0.0'
```

pytest.version_tuple

New in version 7.0.

The current pytest version, as a tuple:

```
>>> import pytest
>>> pytest.version_tuple
(7, 0, 0)
```

For pre-releases, the last component will be a string with the prerelease version:

```
>>> import pytest
>>> pytest.version_tuple
(7, 0, '0rc1')
```

Custom marks

Marks are created dynamically using the factory object `pytest.mark` and applied as a decorator.

For example:

```
@pytest.mark.timeout(10, "slow", method="thread")
def test_function():
    ...
```

Will create and attach a *Mark* object to the collected *Item*, which can then be accessed by fixtures or hooks with *Node.iter_markers*. The mark object will have the following attributes:

```
mark.args == (10, "slow")
mark.kwargs == {"method": "thread"}
```

Example for using multiple custom markers:

```
@pytest.mark.timeout(10, "slow", method="thread")
@pytest.mark.slow
def test_function():
    ...
```

When *Node.iter_markers* or *Node.iter_markers* is used with multiple markers, the marker closest to the function will be iterated over first. The above example will result in `@pytest.mark.slow` followed by `@pytest.mark.timeout(...)`.

3.4.3 Fixtures

Tutorial: *Fixtures reference*.

Fixtures are requested by test functions or other fixtures by declaring them as argument names.

Example of a test requiring a fixture:

```
def test_output(capsys):
    print("hello")
    out, err = capsys.readouterr()
    assert out == "hello\n"
```

Example of a fixture requiring another fixture:

```
@pytest.fixture
def db_session(tmp_path):
    fn = tmp_path / "db.file"
    return connect(fn)
```

For more details, consult the full *fixtures docs*.

@pytest.fixture

```
@fixture(fixture_function: _pytest.fixtures.FixtureFunction, *, scope: Union[_ScopeName, Callable[[str, Config],
    _ScopeName]] = 'function', params: Optional[Iterable[object]] = None, autouse: bool = False, ids:
    Optional[Union[Iterable[Union[None, str, float, int, bool]], Callable[[Any], Optional[object]]]] = None,
    name: Optional[str] = None) → _pytest.fixtures.FixtureFunction
```

```
@fixture(fixture_function: None = None, *, scope: Union[_ScopeName, Callable[[str, Config], _ScopeName]] =
    'function', params: Optional[Iterable[object]] = None, autouse: bool = False, ids:
    Optional[Union[Iterable[Union[None, str, float, int, bool]], Callable[[Any], Optional[object]]]] = None,
    name: Optional[str] = None) → _pytest.fixtures.FixtureFunctionMarker
```

Decorator to mark a fixture factory function.

This decorator can be used, with or without parameters, to define a fixture function.

The name of the fixture function can later be referenced to cause its invocation ahead of running tests: test modules or classes can use the `pytest.mark.usefixtures(fixturename)` marker.

Test functions can directly use fixture names as input arguments in which case the fixture instance returned from the fixture function will be injected.

Fixtures can provide their values to test functions using `return` or `yield` statements. When using `yield` the code block after the `yield` statement is executed as teardown code regardless of the test outcome, and must yield exactly once.

Parameters

- **scope** – The scope for which this fixture is shared; one of "function" (default), "class", "module", "package" or "session".

This parameter may also be a callable which receives (`fixture_name`, `config`) as parameters, and must return a `str` with one of the values mentioned above.

See *Dynamic scope* in the docs for more information.

- **params** – An optional list of parameters which will cause multiple invocations of the fixture function and all of the tests using it. The current parameter is available in `request.param`.
- **autouse** – If True, the fixture func is activated for all tests that can see it. If False (the default), an explicit reference is needed to activate the fixture.
- **ids** – List of string ids each corresponding to the params so that they are part of the test id. If no ids are provided they will be generated automatically from the params.
- **name** – The name of the fixture. This defaults to the name of the decorated function. If a fixture is used in the same module in which it is defined, the function name of the fixture will be shadowed by the function arg that requests the fixture; one way to resolve this is to name the decorated function `fixture_<fixturename>` and then use `@pytest.fixture(name='<fixturename>')`.

config.cache

Tutorial: *How to re-run failed tests and maintain state between test runs.*

The `config.cache` object allows other plugins and fixtures to store and retrieve values across test runs. To access it from fixtures request `pytestconfig` into your fixture and get it with `pytestconfig.cache`.

Under the hood, the cache plugin uses the simple `dumps/loads` API of the `json` stdlib module.

`config.cache` is an instance of `pytest.Cache`:

```
final class Cache
```


Parameters

- **cachedir** (*pathlib.Path*) –
- **config** (*_pytest.config.Config*) –
- **_ispytest** (*bool*) –

Return type *None*

mkdir (*name*)

Return a directory path object with the given name.

If the directory does not yet exist, it will be created. You can use it to manage files to e.g. store/retrieve database dumps across test sessions.

New in version 6.3.

Parameters **name** (*str*) – Must be a string not containing a / separator. Make sure the name contains your plugin or application identifiers to prevent clashes with other cache users.

Return type *pathlib.Path*

makedir (*name*)

Return a directory path object with the given name.

Same as *mkdir()*, but returns a legacy py path instance.

Parameters **name** (*str*) –

Return type *py._path.local.LocalPath*

get (*key, default*)

Return the cached value for the given key.

If no value was yet cached or the value cannot be read, the specified default is returned.

Parameters

- **key** (*str*) – Must be a / separated value. Usually the first name is the name of your plugin or your application.
- **default** – The value to return in case of a cache-miss or invalid cache value.

set (*key, value*)

Save value for the given key.

Parameters

- **key** (*str*) – Must be a / separated value. Usually the first name is the name of your plugin or your application.
- **value** (*object*) – Must be of any combination of basic python types, including nested types like lists of dictionaries.

Return type *None*

capsys

How to capture stdout/stderr output.

capsys()

Enable text capturing of writes to `sys.stdout` and `sys.stderr`.

The captured output is made available via `capsys.readouterr()` method calls, which return a `(out, err)` namedtuple. `out` and `err` will be text objects.

Returns an instance of `CaptureFixture[str]`.

Example:

```
def test_output(capsys):
    print("hello")
    captured = capsys.readouterr()
    assert captured.out == "hello\n"
```

Parameters `request` (`_pytest.fixtures.SubRequest`) –

Return type `Generator[_pytest.capture.CaptureFixture[str], None, None]`

class CaptureFixture

Object returned by the `capsys`, `capsysbinary`, `capfd` and `capfdbinary` fixtures.

readouterr()

Read and return the captured output so far, resetting the internal buffer.

Returns The captured content as a namedtuple with `out` and `err` string attributes.

Return type `_pytest.capture.CaptureResult`

with disabled()

Temporarily disable capturing while inside the `with` block.

Return type `Generator[None, None, None]`

capsysbinary

How to capture stdout/stderr output.

capsysbinary()

Enable bytes capturing of writes to `sys.stdout` and `sys.stderr`.

The captured output is made available via `capsysbinary.readouterr()` method calls, which return a `(out, err)` namedtuple. `out` and `err` will be bytes objects.

Returns an instance of `CaptureFixture[bytes]`.

Example:

```
def test_output(capsysbinary):
    print("hello")
    captured = capsysbinary.readouterr()
    assert captured.out == b"hello\n"
```

Parameters `request` (`_pytest.fixtures.SubRequest`) –

Return type `Generator[_pytest.capture.CaptureFixture[bytes], None, None]`

capfd

How to capture stdout/stderr output.

capfd()

Enable text capturing of writes to file descriptors 1 and 2.

The captured output is made available via `capfd.readouterr()` method calls, which return a `(out, err)` namedtuple. `out` and `err` will be text objects.

Returns an instance of `CaptureFixture[str]`.

Example:

```
def test_system_echo(capfd):
    os.system('echo "hello"')
    captured = capfd.readouterr()
    assert captured.out == "hello\n"
```

Parameters `request` (`_pytest.fixtures.SubRequest`) –

Return type `Generator[_pytest.capture.CaptureFixture[str], None, None]`

capfdbinary

How to capture stdout/stderr output.

capfdbinary()

Enable bytes capturing of writes to file descriptors 1 and 2.

The captured output is made available via `capfd.readouterr()` method calls, which return a `(out, err)` namedtuple. `out` and `err` will be byte objects.

Returns an instance of `CaptureFixture[bytes]`.

Example:

```
def test_system_echo(capfdbinary):
    os.system('echo "hello"')
    captured = capfdbinary.readouterr()
    assert captured.out == b"hello\n"
```

Parameters `request` (`_pytest.fixtures.SubRequest`) –

Return type `Generator[_pytest.capture.CaptureFixture[bytes], None, None]`

doctest_namespace

How to run doctests.

doctest_namespace()

Fixture that returns a `dict` that will be injected into the namespace of doctests.

Usually this fixture is used in conjunction with another `autouse` fixture:

```
@pytest.fixture(autouse=True)
def add_np(doctest_namespace):
    doctest_namespace["np"] = numpy
```

For more details: `'doctest_namespace' fixture`.

Return type Dict[str, Any]

request

Pass different values to a test function, depending on command line options.

The `request` fixture is a special fixture providing information of the requesting test function.

class FixtureRequest

A request for a fixture from a test or fixture function.

A request object gives access to the requesting test context and has an optional `param` attribute in case the fixture is parametrized indirectly.

Parameters `_ispytest` (*bool*) –

Return type None

fixturename: Optional[str]

Fixture for which this request is being performed.

property scope: _ScopeName

Scope string, one of “function”, “class”, “module”, “package”, “session”.

property fixturenames: List[str]

Names of all active fixtures in this request.

property node

Underlying collection node (depends on current request scope).

property config: `_pytest.config.Config`

The pytest config object associated with this request.

property function

Test function object if the request has a per-function scope.

property cls

Class (can be None) where the test function was collected.

property instance

Instance (can be None) on which test function was collected.

property module

Python module object where the test function was collected.

property fspath: `py._path.local.LocalPath`

(deprecated) The file system path of the test module which collected this test.

property keywords: MutableMapping[str, Any]

Keywords/markers dictionary for the underlying node.

property session: `Session`

Pytest session object.

addfinalizer (*finalizer*)

Add finalizer/teardown function to be called after the last test within the requesting test context finished execution.

Parameters `finalizer` (*Callable[[], object]*) –

Return type None

applymarker (*marker*)

Apply a marker to a single test function invocation.

This method is useful if you don't want to have a keyword/marker on all function invocations.

Parameters *marker* (*Union[str, _pytest.mark.structures.MarkDecorator]*) – A *pytest.MarkDecorator* object created by a call to *pytest.mark.NAME(...)*.

Return type *None*

raiseerror (*msg*)

Raise a *FixtureLookupError* with the given message.

Parameters *msg* (*Optional[str]*) –

Return type *NoReturn*

getfixturevalue (*argname*)

Dynamically run a named fixture function.

Declaring fixtures via function argument is recommended where possible. But if you can only decide whether to use another fixture at test setup time, you may use this function to retrieve it inside a fixture or test function body.

Raises *pytest.FixtureLookupError* – If the given fixture could not be found.

Parameters *argname* (*str*) –

Return type *Any*

pytestconfig

pytestconfig ()

Session-scoped fixture that returns the session's *pytest.Config* object.

Example:

```
def test_foo(pytestconfig):
    if pytestconfig.getoption("verbose") > 0:
        ...
```

Parameters *request* (*_pytest.fixtures.FixtureRequest*) –

Return type *_pytest.config.Config*

record_property

Tutorial: *record_property* example.

record_property ()

Add extra properties to the calling test.

User properties become part of the test report and are available to the configured reporters, like JUnit XML.

The fixture is callable with *name*, *value*. The value is automatically XML-encoded.

Example:

```
def test_function(record_property):
    record_property("example_key", 1)
```

Parameters `request` (`_pytest.fixtures.FixtureRequest`) –

Return type `Callable[[str, object], None]`

record_testsuite_property

Tutorial: `record_testsuite_property` example.

record_testsuite_property()

Record a new `<property>` tag as child of the root `<testsuite>`.

This is suitable to writing global information regarding the entire test suite, and is compatible with xunit2 JUnit family.

This is a session-scoped fixture which is called with `(name, value)`. Example:

```
def test_foo(record_testsuite_property):
    record_testsuite_property("ARCH", "PPC")
    record_testsuite_property("STORAGE_TYPE", "CEPH")
```

name must be a string, value will be converted to a string and properly xml-escaped.

Warning: Currently this fixture **does not work** with the `pytest-xdist` plugin. See issue [#7767](#) for details.

Parameters `request` (`_pytest.fixtures.FixtureRequest`) –

Return type `Callable[[str, object], None]`

caplog

How to manage logging.

caplog()

Access and control log capturing.

Captured logs are available through the following properties/methods:

```
* caplog.messages      -> list of format-interpolated log messages
* caplog.text          -> string containing formatted log output
* caplog.records       -> list of logging.LogRecord instances
* caplog.record_tuples -> list of (logger_name, level, message) tuples
* caplog.clear()       -> clear captured records and formatted log output string
```

Returns a `pytest.LogCaptureFixture` instance.

Parameters `request` (`_pytest.fixtures.FixtureRequest`) –

Return type `Generator[_pytest.logging.LogCaptureFixture, None, None]`

final class LogCaptureFixture

Provides access and control of log capturing.

Parameters

- `item(_pytest.nodes.Node)` –
- `_ispytest(bool)` –

Return type `None`

property handler: `_pytest.logging.LogCaptureHandler`

Get the logging handler used by the fixture.

Return type `LogCaptureHandler`

get_records (*when*)

Get the logging records for one of the possible test phases.

Parameters *when* (*str*) – Which test phase to obtain the records from. Valid values are: “setup”, “call” and “teardown”.

Returns The list of captured records at the given stage.

Return type `List[logging.LogRecord]`

New in version 3.4.

property text: `str`

The formatted log text.

property records: `List[logging.LogRecord]`

The list of log records.

property record_tuples: `List[Tuple[str, int, str]]`

A list of a stripped down version of log records intended for use in assertion comparison.

The format of the tuple is:

(*logger_name*, *log_level*, *message*)

property messages: `List[str]`

A list of format-interpolated log messages.

Unlike ‘records’, which contains the format string and parameters for interpolation, log messages in this list are all interpolated.

Unlike ‘text’, which contains the output from the handler, log messages in this list are unadorned with levels, timestamps, etc, making exact comparisons more reliable.

Note that traceback or stack info (from `logging.exception()` or the `exc_info` or `stack_info` arguments to the logging functions) is not included, as this is added by the formatter in the handler.

New in version 3.7.

clear ()

Reset the list of log records and the captured log text.

Return type `None`

set_level (*level*, *logger=None*)

Set the level of a logger for the duration of a test.

Changed in version 3.4: The levels of the loggers changed by this function will be restored to their initial values at the end of the test.

Parameters

- **level** (*int*) – The level.
- **logger** (*str*) – The logger to update. If not given, the root logger.

Return type `None`

with at_level (*level*, *logger=None*)

Context manager that sets the level for capturing of logs. After the end of the ‘with’ statement the level is restored to its original value.

Parameters

- **level** (*int*) – The level.
- **logger** (*str*) – The logger to update. If not given, the root logger.

Return type `Generator[None, None, None]`

monkeypatch

How to monkeypatch/mock modules and environments.

monkeypatch ()

A convenient fixture for monkey-patching.

The fixture provides these methods to modify objects, dictionaries or os.environ:

```
monkeypatch.setattr(obj, name, value, raising=True)
monkeypatch.delattr(obj, name, raising=True)
monkeypatch.setitem(mapping, name, value)
monkeypatch.delitem(obj, name, raising=True)
monkeypatch.setenv(name, value, prepend=None)
monkeypatch.delenv(name, raising=True)
monkeypatch.syspath_prepend(path)
monkeypatch.chdir(path)
```

All modifications will be undone after the requesting test function or fixture has finished. The `raising` parameter determines if a `KeyError` or `AttributeError` will be raised if the set/deletion operation has no target.

Returns a `MonkeyPatch` instance.

Return type `Generator[_pytest.monkeypatch.MonkeyPatch, None, None]`

final class MonkeyPatch

Helper to conveniently monkeypatch attributes/items/environment variables/syspath.

Returned by the `monkeypatch` fixture.

Versionchanged: 6.2 Can now also be used directly as `pytest.MonkeyPatch()`, for when the fixture is not available. In this case, use `with MonkeyPatch.context() as mp:` or remember to call `undo()` explicitly.

Return type `None`

classmethod with context ()

Context manager that returns a new `MonkeyPatch` object which undoes any patching done inside the `with` block upon exit.

Example:

```
import functools

def test_partial(monkeypatch):
    with monkeypatch.context() as m:
        m.setattr(functools, "partial", 3)
```


Useful in situations where it is desired to undo some patches before the test ends, such as mocking `stdlib` functions that might break `pytest` itself if mocked (for examples of this see #3290).

Return type `Generator[_pytest.monkeypatch.MonkeyPatch, None, None]`

setattr (*target: str, name: object, value: _pytest.monkeypatch.Notset = <notset>, raising: bool = True*) → `None`

setattr (*target: object, name: str, value: object, raising: bool = True*) → `None`

Set attribute value on target, memorizing the old value.

For convenience you can specify a string as `target` which will be interpreted as a dotted import path, with the last part being the attribute name. For example, `monkeypatch.setattr("os.getcwd", lambda: "/")` would set the `getcwd` function of the `os` module.

Raises `AttributeError` if the attribute does not exist, unless `raising` is set to `False`.

delattr (*target, name=<notset>, raising=True*)

Delete attribute name from target.

If no `name` is specified and `target` is a string it will be interpreted as a dotted import path with the last part being the attribute name.

Raises `AttributeError` if the attribute does not exist, unless `raising` is set to `False`.

Parameters

- **target** (`Union[object, str]`) –
- **name** (`Union[str, _pytest.monkeypatch.Notset]`) –
- **raising** (`bool`) –

Return type `None`

setitem (*dic, name, value*)

Set dictionary entry name to value.

Parameters

- **dic** (`MutableMapping[_pytest.monkeypatch.K, _pytest.monkeypatch.V]`) –
- **name** (`_pytest.monkeypatch.K`) –
- **value** (`_pytest.monkeypatch.V`) –

Return type `None`

delitem (*dic, name, raising=True*)

Delete name from dict.

Raises `KeyError` if it doesn't exist, unless `raising` is set to `False`.

Parameters

- **dic** (`MutableMapping[_pytest.monkeypatch.K, _pytest.monkeypatch.V]`) –
- **name** (`_pytest.monkeypatch.K`) –
- **raising** (`bool`) –

Return type `None`

setenv (*name, value, prepend=None*)

Set environment variable name to value.

If `prepend` is a character, read the current environment variable value and prepend the value adjoined with the `prepend` character.

Parameters

- **name** (*str*) –
- **value** (*str*) –
- **prepend** (*Optional[str]*) –

Return type `None`

delenv (*name*, *raising=True*)

Delete `name` from the environment.

Raises `KeyError` if it does not exist, unless `raising` is set to `False`.

Parameters

- **name** (*str*) –
- **raising** (*bool*) –

Return type `None`

syspath_prepend (*path*)

Prepend `path` to `sys.path` list of import locations.

Return type `None`

chdir (*path*)

Change the current working directory to the specified path.

Path can be a string or a path object.

Parameters **path** (*Union[str, os.PathLike[str]]*) –

Return type `None`

undo ()

Undo previous changes.

This call consumes the undo stack. Calling it a second time has no effect unless you do more monkeypatching after the undo call.

There is generally no need to call `undo()`, since it is called automatically during tear-down.

Note that the same `monkeypatch` fixture is used across a single test function invocation. If `monkeypatch` is used both by the test function itself and one of the test fixtures, calling `undo()` will undo all of the changes made in both functions.

Return type `None`

pytester

New in version 6.2.

Provides a `Pytester` instance that can be used to run and test pytest itself.

It provides an empty directory where pytest can be executed in isolation, and contains facilities to write tests, configuration files, and match against expected output.

To use it, include in your topmost `conftest.py` file:

```
pytest_plugins = "pytester"
```

final class Pytester

Facilities to write tests/configuration files, execute pytest in isolation, and match against expected output, perfect for black-box testing of pytest plugins.

It attempts to isolate the test run from external factors as much as possible, modifying the current working directory to `path` and environment variables during initialization.

Attributes:

Variables

- **path** (*Path*) – temporary directory path used to create files/run tests from, etc.
- **plugins** – A list of plugins to use with `parseconfig()` and `runpytest()`. Initially this is an empty list but plugins can be added to the list. The type of items to add to the list depends on the method using them so refer to them for details.

Parameters

- **request** (`_pytest.fixtures.FixtureRequest`) –
- **tmp_path_factory** (`_pytest.tmpdir.TempPathFactory`) –
- **_ispytest** (*bool*) –

Return type `None`

exception TimeoutExpired

property **path**: `pathlib.Path`

Temporary directory where files are created and pytest is executed.

make_hook_recorder (*pluginmanager*)

Create a new `HookRecorder` for a `PluginManager`.

Parameters **pluginmanager** (`_pytest.config.PytestPluginManager`) –

Return type `_pytest.pytester.HookRecorder`

chdir ()

Cd into the temporary directory.

This is done automatically upon instantiation.

Return type `None`

makefile (*ext*, **args*, ***kwargs*)

Create new text file(s) in the test directory.

Parameters

- **ext** (*str*) – The extension the file(s) should use, including the dot, e.g. `.py`.
- **args** (*str*) – All args are treated as strings and joined using newlines. The result is written as contents to the file. The name of the file is based on the test function requesting this fixture.
- **kwargs** (*str*) – Each keyword is the name of a file, while the value of it will be written as contents of the file.

Return type `pathlib.Path`

Examples:

```

pytester.makefile(".txt", "line1", "line2")

pytester.makefile(".ini", pytest="[pytest]\naddopts=-rs\n")

```

To create binary files, use `pathlib.Path.write_bytes()` directly:

```

filename = pytester.path.joinpath("foo.bin")
filename.write_bytes(b"...")

```

makeconftest (*source*)

Write a `conftest.py` file with ‘source’ as contents.

Parameters `source` (*str*) –

Return type `pathlib.Path`

makeini (*source*)

Write a `tox.ini` file with ‘source’ as contents.

Parameters `source` (*str*) –

Return type `pathlib.Path`

getinicfg (*source*)

Return the `pytest` section from the `tox.ini` config file.

Parameters `source` (*str*) –

Return type `iniconfig.SectionWrapper`

makepyprojecttoml (*source*)

Write a `pyproject.toml` file with ‘source’ as contents.

New in version 6.0.

Parameters `source` (*str*) –

Return type `pathlib.Path`

makepyfile (**args, **kwargs*)

Shortcut for `.makefile()` with a `.py` extension.

Defaults to the test name with a `.py` extension, e.g `test_foobar.py`, overwriting existing files.

Examples:

```

def test_something(pytester):
    # Initial file is created test_something.py.
    pytester.makepyfile("foobar")
    # To create multiple files, pass kwargs accordingly.
    pytester.makepyfile(custom="foobar")
    # At this point, both 'test_something.py' & 'custom.py' exist in the test_
    ↪directory.

```

Return type `pathlib.Path`

maketxtfile (**args, **kwargs*)

Shortcut for `.makefile()` with a `.txt` extension.

Defaults to the test name with a `.txt` extension, e.g `test_foobar.txt`, overwriting existing files.

Examples:

```
def test_something(pytester):
    # Initial file is created test_something.txt.
    pytester.maketxtfile("foobar")
    # To create multiple files, pass kwargs accordingly.
    pytester.maketxtfile(custom="foobar")
    # At this point, both 'test_something.txt' & 'custom.txt' exist in the
    ↪test directory.
```

Return type `pathlib.Path`

syspathinsert (*path=None*)

Prepend a directory to `sys.path`, defaults to *path*.

This is undone automatically when this object dies at the end of each test.

Parameters *path* (*Optional[Union[str, os.PathLike[str]]]*) –

Return type `None`

mkdir (*name*)

Create a new (sub)directory.

Parameters *name* (*str*) –

Return type `pathlib.Path`

mkpydir (*name*)

Create a new python package.

This creates a (sub)directory with an empty `__init__.py` file so it gets recognised as a Python package.

Parameters *name* (*str*) –

Return type `pathlib.Path`

copy_example (*name=None*)

Copy file from project's directory into the testdir.

Parameters *name* (*str*) – The name of the file to copy.

Returns path to the copied directory (inside `self.path`).

Return type `pathlib.Path`

class Session (**k, **kw*)

exception Failed

Signals a stop as failed test run.

exception Interrupted

Signals that the test run was interrupted.

for ... in collect ()

Return a list of children (items and collectors) for this collection node.

Return type `Iterator[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

perform_collect (*args=None, genitems=True*)

Perform the collection phase for this session.

This is called by the default `pytest_collection` hook implementation; see the documentation of this hook for more details. For testing purposes, it may also be called directly on a fresh `Session`.

This function normally recursively expands any collectors collected from the session to their items, and only items are returned. For testing purposes, this may be suppressed by passing `genitems=False`, in which case the return value contains these collectors unexpanded, and `session.items` is empty.

Parameters

- **args** (*Optional[Sequence[str]]*) –
- **genitems** (*bool*) –

Return type `Sequence[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

property stardir: `py._path.local.LocalPath`

The path from which pytest was invoked.

Prefer to use `startpath` which is a `pathlib.Path`.

Type `LEGACY_PATH`

property startpath: `pathlib.Path`

The path from which pytest was invoked.

New in version 6.3.0.

getnode (*config, arg*)

Return the collection node of a file.

Parameters

- **config** (`pytest.Config`) – A pytest config. See `parseconfig()` and `parseconfigfigure()` for creating it.
- **arg** (*os.PathLike[str]*) – Path to the file.

Return type `Optional[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

getpathnode (*path*)

Return the collection node of a file.

This is like `getnode()` but uses `parseconfigfigure()` to create the (configured) pytest Config instance.

Parameters **path** (*os.PathLike[str]*) – Path to the file.

genitems (*colitems*)

Generate all test items from a collection node.

This recurses into the collection node and returns a list of all the test items contained within.

Parameters **colitems** (*Sequence[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]*) –

Return type `List[_pytest.nodes.Item]`

runitem (*source*)

Run the “test_func” Item.

The calling test instance (class containing the test method) must provide a `.getrunner()` method which should return a runner which can run the test protocol for a single item, e.g. `_pytest.runner.runtestprotocol()`.

Parameters **source** (*str*) –

Return type Any

inline_runsource (*source, *cmdlineargs*)

Run a test module in process using `pytest.main()`.

This run writes “source” into a temporary file and runs `pytest.main()` on it, returning a `HookRecorder` instance for the result.

Parameters

- **source** (*str*) – The source code of the test module.
- **cmdlineargs** – Any extra command line arguments to use.

Returns HookRecorder instance of the result.

Return type *pytest.pytester.HookRecorder*

inline_genitems (**args*)

Run `pytest.main(['--collectonly'])` in-process.

Runs the `pytest.main()` function to run all of pytest inside the test process itself like `inline_run()`, but returns a tuple of the collected items and a HookRecorder instance.

Return type `Tuple[List[_pytest.nodes.Item], _pytest.pytester.HookRecorder]`

inline_run (**args, plugins=(), no_reraise_ctrlc=False*)

Run `pytest.main()` in-process, returning a HookRecorder.

Runs the `pytest.main()` function to run all of pytest inside the test process itself. This means it can return a HookRecorder instance which gives more detailed results from that run than can be done by matching stdout/stderr from `runpytest()`.

Parameters

- **args** (`Union[str, os.PathLike[str]]`) – Command line arguments to pass to `pytest.main()`.
- **plugins** – Extra plugin instances the `pytest.main()` instance should use.
- **no_reraise_ctrlc** (*bool*) – Typically we reraise keyboard interrupts from the child run. If True, the KeyboardInterrupt exception is captured.

Returns A HookRecorder instance.

Return type *pytest.pytester.HookRecorder*

runpytest_inprocess (**args, **kwargs*)

Return result of running pytest in-process, providing a similar interface to what `self.runpytest()` provides.

Parameters

- **args** (`Union[str, os.PathLike[str]]`) –
- **kwargs** (*Any*) –

Return type *pytest.pytester.RunResult*

runpytest (**args, **kwargs*)

Run pytest inline or in a subprocess, depending on the command line option “`--runpytest`” and return a Run-Result.

Parameters

- **args** (`Union[str, os.PathLike[str]]`) –
- **kwargs** (*Any*) –

Return type *pytest.pytester.RunResult*

parseconfig (**args*)

Return a new pytest Config instance from given commandline args.

This invokes the pytest bootstrapping code in `_pytest.config` to create a new `_pytest.core.PluginManager` and call the `pytest_cmdline_parse` hook to create a new `pytest.Config` instance.

If `plugins` has been populated they should be plugin modules to be registered with the `PluginManager`.

Parameters `args` (`Union[str, os.PathLike[str]]`) –

Return type `_pytest.config.Config`

parseconfigure (`*args`)

Return a new pytest configured `Config` instance.

Returns a new `pytest.Config` instance like `parseconfig()`, but also calls the `pytest_configure` hook.

Parameters `args` (`Union[str, os.PathLike[str]]`) –

Return type `_pytest.config.Config`

getitem (`source, funcname='test_func'`)

Return the test item for a test function.

Writes the source to a python file and runs pytest's collection on the resulting module, returning the test item for the requested function name.

Parameters

- **source** (`Union[str, os.PathLike[str]]`) – The module source.
- **funcname** (`str`) – The name of the test function for which to return a test item.

Return type `_pytest.nodes.Item`

getitems (`source`)

Return all test items collected from the module.

Writes the source to a Python file and runs pytest's collection on the resulting module, returning all test items contained within.

Parameters `source` (`Union[str, os.PathLike[str]]`) –

Return type `List[_pytest.nodes.Item]`

getmodulecol (`source, configargs=(), *, withinit=False`)

Return the module collection node for `source`.

Writes `source` to a file using `makepyfile()` and then runs the pytest collection on it, returning the collection node for the test module.

Parameters

- **source** (`Union[str, os.PathLike[str]]`) – The source code of the module to collect.
- **configargs** – Any extra arguments to pass to `parseconfigure()`.
- **withinit** (`bool`) – Whether to also write an `__init__.py` file to the same directory to ensure it is a package.

collect_by_name (`modcol, name`)

Return the collection node for `name` from the module collection.

Searchs a module collection node for a collection node matching the given name.

Parameters

- **modcol** (`_pytest.nodes.Collector`) – A module collection node; see `getmodulecol()`.
- **name** (`str`) – The name of the node to return.

Return type `Optional[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

popen (*cmdargs*, *stdout=- 1*, *stderr=- 1*, *stdin=NotSetType.token*, ***kw*)

Invoke `subprocess.Popen`.

Calls `subprocess.Popen` making sure the current working directory is in `PYTHONPATH`.

You probably want to use `run()` instead.

Parameters

- **cmdargs** (*Sequence[Union[str, os.PathLike[str]]]*) –
- **stdout** (*Union[int, TextIO]*) –
- **stderr** (*Union[int, TextIO]*) –
- **stdin** (*Union[_pytest.compat.NotSetType, bytes, IO[Any], int]*) –

run (**cmdargs*, *timeout=None*, *stdin=NotSetType.token*)

Run a command with arguments.

Run a process using `subprocess.Popen` saving the `stdout` and `stderr`.

Parameters

- **cmdargs** (*Union[str, os.PathLike[str]]*) – The sequence of arguments to pass to `subprocess.Popen`, with path-like objects being converted to `str` automatically.
- **timeout** (*Optional[float]*) – The period in seconds after which to timeout and raise `Pytester.TimeoutExpired`.
- **stdin** (*Union[_pytest.compat.NotSetType, bytes, IO[Any], int]*) – Optional standard input.
 - If it is `CLOSE_STDIN` (Default), then this method calls `subprocess.Popen` with `stdin=subprocess.PIPE`, and the standard input is closed immediately after the new command is started.
 - If it is of type `bytes`, these bytes are sent to the standard input of the command.
 - Otherwise, it is passed through to `subprocess.Popen`. For further information in this case, consult the document of the `stdin` parameter in `subprocess.Popen`.

Return type `_pytest.pytester.RunResult`

runpython (*script*)

Run a python script using `sys.executable` as interpreter.

Parameters **script** (*os.PathLike[str]*) –

Return type `_pytest.pytester.RunResult`

runpython_c (*command*)

Run `python -c "command"`.

Parameters **command** (*str*) –

Return type `_pytest.pytester.RunResult`

runpytest_subprocess (**args*, *timeout=None*)

Run `pytest` as a subprocess with given arguments.

Any plugins added to the `plugins` list will be added using the `-p` command line option. Additionally `--basetemp` is used to put any temporary files and directories in a numbered directory prefixed with “`runpytest-`” to not conflict with the normal numbered `pytest` location for temporary files and directories.

Parameters

- **args** (*Union[str, os.PathLike[str]]*) – The sequence of arguments to pass to the pytest subprocess.
- **timeout** (*Optional[float]*) – The period in seconds after which to timeout and raise *Pytester.TimeoutExpired*.

Return type *_pytest.pytester.RunResult*

spawn_pytest (*string, expect_timeout=10.0*)

Run pytest using pexpect.

This makes sure to use the right pytest and sets up the temporary directory locations.

The pexpect child is returned.

Parameters

- **string** (*str*) –
- **expect_timeout** (*float*) –

Return type *pexpect.spawn*

spawn (*cmd, expect_timeout=10.0*)

Run a command using pexpect.

The pexpect child is returned.

Parameters

- **cmd** (*str*) –
- **expect_timeout** (*float*) –

Return type *pexpect.spawn*

class RunResult

The result of running a command.

Parameters

- **ret** (*Union[int, _pytest.config.ExitCode]*) –
- **outlines** (*List[str]*) –
- **errlines** (*List[str]*) –
- **duration** (*float*) –

Return type *None*

ret: *Union[int, _pytest.config.ExitCode]*

The return value.

outlines

List of lines captured from stdout.

errlines

List of lines captured from stderr.

stdout

LineMatcher of stdout.

Use e.g. *str(stdout)* to reconstruct stdout, or the commonly used *stdout.fnmatch_lines()* method.

stderr*LineMatcher* of `stderr`.**duration**

Duration in seconds.

parseoutcomes()

Return a dictionary of outcome noun -> count from parsing the terminal output that the test process produced.

The returned nouns will always be in plural form:

```
===== 1 failed, 1 passed, 1 warning, 1 error in 0.13s =====
```

Will return `{"failed": 1, "passed": 1, "warnings": 1, "errors": 1}`.**Return type** `Dict[str, int]`**classmethod parse_summary_nouns(lines)**

Extract the nouns from a pytest terminal summary line.

It always returns the plural noun for consistency:

```
===== 1 failed, 1 passed, 1 warning, 1 error in 0.13s =====
```

Will return `{"failed": 1, "passed": 1, "warnings": 1, "errors": 1}`.**Return type** `Dict[str, int]`**assert_outcomes** (*passed=0, skipped=0, failed=0, errors=0, xpassed=0, xfailed=0, warnings=0, deselected=0*)

Assert that the specified outcomes appear with the respective numbers (0 means it didn't occur) in the text output from a test run.

Parameters

- **passed** (*int*) –
- **skipped** (*int*) –
- **failed** (*int*) –
- **errors** (*int*) –
- **xpassed** (*int*) –
- **xfailed** (*int*) –
- **warnings** (*int*) –
- **deselected** (*int*) –

Return type `None`**class LineMatcher**

Flexible matching of text.

This is a convenience class to test large texts like the output of commands.

The constructor takes a list of lines without their trailing newlines, i.e. `text.splitlines()`.**Parameters** **lines** (*List[str]*) –**Return type** `None`

`__str__()`

Return the entire original text.

New in version 6.2: You can use `str()` in older versions.

Return type `str`

`fnmatch_lines_random(lines2)`

Check lines exist in the output in any order (using `fnmatch.fnmatch()`).

Parameters `lines2` (`Sequence[str]`) –

Return type `None`

`re_match_lines_random(lines2)`

Check lines exist in the output in any order (using `re.match()`).

Parameters `lines2` (`Sequence[str]`) –

Return type `None`

`get_lines_after(fline)`

Return all lines following the given line in the text.

The given line can contain glob wildcards.

Parameters `fline` (`str`) –

Return type `Sequence[str]`

`fnmatch_lines(lines2, *, consecutive=False)`

Check lines exist in the output (using `fnmatch.fnmatch()`).

The argument is a list of lines which have to match and can use glob wildcards. If they do not match a `pytest.fail()` is called. The matches and non-matches are also shown as part of the error message.

Parameters

- `lines2` (`Sequence[str]`) – String patterns to match.
- `consecutive` (`bool`) – Match lines consecutively?

Return type `None`

`re_match_lines(lines2, *, consecutive=False)`

Check lines exist in the output (using `re.match()`).

The argument is a list of lines which have to match using `re.match`. If they do not match a `pytest.fail()` is called.

The matches and non-matches are also shown as part of the error message.

Parameters

- `lines2` (`Sequence[str]`) – string patterns to match.
- `consecutive` (`bool`) – match lines consecutively?

Return type `None`

`no_fnmatch_line(pat)`

Ensure captured lines do not match the given pattern, using `fnmatch.fnmatch`.

Parameters `pat` (`str`) – The pattern to match lines.

Return type `None`

no_re_match_line (*pat*)

Ensure captured lines do not match the given pattern, using `re.match`.

Parameters **pat** (*str*) – The regular expression to match lines.

Return type `None`

str ()

Return the entire original text.

Return type `str`

class HookRecorder

Record all hooks called in a plugin manager.

This wraps all the hook calls in the plugin manager, recording each call before propagating the normal calls.

Parameters **pluginmanager** (`_pytest.config.PytestPluginManager`) –

Return type `None`

matchreport (*inamepart*="", *names*=('pytest_runtest_logreport', 'pytest_collectreport'), *when*=None)

Return a testreport whose dotted import path matches.

Parameters

- **inamepart** (*str*) –
- **names** (`Union[str, Iterable[str]]`) –
- **when** (`Optional[str]`) –

Return type `Union[_pytest.reports.CollectReport, _pytest.reports.TestReport]`

testdir

Identical to `pytester`, but provides an instance whose methods return legacy `py.path.local` objects instead when applicable.

New code should avoid using `testdir` in favor of `pytester`.

final class Testdir

Similar to `Pytester`, but this class works with legacy `legacy_path` objects instead.

All methods just forward to an internal `Pytester` instance, converting results to `legacy_path` objects as necessary.

Parameters

- **pytester** (`_pytest.pytester.Pytester`) –
- **_ispytest** (*bool*) –

Return type `None`

exception TimeoutExpired

class Session (**k, **kw*)

exception Failed

Signals a stop as failed test run.

exception Interrupted

Signals that the test run was interrupted.

for ... in collect()
 Return a list of children (items and collectors) for this collection node.
Return type Iterator[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]

perform_collect (*args=None, genitems=True*)
 Perform the collection phase for this session.
 This is called by the default `pytest_collection` hook implementation; see the documentation of this hook for more details. For testing purposes, it may also be called directly on a fresh `Session`.
 This function normally recursively expands any collectors collected from the session to their items, and only items are returned. For testing purposes, this may be suppressed by passing `genitems=False`, in which case the return value contains these collectors unexpanded, and `session.items` is empty.

Parameters

- **args** (*Optional[Sequence[str]]*) –
- **genitems** (*bool*) –

Return type Sequence[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]

property stardir: py._path.local.LocalPath
 The path from which pytest was invoked.
 Prefer to use `startpath` which is a `pathlib.Path`.
Type LEGACY_PATH

property startpath: pathlib.Path
 The path from which pytest was invoked.
 New in version 6.3.0.

property tmpdir: py._path.local.LocalPath
 Temporary directory where tests are executed.

make_hook_recorder (*pluginmanager*)
 See `Pytester.make_hook_recorder()`.
Return type `_pytest.pytester.HookRecorder`

chdir ()
 See `Pytester.chdir()`.
Return type `None`

finalize ()
 See `Pytester._finalize()`.
Return type `None`

makefile (*ext, *args, **kwargs*)
 See `Pytester.makefile()`.
Return type `py._path.local.LocalPath`

makeconftest (*source*)
 See `Pytester.makeconftest()`.
Return type `py._path.local.LocalPath`

makeini (*source*)
 See `Pytester.makeini()`.
Return type `py._path.local.LocalPath`

getinicfg (*source*)
 See `Pytester.getinicfg()`.

Parameters `source` (*str*) –

Return type `iniconfig.SectionWrapper`

makepyprojecttoml (*source*)

See `Pytester.makepyprojecttoml()`.

Return type `py._path.local.LocalPath`

makepyfile (**args, **kwargs*)

See `Pytester.makepyfile()`.

Return type `py._path.local.LocalPath`

maketxtfile (**args, **kwargs*)

See `Pytester.maketxtfile()`.

Return type `py._path.local.LocalPath`

syspathinsert (*path=None*)

See `Pytester.syspathinsert()`.

Return type `None`

mkdir (*name*)

See `Pytester.mkdir()`.

Return type `py._path.local.LocalPath`

mkpydir (*name*)

See `Pytester.mkpydir()`.

Return type `py._path.local.LocalPath`

copy_example (*name=None*)

See `Pytester.copy_example()`.

Return type `py._path.local.LocalPath`

getnode (*config, arg*)

See `Pytester.getnode()`.

Parameters `config` (`_pytest.config.Config`) –

Return type `Optional[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

getpathnode (*path*)

See `Pytester.getpathnode()`.

genitems (*colitems*)

See `Pytester.genitems()`.

Parameters `colitems` (`List[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`) –

Return type `List[_pytest.nodes.Item]`

runitem (*source*)

See `Pytester.runitem()`.

inline_runsource (*source, *cmdlineargs*)

See `Pytester.inline_runsource()`.

inline_genitems (**args*)

See `Pytester.inline_genitems()`.

inline_run (*args, plugins=(), no_reraise_ctrlc=False)
 See `Pytester.inline_run()`.

Parameters `no_reraise_ctrlc` (*bool*) –

runpytest_inprocess (*args, **kwargs)
 See `Pytester.runpytest_inprocess()`.

Return type `_pytest.pytester.RunResult`

runpytest (*args, **kwargs)
 See `Pytester.runpytest()`.

Return type `_pytest.pytester.RunResult`

parseconfig (*args)
 See `Pytester.parseconfig()`.

Return type `_pytest.config.Config`

parseconfigure (*args)
 See `Pytester.parseconfigure()`.

Return type `_pytest.config.Config`

getitem (source, funcname='test_func')
 See `Pytester.getitem()`.

getitems (source)
 See `Pytester.getitems()`.

getmodulecol (source, configargs=(), withinit=False)
 See `Pytester.getmodulecol()`.

collect_by_name (modcol, name)
 See `Pytester.collect_by_name()`.

Parameters

- `modcol` (`_pytest.nodes.Collector`) –
- `name` (*str*) –

Return type `Optional[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

popen (cmdargs, stdout=- 1, stderr=- 1, stdin=NotSetType.token, **kw)
 See `Pytester.popen()`.

run (*cmdargs, timeout=None, stdin=NotSetType.token)
 See `Pytester.run()`.

Return type `_pytest.pytester.RunResult`

runpython (script)
 See `Pytester.runpython()`.

Return type `_pytest.pytester.RunResult`

runpython_c (command)
 See `Pytester.runpython_c()`.

runpytest_subprocess (*args, timeout=None)
 See `Pytester.runpytest_subprocess()`.

Return type `_pytest.pytester.RunResult`

spawn_pytest (*string*, *expect_timeout=10.0*)

See `Pytester.spawn_pytest()`.

Parameters

- **string** (*str*) –
- **expect_timeout** (*float*) –

Return type `pexpect.spawn`

spawn (*cmd*, *expect_timeout=10.0*)

See `Pytester.spawn()`.

Parameters

- **cmd** (*str*) –
- **expect_timeout** (*float*) –

Return type `pexpect.spawn`

recwarn

Tutorial: *Asserting warnings with the warns function*

recwarn ()

Return a `WarningsRecorder` instance that records all warnings emitted by test functions.

See <https://docs.python.org/library/how-to/capture-warnings.html> for information on warning categories.

Return type `Generator[_pytest.recwarn.WarningsRecorder, None, None]`

class WarningsRecorder

A context manager to record raised warnings.

Adapted from `warnings.catch_warnings`.

Parameters **_ispytest** (*bool*) –

Return type `None`

property list: `List[warnings.WarningMessage]`

The list of recorded warnings.

pop (*cls=<class 'Warning'>*)

Pop the first recorded warning, raise exception if not exists.

Parameters **cls** (*Type[Warning]*) –

Return type `warnings.WarningMessage`

clear ()

Clear the list of recorded warnings.

Return type `None`

Each recorded warning is an instance of `warnings.WarningMessage`.

Note: `DeprecationWarning` and `PendingDeprecationWarning` are treated differently; see *Ensuring code triggers a deprecation warning*.

tmp_path

How to use temporary directories and files in tests

tmp_path()

Return a temporary directory path object which is unique to each test function invocation, created as a sub directory of the base temporary directory.

By default, a new base temporary directory is created each test session, and old bases are removed after 3 sessions, to aid in debugging. If `--basetemp` is used then it is cleared each session. See *The default base temporary directory*.

The returned object is a `pathlib.Path` object.

Parameters

- **request** (`_pytest.fixtures.FixtureRequest`) –
- **tmp_path_factory** (`_pytest.tmpdir.TempPathFactory`) –

Return type `pathlib.Path`

tmp_path_factory

The tmp_path_factory fixture

`tmp_path_factory` is an instance of `TempPathFactory`:

final class TempPathFactory

Factory for temporary directories under the common base temp directory.

The base directory can be configured using the `--basetemp` option.

Parameters

- **given_basetemp** (`Optional[pathlib.Path]`) –
- **basetemp** (`Optional[pathlib.Path]`) –
- **_ispytest** (`bool`) –

Return type `None`

mktemp(basename, numbered=True)

Create a new temporary directory managed by the factory.

Parameters

- **basename** (`str`) – Directory base name, must be a relative path.
- **numbered** (`bool`) – If `True`, ensure the directory is unique by adding a numbered suffix greater than any existing one: `basename="foo-"` and `numbered=True` means that this function will create directories named `"foo-0"`, `"foo-1"`, `"foo-2"` and so on.

Returns The path to the new directory.

Return type `pathlib.Path`

getbasetemp()

Return the base temporary directory, creating it if needed.

Return type `pathlib.Path`

tmpdir

The tmpdir and tmpdir_factory fixtures

tmpdir()

Return a temporary directory path object which is unique to each test function invocation, created as a sub directory of the base temporary directory.

By default, a new base temporary directory is created each test session, and old bases are removed after 3 sessions, to aid in debugging. If `--basetemp` is used then it is cleared each session. See *The default base temporary directory*.

The returned object is a `legacy_path` object.

Parameters `tmp_path` (`pathlib.Path`) –

Return type `py._path.local.LocalPath`

tmpdir_factory

The tmpdir and tmpdir_factory fixtures

`tmpdir_factory` is an instance of `TempdirFactory`:

final class TempdirFactory

Backward compatibility wrapper that implements `:class:_pytest.compat.LEGACY_PATH` for `:class:TempPathFactory`.

Parameters

- `tmppath_factory` (`_pytest.tmpdir.TempPathFactory`) –
- `_ispytest` (`bool`) –

Return type `None`

mktemp(basename, numbered=True)

Same as `TempPathFactory.mktemp()`, but returns a `_pytest.compat.LEGACY_PATH` object.

Parameters

- `basename` (`str`) –
- `numbered` (`bool`) –

Return type `py._path.local.LocalPath`

getbasetemp()

Backward compat wrapper for `_tmppath_factory.getbasetemp`.

Return type `py._path.local.LocalPath`

3.4.4 Hooks

Writing plugins.

Reference to all hooks which can be implemented by *conftest.py files* and *plugins*.

Bootstrapping hooks

Bootstrapping hooks called for plugins registered early enough (internal and setuptools plugins).

pytest_load_initial_conftests (*early_config, parser, args*)

Called to implement the loading of initial conftest files ahead of command line option parsing.

Note: This hook will not be called for `conftest.py` files, only for setuptools plugins.

Parameters

- **early_config** (`pytest.Config`) – The pytest config object.
- **args** (`List[str]`) – Arguments passed on the command line.
- **parser** (`pytest.Parser`) – To add command line options.

Return type `None`

pytest_cmdline_preparse (*config, args*)

(**Deprecated**) modify command line arguments before option parsing.

This hook is considered deprecated and will be removed in a future pytest version. Consider using `pytest_load_initial_conftests()` instead.

Note: This hook will not be called for `conftest.py` files, only for setuptools plugins.

Parameters

- **config** (`pytest.Config`) – The pytest config object.
- **args** (`List[str]`) – Arguments passed on the command line.

Return type `None`

pytest_cmdline_parse (*pluginmanager, args*)

Return an initialized config object, parsing the specified args.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Note: This hook will only be called for plugin classes passed to the `plugins` arg when using `pytest.main` to perform an in-process test run.

Parameters

- **pluginmanager** (`pytest.PytestPluginManager`) – The pytest plugin manager.
- **args** (`List[str]`) – List of arguments passed on the command line.

Return type `Optional[Config]`

pytest_cmdline_main (*config*)

Called for performing the main command line action. The default implementation will invoke the configure hooks and runtest_mainloop.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters **config** (`pytest.Config`) – The pytest config object.

Return type `Optional[Union[ExitCode, int]]`

Initialization hooks

Initialization hooks called for plugins and `conftest.py` files.

pytest_addoption (*parser, pluginmanager*)

Register argparse-style options and ini-style config values, called once at the beginning of a test run.

Note: This function should be implemented only in plugins or `conftest.py` files situated at the tests root directory due to how pytest *discovers plugins during startup*.

Parameters

- **parser** (`pytest.Parser`) – To add command line options, call `parser.addoption(...)`. To add ini-file values call `parser.addini(...)`.
- **pluginmanager** (`pytest.PytestPluginManager`) – The pytest plugin manager, which can be used to install `hookspec()`'s or `hookimpl()`'s and allow one plugin to call another plugin's hooks to change how command line options are added.

Return type `None`

Options can later be accessed through the `config` object, respectively:

- `config.getoption(name)` to retrieve the value of a command line option.
- `config.getini(name)` to retrieve a value read from an ini-style file.

The config object is passed around on many internal objects via the `.config` attribute or can be retrieved as the `pytestconfig` fixture.

Note: This hook is incompatible with `hookwrapper=True`.

pytest_addhooks (*pluginmanager*)

Called at plugin registration time to allow adding new hooks via a call to `pluginmanager.add_hookspecs(module_or_class, prefix)`.

Parameters **pluginmanager** (`pytest.PytestPluginManager`) – The pytest plugin manager.

Return type `None`

Note: This hook is incompatible with `hookwrapper=True`.

pytest_configure (*config*)

Allow plugins and `conftest` files to perform initial configuration.

This hook is called for every plugin and initial conftest file after command line options have been parsed. After that, the hook is called for other conftest files as they are imported.

Note: This hook is incompatible with `hookwrapper=True`.

Parameters `config` (`pytest.Config`) – The pytest config object.

Return type `None`

pytest_unconfigure (`config`)

Called before test process is exited.

Parameters `config` (`pytest.Config`) – The pytest config object.

Return type `None`

pytest_sessionstart (`session`)

Called after the `Session` object has been created and before performing collection and entering the run test loop.

Parameters `session` (`pytest.Session`) – The pytest session object.

Return type `None`

pytest_sessionfinish (`session`, `exitstatus`)

Called after whole test run finished, right before returning the exit status to the system.

Parameters

- `session` (`pytest.Session`) – The pytest session object.
- `exitstatus` (`int`) – The status which pytest will return to the system.

Return type `None`

pytest_plugin_registered (`plugin`, `manager`)

A new pytest plugin got registered.

Parameters

- `plugin` (`_PluggyPlugin`) – The plugin module or instance.
- `manager` (`pytest.PytestPluginManager`) – pytest plugin manager.

Return type `None`

Note: This hook is incompatible with `hookwrapper=True`.

Collection hooks

pytest calls the following hooks for collecting files and directories:

pytest_collection (`session`)

Perform the collection phase for the given session.

Stops at first non-None result, see *firstresult: stop at first non-None result*. The return value is not used, but only stops further processing.

The default collection phase is this (see individual hooks for full details):

1. Starting from `session` as the initial collector:

1. `pytest_collectstart`(collector)
 2. `report = pytest_make_collect_report`(collector)
 3. `pytest_exception_interact`(collector, call, report) if an interactive exception occurred
 4. For each collected node:
 1. If an item, `pytest_itemcollected`(item)
 2. If a collector, recurse into it.
 5. `pytest_collectreport`(report)
2. `pytest_collection_modifyitems`(session, config, items)
 1. `pytest_deselected`(items) for any deselected items (may be called multiple times)
 3. `pytest_collection_finish`(session)
 4. Set `session.items` to the list of collected items
 5. Set `session.testscollected` to the number of collected items

You can implement this hook to only perform some action before collection, for example the terminal plugin uses it to start displaying the collection counter (and returns `None`).

Parameters `session` (`pytest.Session`) – The pytest session object.

Return type Optional[object]

pytest_ignore_collect (*fspath*, *path*, *config*)

Return True to prevent considering this path for collection.

This hook is consulted for all files and directories prior to calling more specific hooks.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters

- **fspath** (`pathlib.Path`) – The path to analyze.
- **path** (`LEGACY_PATH`) – The path to analyze.
- **config** (`pytest.Config`) – The pytest config object.

Return type Optional[bool]

Changed in version 6.3.0: The `fspath` parameter was added as a `pathlib.Path` equivalent of the `path` parameter.

pytest_collect_file (*fspath*, *path*, *parent*)

Create a Collector for the given path, or None if not relevant.

The new node needs to have the specified `parent` as a parent.

Parameters

- **fspath** (`pathlib.Path`) – The path to analyze.
- **path** (`LEGACY_PATH`) – The path to collect.
- **parent** (`Collector`) –

Return type Optional[Collector]

Changed in version 6.3.0: The `fspath` parameter was added as a `pathlib.Path` equivalent of the `path` parameter.

pytest_pycollect_makemodule (*fspath, path, parent*)

Return a Module collector or None for the given path.

This hook will be called for each matching test module path. The `pytest_collect_file` hook needs to be used if you want to create test modules for files that do not match as a test module.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters

- **fspath** (*pathlib.Path*) – The path of the module to collect.
- **path** (*legacy_path*) – The path of the module to collect.

Return type Optional[*Module*]

Changed in version 6.3.0: The `fspath` parameter was added as a `pathlib.Path` equivalent of the `path` parameter.

For influencing the collection of objects in Python modules you can use the following hook:

pytest_pycollect_makeitem (*collector, name, obj*)

Return a custom item/collector for a Python object in a module, or None.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters

- **collector** (*PyCollector*) –
- **name** (*str*) –
- **obj** (*object*) –

Return type Union[None, *Item*, *Collector*, List[Union[*Item*, *Collector*]]]

pytest_generate_tests (*metafunc*)

Generate (multiple) parametrized calls to a test function.

Parameters *metafunc* (*Metafunc*) –

Return type None

pytest_make_parametrize_id (*config, val, argname*)

Return a user-friendly string representation of the given `val` that will be used by `@pytest.mark.parametrize` calls, or None if the hook doesn't know about `val`.

The parameter name is available as `argname`, if required.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters

- **config** (*pytest.Config*) – The pytest config object.
- **val** (*object*) – The parametrized value.
- **argname** (*str*) – The automatic parameter name produced by pytest.

Return type Optional[*str*]

After collection is complete, you can modify the order of items, delete or otherwise amend the test items:

pytest_collection_modifyitems (*session, config, items*)

Called after collection has been performed. May filter or re-order the items in-place.

Parameters

- **session** (`pytest.Session`) – The pytest session object.
- **config** (`pytest.Config`) – The pytest config object.
- **items** (`List [pytest.Item]`) – List of item objects.

Return type `None`

Note: If this hook is implemented in `conftest.py` files, it always receives all collected items, not only those under the `conftest.py` where it is implemented.

pytest_collection_finish (`session`)

Called after collection has been performed and modified.

Parameters **session** (`pytest.Session`) – The pytest session object.**Return type** `None`**Test running (runtest) hooks**All runtest related hooks receive a `pytest.Item` object.**pytest_runtestloop** (`session`)

Perform the main runtest loop (after collection finished).

The default hook implementation performs the runtest protocol for all items collected in the session (`session.items`), unless the collection failed or the `collectonly` pytest option is set.If at any point `pytest.exit()` is called, the loop is terminated immediately.If at any point `session.shouldfail` or `session.shouldstop` are set, the loop is terminated after the runtest protocol for the current item is finished.**Parameters** **session** (`pytest.Session`) – The pytest session object.**Return type** `Optional[object]`Stops at first non-None result, see *firstresult: stop at first non-None result*. The return value is not used, but only stops further processing.**pytest_runtest_protocol** (`item, nextitem`)

Perform the runtest protocol for a single test item.

The default runtest protocol is this (see individual hooks for full details):

- `pytest_runtest_logstart(nodeid, location)`
- **Setup phase:**
 - `call = pytest_runtest_setup(item)` (wrapped in `CallInfo(when="setup")`)
 - `report = pytest_runtest_makereport(item, call)`
 - `pytest_runtest_logreport(report)`
 - `pytest_exception_interact(call, report)` if an interactive exception occurred
- **Call phase, if the the setup passed and the `setuponly` pytest option is not set:**
 - `call = pytest_runtest_call(item)` (wrapped in `CallInfo(when="call")`)
 - `report = pytest_runtest_makereport(item, call)`

- `pytest_runtest_logreport`(report)
- `pytest_exception_interact`(call, report) if an interactive exception occurred

- **Teardown phase:**

- `call = pytest_runtest_teardown`(item, nextitem) (wrapped in `Call-Info`(when="teardown"))
- `report = pytest_runtest_makereport`(item, call)
- `pytest_runtest_logreport`(report)
- `pytest_exception_interact`(call, report) if an interactive exception occurred

- `pytest_runtest_logfinish`(nodeid, location)

Parameters

- **item** (`Item`) – Test item for which the runtest protocol is performed.
- **nextitem** (`Optional[Item]`) – The scheduled-to-be-next test item (or `None` if this is the end my friend).

Return type `Optional[object]`

Stops at first non-`None` result, see *firstresult: stop at first non-None result*. The return value is not used, but only stops further processing.

pytest_runtest_logstart (nodeid, location)

Called at the start of running the runtest protocol for a single item.

See *pytest_runtest_protocol()* for a description of the runtest protocol.

Parameters

- **nodeid** (`str`) – Full node ID of the item.
- **location** (`Tuple[str, Optional[int], str]`) – A tuple of (filename, lineno, testname).

Return type `None`

pytest_runtest_logfinish (nodeid, location)

Called at the end of running the runtest protocol for a single item.

See *pytest_runtest_protocol()* for a description of the runtest protocol.

Parameters

- **nodeid** (`str`) – Full node ID of the item.
- **location** (`Tuple[str, Optional[int], str]`) – A tuple of (filename, lineno, testname).

Return type `None`

pytest_runtest_setup (item)

Called to perform the setup phase for a test item.

The default implementation runs `setup()` on `item` and all of its parents (which haven't been setup yet). This includes obtaining the values of fixtures required by the item (which haven't been obtained yet).

Parameters **item** (`Item`) –

Return type `None`

pytest_runtest_call (*item*)

Called to run the test for test item (the call phase).

The default implementation calls `item.runtest()`.

Parameters `item` (`Item`) –

Return type `None`

pytest_runtest_teardown (*item, nextitem*)

Called to perform the teardown phase for a test item.

The default implementation runs the finalizers and calls `teardown()` on `item` and all of its parents (which need to be torn down). This includes running the teardown phase of fixtures required by the item (if they go out of scope).

Parameters

- **nextitem** (`Optional[Item]`) – The scheduled-to-be-next test item (None if no further test item is scheduled). This argument is used to perform exact teardowns, i.e. calling just enough finalizers so that `nextitem` only needs to call setup functions.
- **item** (`Item`) –

Return type `None`

pytest_runtest_makereport (*item, call*)

Called to create a `pytest.reports.TestReport` for each of the setup, call and teardown runttest phases of a test item.

See `pytest_runtest_protocol()` for a description of the runttest protocol.

Parameters

- **call** (`CallInfo[None]`) – The `CallInfo` for the phase.
- **item** (`Item`) –

Return type `Optional[TestReport]`

Stops at first non-None result, see *firstresult: stop at first non-None result*.

For deeper understanding you may look at the default implementation of these hooks in `pytest.runner` and maybe also in `pytest.pdb` which interacts with `pytest.capture` and its input/output capturing in order to immediately drop into interactive debugging when a test failure occurs.

pytest_pyfunc_call (*pyfuncitem*)

Call underlying test function.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters `pyfuncitem` (`Function`) –

Return type `Optional[object]`

Reporting hooks

Session related reporting hooks:

pytest_collectstart (*collector*)

Collector starts collecting.

Parameters `collector` (`Collector`) –

Return type `None`

pytest_make_collect_report (*collector*)

Perform `collector.collect()` and return a `CollectReport`.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters `collector` (`Collector`) –

Return type `Optional[CollectReport]`

pytest_itemcollected (*item*)

We just collected a test item.

Parameters `item` (`Item`) –

Return type `None`

pytest_collectreport (*report*)

Collector finished collecting.

Parameters `report` (`CollectReport`) –

Return type `None`

pytest_deselected (*items*)

Called for deselected test items, e.g. by keyword.

May be called multiple times.

Parameters `items` (`Sequence[Item]`) –

Return type `None`

pytest_report_header (*config, startpath, startdir*)

Return a string or list of strings to be displayed as header info for terminal reporting.

Parameters

- **config** (`pytest.Config`) – The pytest config object.
- **startpath** (`Path`) – The starting dir.
- **startdir** (`LEGACY_PATH`) – The starting dir.

Return type `Union[str, List[str]]`

Note: Lines returned by a plugin are displayed before those of plugins which ran before it. If you want to have your line(s) displayed first, use `trylast=True`.

Note: This function should be implemented only in plugins or `conftest.py` files situated at the tests root directory due to how pytest *discovers plugins during startup*.

Changed in version 6.3.0: The `startpath` parameter was added as a `pathlib.Path` equivalent of the `startdir` parameter.

pytest_report_collectionfinish (*config, startpath, startdir, items*)

Return a string or list of strings to be displayed after collection has finished successfully.

These strings will be displayed after the standard “collected X items” message.

New in version 3.2.

Parameters

- **config** (`pytest.Config`) – The pytest config object.
- **startpath** (`Path`) – The starting path.
- **startdir** (`LEGACY_PATH`) – The starting dir.
- **items** (`Sequence[Item]`) – List of pytest items that are going to be executed; this list should not be modified.

Return type `Union[str, List[str]]`

Note: Lines returned by a plugin are displayed before those of plugins which ran before it. If you want to have your line(s) displayed first, use `trylast=True`.

Changed in version 6.3.0: The `startpath` parameter was added as a `pathlib.Path` equivalent of the `startdir` parameter.

pytest_report_teststatus (*report, config*)

Return result-category, shortletter and verbose word for status reporting.

The result-category is a category in which to count the result, for example “passed”, “skipped”, “error” or the empty string.

The shortletter is shown as testing progresses, for example “.”, “s”, “E” or the empty string.

The verbose word is shown as testing progresses in verbose mode, for example “PASSED”, “SKIPPED”, “ERROR” or the empty string.

pytest may style these implicitly according to the report outcome. To provide explicit styling, return a tuple for the verbose word, for example `"rerun", "R", ("RERUN", {"yellow": True})`.

Parameters

- **report** (`Union[CollectReport, TestReport]`) – The report object whose status is to be returned.
- **config** (`pytest.Config`) – The pytest config object.

Return type `Tuple[str, str, Union[str, Mapping[str, bool]]]`

Stops at first non-None result, see *firstresult: stop at first non-None result*.

pytest_terminal_summary (*terminalreporter, exitstatus, config*)

Add a section to terminal summary reporting.

Parameters

- **terminalreporter** (`_pytest.terminal.TerminalReporter`) – The internal terminal reporter object.
- **exitstatus** (`int`) – The exit status that will be reported back to the OS.
- **config** (`pytest.Config`) – The pytest config object.

Return type `None`

New in version 4.2: The `config` parameter.

pytest_fixture_setup (*fixturedef*, *request*)

Perform fixture setup execution.

Returns The return value of the call to the fixture function.

Parameters

- **fixturedef** (`FixtureDef [Any]`) –
- **request** (`SubRequest`) –

Return type `Optional[object]`

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Note: If the fixture function returns None, other implementations of this hook function will continue to be called, according to the behavior of the *firstresult: stop at first non-None result* option.

pytest_fixture_post_finalizer (*fixturedef*, *request*)

Called after fixture teardown, but before the cache is cleared, so the fixture result `fixturedef. cached_result` is still available (not None).

Parameters

- **fixturedef** (`FixtureDef [Any]`) –
- **request** (`SubRequest`) –

Return type `None`

pytest_warning_captured (*warning_message*, *when*, *item*, *location*)

(Deprecated) Process a warning captured by the internal pytest warnings plugin.

Deprecated since version 6.0.

This hook is considered deprecated and will be removed in a future pytest version. Use `pytest_warning_recorded()` instead.

Parameters

- **warning_message** (`warnings.WarningMessage`) – The captured warning. This is the same object produced by `warnings.catch_warnings()`, and contains the same attributes as the parameters of `warnings.showwarning()`.
- **when** (`str`) – Indicates when the warning was captured. Possible values:
 - "config": during pytest configuration/initialization stage.
 - "collect": during test collection.
 - "runtest": during test execution.
- **item** (`pytest.Item/None`) – The item being executed if when is "runtest", otherwise None.
- **location** (`tuple`) – When available, holds information about the execution context of the captured warning (filename, linenumber, function). `function` evaluates to <module> when the execution context is at the module level.

Return type `None`

pytest_warning_recorded (*warning_message, when, nodeid, location*)

Process a warning captured by the internal pytest warnings plugin.

Parameters

- **warning_message** (*warnings.WarningMessage*) – The captured warning. This is the same object produced by `warnings.catch_warnings()`, and contains the same attributes as the parameters of `warnings.showwarning()`.
- **when** (*str*) – Indicates when the warning was captured. Possible values:
 - "config": during pytest configuration/initialization stage.
 - "collect": during test collection.
 - "runtest": during test execution.
- **nodeid** (*str*) – Full id of the item.
- **location** (*tuple|None*) – When available, holds information about the execution context of the captured warning (filename, linenumber, function). `function` evaluates to `<module>` when the execution context is at the module level.

Return type `None`

New in version 6.0.

Central hook for reporting about test execution:

pytest_runtest_logreport (*report*)

Process the `pytest.reports.TestReport` produced for each of the setup, call and teardown runtest phases of an item.

See `pytest_runtest_protocol()` for a description of the runtest protocol.

Parameters **report** (`TestReport`) –

Return type `None`

Assertion related hooks:

pytest_assertrepr_compare (*config, op, left, right*)

Return explanation for comparisons in failing assert expressions.

Return `None` for no custom explanation, otherwise return a list of strings. The strings will be joined by newlines but any newlines *in* a string will be escaped. Note that all but the first line will be indented slightly, the intention is for the first line to be a summary.

Parameters

- **config** (`pytest.Config`) – The pytest config object.
- **op** (*str*) –
- **left** (*object*) –
- **right** (*object*) –

Return type `Optional[List[str]]`

pytest_assertion_pass (*item, lineno, orig, expl*)

Called whenever an assertion passes.

New in version 5.0.

Use this hook to do some processing after a passing assertion. The original assertion information is available in the `orig` string and the pytest introspected assertion information is available in the `expl` string.

This hook must be explicitly enabled by the `enable_assertion_pass_hook` ini-file option:

```
[pytest]
enable_assertion_pass_hook=true
```

You need to **clean the .pyc** files in your project directory and interpreter libraries when enabling this option, as assertions will require to be re-written.

Parameters

- **item** (`pytest.Item`) – pytest item object of current test.
- **lineno** (`int`) – Line number of the assert statement.
- **orig** (`str`) – String with the original assertion.
- **expl** (`str`) – String with the assert explanation.

Return type `None`

Debugging/Interaction hooks

There are few hooks which can be used for special reporting or interaction with exceptions:

pytest_internalerror (`excrepr, excinfo`)

Called for internal errors.

Return True to suppress the fallback handling of printing an INTERNALERROR message directly to `sys.stderr`.

Parameters

- **excrepr** (`ExceptionRepr`) –
- **excinfo** (`ExceptionInfo[BaseException]`) –

Return type `Optional[bool]`

pytest_keyboard_interrupt (`excinfo`)

Called for keyboard interrupt.

Parameters **excinfo** (`ExceptionInfo[Union[KeyboardInterrupt, Exit]]`) –

Return type `None`

pytest_exception_interact (`node, call, report`)

Called when an exception was raised which can potentially be interactively handled.

May be called during collection (see `pytest_make_collect_report()`), in which case `report` is a `_pytest.reports.CollectReport`.

May be called during runtest of an item (see `pytest_runtest_protocol()`), in which case `report` is a `_pytest.reports.TestReport`.

This hook is not called if the exception that was raised is an internal exception like `skip.Exception`.

Parameters

- **node** (`Union[Item, Collector]`) –
- **call** (`CallInfo[Any]`) –
- **report** (`Union[CollectReport, TestReport]`) –

Return type `None`

`pytest_enter_pdb` (*config*, *pdb*)

Called upon `pdb.set_trace()`.

Can be used by plugins to take special action just before the python debugger enters interactive mode.

Parameters

- **config** (`pytest.Config`) – The pytest config object.
- **pdb** (`pdb.Pdb`) – The Pdb instance.

Return type `None`

3.4.5 Objects

Full reference to objects accessible from *fixtures* or *hooks*.

CallInfo

final class `CallInfo`

Result/Exception info of a function invocation.

excinfo: `Optional[_pytest._code.code.ExceptionInfo[BaseException]]`

The captured exception of the call, if it raised.

start: `float`

The system time when the call started, in seconds since the epoch.

stop: `float`

The system time when the call ended, in seconds since the epoch.

duration: `float`

The call duration, in seconds.

when: `Literal['collect', 'setup', 'call', 'teardown']`

The context of invocation: “collect”, “setup”, “call” or “teardown”.

property result: `_pytest.runner.TResult`

The return value of the call, if it didn't raise.

Can only be accessed if `excinfo` is `None`.

classmethod `from_call` (*func*, *when*, *reraise=None*)

Call `func`, wrapping the result in a `CallInfo`.

Parameters

- **func** (`Callable[[], TResult]`) – The function to call. Called without arguments.
- **when** (`Literal['collect', 'setup', 'call', 'teardown']`) – The phase in which the function is called.
- **reraise** (`Optional[Union[Type[BaseException], Tuple[Type[BaseException], ...]]]`) – Exception or exceptions that shall propagate if raised by the function, instead of being wrapped in the `CallInfo`.

Return type `CallInfo[TResult]`

Class

class Class

Bases: `_pytest.python.PyCollector`

Collector for test methods.

classmethod `from_parent` (*parent*, *, *name*, *obj=None*, ***kw*)
 The public constructor.

collect ()
 Return a list of children (items and collectors) for this collection node.

Return type `Iterable[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

Collector

class Collector

Bases: `_pytest.nodes.Node`

Collector instances create children through `collect()` and thus iteratively build a tree.

exception `CollectError`
 Bases: `Exception`

An error during collection, contains a custom message.

collect ()
 Return a list of children (items and collectors) for this collection node.

Return type `Iterable[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

repr_failure (*excinfo*)
 Return a representation of a collection failure.

Parameters `excinfo` (`_pytest._code.code.ExceptionInfo[BaseException]`)
 – Exception information for the failure.

Return type `Union[str, _pytest._code.code.TerminalRepr]`

CollectReport

final class CollectReport

Bases: `_pytest.reports.BaseReport`

Collection report object.

Parameters

- **nodeid** (*str*) –
- **outcome** (`Literal['passed', 'failed', 'skipped']`) –
- **longrepr** (`Union[None, _pytest._code.code.ExceptionInfo[BaseException], Tuple[str, int, str], str, _pytest._code.code.TerminalRepr]`) –
- **result** (`Optional[List[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]]`) –
- **sections** (`Iterable[Tuple[str, str]]`) –

Return type `None`

nodeid: `str`

Normalized collection nodeid.

outcome: `Literal['passed', 'failed', 'skipped']`

Test outcome, always one of “passed”, “failed”, “skipped”.

longrepr: `Union[None, _pytest._code.code.ExceptionInfo[BaseException],`

`Tuple[str, int, str], str, _pytest._code.code.TerminalRepr]`

None or a failure representation.

result

The collected items and collection nodes.

sections: `List[Tuple[str, str]]`

Tuples of `str` (`heading`, `content`) with extra information for the test report. Used by pytest to add text captured from `stdout`, `stderr`, and intercepted logging events. May be used by other plugins to add arbitrary information to reports.

property caplog: `str`

Return captured log lines, if log capturing is enabled.

New in version 3.5.

property capstderr: `str`

Return captured text from `stderr`, if capturing is enabled.

New in version 3.0.

property capstdout: `str`

Return captured text from `stdout`, if capturing is enabled.

New in version 3.0.

property count_towards_summary: `bool`

Experimental Whether this report should be counted towards the totals shown at the end of the test session: “1 passed, 1 failure, etc”.

Note: This function is considered **experimental**, so beware that it is subject to changes even in patch releases.

property head_line: `Optional[str]`

Experimental The head line shown with `longrepr` output for this report, more commonly during traceback representation during failures:

```
_____ Test.foo _____
```

In the example above, the `head_line` is “Test.foo”.

Note: This function is considered **experimental**, so beware that it is subject to changes even in patch releases.

property longreprtext: `str`

Read-only property that returns the full string representation of `longrepr`.

New in version 3.0.

Config

final class Config

Access to configuration values, pluginmanager and plugin hooks.

Parameters

- **pluginmanager** (`PytestPluginManager`) – A pytest PluginManager.
- **invocation_params** (`InvocationParams`) – Object containing parameters regarding the `pytest.main()` invocation.

Return type `None`

final class InvocationParams (args, plugins, dir)

Holds parameters passed during `pytest.main()`.

The object attributes are read-only.

New in version 5.1.

Note: Note that the environment variable `PYTEST_ADDOPTS` and the `addopts` ini option are handled by pytest, not being included in the `args` attribute.

Plugins accessing `InvocationParams` must be aware of that.

Parameters

- **args** (`Tuple[str, ...]`) –
- **plugins** (`Optional[Sequence[Union[str, object]]]`) –
- **dir** (`pathlib.Path`) –

Return type `None`

args: `Tuple[str, ...]`

The command-line arguments as passed to `pytest.main()`.

plugins: `Optional[Sequence[Union[str, object]]]`

Extra plugins, might be `None`.

dir: `pathlib.Path`

The directory from which `pytest.main()` was invoked.

option

Access to command line option as attributes.

Type `argparse.Namespace`

invocation_params

The parameters with which pytest was invoked.

Type `InvocationParams`

pluginmanager

The plugin manager handles plugin registration and hook invocation.

Type `PytestPluginManager`

stash

A place where plugins can store information on the config for their own use.

Type *Stash*

property invocation_dir: `py._path.local.LocalPath`

The directory from which pytest was invoked.

Prefer to use `invocation_params.dir`, which is a `pathlib.Path`.

Type LEGACY_PATH

property rootpath: `pathlib.Path`

The path to the `rootdir`.

Type `pathlib.Path`

New in version 6.1.

property rootdir: `py._path.local.LocalPath`

The path to the `rootdir`.

Prefer to use `rootpath`, which is a `pathlib.Path`.

Type LEGACY_PATH

property inipath: `Optional[pathlib.Path]`

The path to the `configfile`.

Type `Optional[pathlib.Path]`

New in version 6.1.

property inifile: `Optional[py._path.local.LocalPath]`

The path to the `configfile`.

Prefer to use `inipath`, which is a `pathlib.Path`.

Type `Optional[LEGACY_PATH]`

add_cleanup (*func*)

Add a function to be called when the config object gets out of use (usually coinciding with `pytest_unconfigure`).

Parameters `func` (`Callable[[], None]`) –

Return type `None`

classmethod fromdictargs (*option_dict*, *args*)

Constructor usable for subprocesses.

Return type `_pytest.config.Config`

issue_config_time_warning (*warning*, *stacklevel*)

Issue and handle a warning during the “configure” stage.

During `pytest_configure` we can’t capture warnings using the `catch_warnings_for_item` function because it is not possible to have hookwrappers around `pytest_configure`.

This function is mainly intended for plugins that need to issue warnings during `pytest_configure` (or similar stages).

Parameters

- **warning** (`Warning`) – The warning instance.
- **stacklevel** (`int`) – stacklevel forwarded to `warnings.warn`.

Return type `None`

addinvalue_line (*name*, *line*)

Add a line to an ini-file option. The option must have been declared but might not yet be set in which case the line becomes the first line in its value.

Parameters

- **name** (*str*) –
- **line** (*str*) –

Return type `None`

getini (*name*)

Return configuration value from an *ini file*.

If the specified name hasn't been registered through a prior `parser.addini` call (usually from a plugin), a `ValueError` is raised.

Parameters **name** (*str*) –

getoption (*name*, *default*=<NOTSET>, *skip*=*False*)

Return command line option value.

Parameters

- **name** (*str*) – Name of the option. You may also specify the literal `--OPT` option instead of the “dest” option name.
- **default** – Default value if no option of that name exists.
- **skip** (*bool*) – If True, raise `pytest.skip` if option does not exists or has a `None` value.

getvalue (*name*, *path*=*None*)

Deprecated, use `getoption()` instead.

Parameters **name** (*str*) –

getvalueorskip (*name*, *path*=*None*)

Deprecated, use `getoption(skip=True)` instead.

Parameters **name** (*str*) –

ExceptionInfo

final class `ExceptionInfo`

Wraps `sys.exc_info()` objects and offers help for navigating the traceback.

classmethod `from_exc_info` (*exc_info*, *exprinfo*=*None*)

Return an `ExceptionInfo` for an existing `exc_info` tuple.

Warning: Experimental API

Parameters

- **exprinfo** (*Optional[str]*) – A text string helping to determine if we should strip `AssertionError` from the output. Defaults to the exception message/`__str__()`.
- **exc_info** (*Tuple[Type[_pytest._code.code.E], _pytest._code.code.E, types.TracebackType]*) –

Return type `_pytest._code.code.ExceptionInfo[_pytest._code.code.E]`

classmethod from_current (*exprinfo=None*)
 Return an ExceptionInfo matching the current traceback.

Warning: Experimental API

Parameters **exprinfo** (*Optional[str]*) – A text string helping to determine if we should strip AssertionError from the output. Defaults to the exception message/`__str__()`.

Return type `_pytest._code.code.ExceptionInfo[BaseException]`

classmethod for_later ()
 Return an unfilled ExceptionInfo.

Return type `_pytest._code.code.ExceptionInfo[_pytest._code.code.E]`

fill_unfilled (*exc_info*)
 Fill an unfilled ExceptionInfo created with `for_later()`.

Parameters **exc_info** (*Tuple[Type[_pytest._code.code.E], _pytest._code.code.E, types.TracebackType]*) –

Return type `None`

property type: `Type[_pytest._code.code.E]`
 The exception class.

property value: `_pytest._code.code.E`
 The exception value.

property tb: `types.TracebackType`
 The exception raw traceback.

property typename: `str`
 The type name of the exception.

property traceback: `_pytest._code.code.Traceback`
 The traceback.

exonly (*tryshort=False*)
 Return the exception as a string.

When ‘tryshort’ resolves to True, and the exception is an AssertionError, only the actual exception part of the exception representation is returned (so ‘AssertionError: ‘ is removed from the beginning).

Parameters **tryshort** (*bool*) –

Return type `str`

errisinstance (*exc*)
 Return True if the exception is an instance of exc.

Consider using `isinstance(excinfo.value, exc)` instead.

Parameters **exc** (*Union[Type[BaseException], Tuple[Type[BaseException], ...]]*) –

Return type `bool`

getrepr (*showlocals=False, style='long', abspath=False, tbfilter=True, funcargs=False, truncate_locals=True, chain=True*)

Return str(able representation of this exception info.

Parameters

- **showlocals** (*bool*) – Show locals per traceback entry. Ignored if `style=="native"`.
- **style** (*str*) – long|short|no|native|value traceback style.
- **abspath** (*bool*) – If paths should be changed to absolute or left unchanged.
- **tbfilter** (*bool*) – Hide entries that contain a local variable `__traceback-hide__==True`. Ignored if `style=="native"`.
- **funcargs** (*bool*) – Show fixtures (“funcargs” for legacy purposes) per traceback entry.
- **truncate_locals** (*bool*) – With `showlocals==True`, make sure locals can be safely represented as strings.
- **chain** (*bool*) – If chained exceptions in Python 3 should be shown.

Return type Union[ReprExceptionInfo, ExceptionChainRepr]

Changed in version 3.9: Added the `chain` parameter.

match (*regexp*)

Check whether the regular expression `regexp` matches the string representation of the exception using `re.search()`.

If it matches `True` is returned, otherwise an `AssertionError` is raised.

Parameters `regexp` (Union[*str*, Pattern[*str*]]) –

Return type Literal[True]

ExitCode

final class ExitCode (*value*)

Encodes the valid exit codes by pytest.

Currently users and plugins may supply other exit codes as well.

New in version 5.0.

OK = 0

Tests passed.

TESTS_FAILED = 1

Tests failed.

INTERRUPTED = 2

pytest was interrupted.

INTERNAL_ERROR = 3

An internal error got in the way.

USAGE_ERROR = 4

pytest was misused.

NO_TESTS_COLLECTED = 5

pytest couldn't find tests.

File

class File

Bases: `_pytest.nodes.FSCollector`

Base class for collecting tests from a file.

Working with non-python tests.

FixtureDef

final class FixtureDef

Bases: `Generic[_pytest.fixtures.FixtureValue]`

A container for a factory definition.

property scope: `_ScopeName`

Scope string, one of “function”, “class”, “module”, “package”, “session”.

FSCollector

class FSCollector

Bases: `_pytest.nodes.Collector`

classmethod from_parent (*parent*, *, *fspath=None*, *path=None*, ***kw*)

The public constructor.

Parameters

- **fspath** (*Optional*[`py._path.local.LocalPath`]) –
- **path** (*Optional*[`pathlib.Path`]) –

Function

class Function

Bases: `_pytest.python.PyobjMixin`, `_pytest.nodes.Item`

An Item responsible for setting up and executing a Python test function.

param name: The full function name, including any decorations like those added by parametrization (`my_func[my_param]`).

param parent: The parent Node.

param config: The pytest Config object.

param callspec: If given, this is function has been parametrized and the callspec contains meta information about the parametrization.

param callobj: If given, the object which will be called when the Function is invoked, otherwise the callobj will be obtained from parent using `originalname`.

param keywords: Keywords bound to the function object for “-k” matching.

param session: The pytest Session object.

param fixtureinfo: Fixture information already resolved at this fixture node..

param originalname: The attribute name to use for accessing the underlying function object. Defaults to `name`. Set this if name is different from the original name, for example when it contains decorations like those added by parametrization (`my_func[my_param]`).

originalname

Original function name, without any decorations (for example parametrization adds a "[...]" suffix to function names), used to access the underlying function object from `parent` (in case `callobj` is not given explicitly).

New in version 3.0.

classmethod from_parent (*parent*, ***kw*)

The public constructor.

property function

Underlying python 'function' object.

runtest ()

Execute the underlying test function.

Return type `None`

repr_failure (*excinfo*)

Return a representation of a collection or test failure.

Parameters `excinfo` (`_pytest._code.code.ExceptionInfo` [`BaseException`])

– Exception information for the failure.

Return type `Union[str, _pytest._code.code.TerminalRepr]`

FunctionDefinition

class `FunctionDefinition`

Bases: `_pytest.python.Function`

This class is a step gap solution until we evolve to have actual function definition nodes and manage to get rid of `metafunc`.

runtest ()

Execute the underlying test function.

Return type `None`

setup ()

Execute the underlying test function.

Return type `None`

Item

class `Item`

Bases: `_pytest.nodes.Node`

A basic test invocation item.

Note that for a single function there might be multiple test invocation items.

user_properties: `List[Tuple[str, object]]`

A list of tuples (name, value) that holds user defined properties for this test.

add_report_section (*when, key, content*)

Add a new report section, similar to what's done internally to add stdout and stderr captured output:

```
item.add_report_section("call", "stdout", "report section contents")
```

Parameters

- **when** (*str*) – One of the possible capture states, "setup", "call", "teardown".
- **key** (*str*) – Name of the section, can be customized at will. Pytest uses "stdout" and "stderr" internally.
- **content** (*str*) – The full contents as a string.

Return type `None`

MarkDecorator

class MarkDecorator

A decorator for applying a mark on test functions and classes.

MarkDecorators are created with `pytest.mark`:

```
mark1 = pytest.mark.NAME # Simple MarkDecorator
mark2 = pytest.mark.NAME(name1=value) # Parametrized MarkDecorator
```

and can then be applied as decorators to test functions:

```
@mark2
def test_function():
    pass
```

When a `MarkDecorator` is called, it does the following:

1. If called with a single class as its only positional argument and no additional keyword arguments, it attaches the mark to the class so it gets applied automatically to all test cases found in that class.
2. If called with a single function as its only positional argument and no additional keyword arguments, it attaches the mark to the function, containing all the arguments already stored internally in the `MarkDecorator`.
3. When called in any other case, it returns a new `MarkDecorator` instance with the original `MarkDecorator`'s content updated with the arguments passed to this call.

Note: The rules above prevent a `MarkDecorator` from storing only a single function or class reference as its positional argument with no additional keyword or positional arguments. You can work around this by using `with_args()`.

Parameters

- **mark** (`_pytest.mark.structures.Mark`) –
- **_ispytest** (*bool*) –

Return type `None`

property name: `str`

Alias for `mark.name`.

property args: `Tuple[Any, ...]`

Alias for `mark.args`.

property `kwargs: Mapping[str, Any]`

Alias for `mark.kwargs`.

with_args (**args, **kwargs*)

Return a `MarkDecorator` with extra arguments added.

Unlike calling the `MarkDecorator`, `with_args()` can be used even if the sole argument is a callable/class.

Parameters

- **args** (*object*) –
- **kwargs** (*object*) –

Return type `_pytest.mark.structures.MarkDecorator`

MarkGenerator

final class `MarkGenerator`

Factory for `MarkDecorator` objects - exposed as a `pytest.mark` singleton instance.

Example:

```
import pytest

@pytest.mark.slowest
def test_function():
    pass
```

applies a 'slowest' `Mark` on `test_function`.

Parameters `_ispytest (bool)` –

Return type `None`

Mark

final class `Mark`

Parameters

- **name** (*str*) –
- **args** (*Tuple[Any, ...]*) –
- **kwargs** (*Mapping[str, Any]*) –
- **param_ids_from** (*Optional[Mark]*) –
- **param_ids_generated** (*Optional[Sequence[str]]*) –
- **_ispytest** (*bool*) –

Return type `None`

name: `str`

Name of the mark.

args: `Tuple[Any, ...]`

Positional arguments of the mark decorator.

kwargs: `Mapping[str, Any]`

Keyword arguments of the mark decorator.

combined_with (*other*)

Return a new Mark which is a combination of this Mark and another Mark.

Combines by appending args and merging kwargs.

Parameters *other* (*Mark*) – The mark to combine with.

Return type *Mark*

Metafunc

final class Metafunc

Objects passed to the `pytest_generate_tests` hook.

They help to inspect a test function and to generate tests according to test configuration or values specified in the class or module where a test function is defined.

Parameters

- **definition** (*FunctionDefinition*) –
- **fixtureinfo** (`pytest.fixtures.FuncFixtureInfo`) –
- **config** (`pytest.config.Config`) –
- **_ispytest** (*bool*) –

Return type *None*

definition

Access to the underlying `pytest.python.FunctionDefinition`.

config

Access to the `pytest.Config` object for the test session.

module

The module object where the test function is defined in.

function

Underlying Python test function.

fixturenames

Set of fixture names required by the test function.

cls

Class object where the test function is defined in or *None*.

parametrize (*argnames, argvalues, indirect=False, ids=None, scope=None, *, _param_mark=None*)

Add new invocations to the underlying test function using the list of *argvalues* for the given *argnames*. Parametrization is performed during the collection phase. If you need to setup expensive resources see about setting *indirect* to do it rather than at test setup time.

Can be called multiple times, in which case each call parametrizes all previous parametrizations, e.g.

```
unparametrized:      t
parametrize ["x", "y"]: t[x], t[y]
parametrize [1, 2]:  t[x-1], t[x-2], t[y-1], t[y-2]
```

Parameters

- **argnames** (*Union[str, List[str], Tuple[str, ...]]*) – A comma-separated string denoting one or more argument names, or a list/tuple of argument strings.

- **argvalues** (*Iterable[Union[_pytest.mark.structures.ParameterSet, Sequence[object], object]]*) – The list of argvalues determines how often a test is invoked with different argument values.

If only one argname was specified argvalues is a list of values. If N argnames were specified, argvalues must be a list of N-tuples, where each tuple-element specifies a value for its respective argname.

- **indirect** (*Union[bool, Sequence[str]]*) – A list of arguments’ names (subset of argnames) or a boolean. If True the list contains all names from the argnames. Each argvalue corresponding to an argname in this list will be passed as request.param to its respective argname fixture function so that it can perform more expensive setups during the setup phase of a test rather than at collection time.

- **ids** (*Optional[Union[Iterable[Union[None, str, float, int, bool]], Callable[[Any], Optional[object]]]]*) – Sequence of (or generator for) ids for argvalues, or a callable to return part of the id for each argvalue.

With sequences (and generators like `itertools.count()`) the returned ids should be of type string, int, float, bool, or None. They are mapped to the corresponding index in argvalues. None means to use the auto-generated id.

If it is a callable it will be called for each entry in argvalues, and the return value is used as part of the auto-generated id for the whole set (where parts are joined with dashes (“-“)). This is useful to provide more specific ids for certain items, e.g. dates. Returning None will use an auto-generated id.

If no ids are provided they will be generated automatically from the argvalues.

- **scope** (*Optional[_ScopeName]*) – If specified it denotes the scope of the parameters. The scope is used for grouping tests by parameter instances. It will also override any fixture-function defined scope, allowing to set a dynamic scope using test context or configuration.

- **_param_mark** (*Optional[_pytest.mark.structures.Mark]*) –

Return type None

Module

class Module

Bases: `_pytest.nodes.File`, `_pytest.python.PyCollector`

Collector for test classes and functions.

`collect()`

Return a list of children (items and collectors) for this collection node.

Return type `Iterable[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

Node

class Node

Base class for Collector and Item, the components of the test collection tree.

Collector subclasses have children; Items are leaf nodes.

name

A unique name within the scope of the parent node.

parent

The parent collector node.

path

Filesystem path where this node was collected from (can be None).

keywords: MutableMapping[str, Any]

Keywords/markers collected from all scopes.

own_markers: List[_pytest.mark.structures.Mark]

The marker objects belonging to this node.

extra_keyword_matches: Set[str]

Allow adding of extra keywords to use for matching.

stash

A place where plugins can store information on the node for their own use.

Type *Stash*

property fspath: py._path.local.LocalPath

(deprecated) returns a legacy_path copy of self.path

classmethod from_parent (parent, **kw)

Public constructor for Nodes.

This indirection got introduced in order to enable removing the fragile logic from the node constructors.

Subclasses can use `super().from_parent(...)` when overriding the construction.

Parameters `parent` (`_pytest.nodes.Node`) – The parent node of this Node.

property ihook

fspath-sensitive hook proxy used to call pytest hooks.

warn (warning)

Issue a warning for this Node.

Warnings will be displayed after the test session, unless explicitly suppressed.

Parameters `warning` (`Warning`) – The warning instance to issue.

Raises `ValueError` – If `warning` instance is not a subclass of `Warning`.

Return type `None`

Example usage:

```
node.warn(PytestWarning("some message"))
node.warn(UserWarning("some message"))
```

Changed in version 6.2: Any subclass of `Warning` is now accepted, rather than only `PytestWarning` subclasses.

property nodeid: `str`

A ::-separated string denoting its collection tree address.

listchain ()

Return list of all parent collectors up to self, starting from the root of collection tree.

Return type `List[_pytest.nodes.Node]`

add_marker (*marker*, *append=True*)

Dynamically add a marker object to the node.

Parameters

- **append** (*bool*) – Whether to append the marker, or prepend it.
- **marker** (`Union[str, _pytest.mark.structures.MarkDecorator]`) –

Return type `None`

iter_markers (*name=None*)

Iterate over all markers of the node.

Parameters **name** (`Optional[str]`) – If given, filter the results by the name attribute.

Return type `Iterator[_pytest.mark.structures.Mark]`

for ... in iter_markers_with_node (*name=None*)

Iterate over all markers of the node.

Parameters **name** (`Optional[str]`) – If given, filter the results by the name attribute.

Returns An iterator of (node, mark) tuples.

Return type `Iterator[Tuple[_pytest.nodes.Node, _pytest.mark.structures.Mark]]`

get_closest_marker (*name: str*) → `Optional[_pytest.mark.structures.Mark]`

get_closest_marker (*name: str, default: _pytest.mark.structures.Mark*) → `_pytest.mark.structures.Mark`

Return the first marker matching the name, from closest (for example function) to farther level (for example module level).

Parameters

- **default** – Fallback return value if no marker was found.
- **name** – Name to filter by.

listextrakeywords ()

Return a set of all extra keywords in self and any parents.

Return type `Set[str]`

addfinalizer (*fin*)

Register a function to be called when this node is finalized.

This method can only be called when this node is active in a setup chain, for example during `self.setup()`.

Parameters **fin** (`Callable[[], object]`) –

Return type `None`

getparent (*cls*)

Get the next parent node (including self) which is an instance of the given class.

Parameters **cls** (`Type[_pytest.nodes._NodeType]`) –

Return type `Optional[_pytest.nodes._NodeType]`

repr_failure (*excinfo, style=None*)

Return a representation of a collection or test failure.

Parameters

- **excinfo** (`_pytest._code.code.ExceptionInfo [BaseException]`) – Exception information for the failure.
- **style** (`Optional [_TracebackStyle]`) –

Return type `Union[str, _pytest._code.code.TerminalRepr]`

Parser

final class Parser

Parser for command line arguments and ini-file values.

Variables **extra_info** – Dict of generic param -> value to display in case there's an error processing the command line arguments.

Parameters

- **usage** (`Optional [str]`) –
- **processopt** (`Optional [Callable [[Argument], None]]`) –
- **_ispytest** (`bool`) –

Return type `None`

getgroup (*name, description="", after=None*)

Get (or create) a named option Group.

Name Name of the option group.

Description Long description for `-help` output.

After Name of another group, used for ordering `-help` output.

Parameters

- **name** (`str`) –
- **description** (`str`) –
- **after** (`Optional [str]`) –

Return type `_pytest.config.argparsing.OptionGroup`

The returned group object has an `addoption` method with the same signature as `parser.addoption` but will be shown in the respective group in the output of `pytest --help`.

addoption (**opts, **attrs*)

Register a command line option.

Opts Option names, can be short or long options.

Attrs Same attributes which the `add_argument()` function of the `argparse` library accepts.

Parameters

- **opts** (`str`) –
- **attrs** (`Any`) –

Return type `None`

After command line parsing, options are available on the `pytest` config object via `config.option.NAME` where `NAME` is usually set by passing a `dest` attribute, for example `addoption("--long", dest="NAME", ...)`.

parse_known_args (*args, namespace=None*)

Parse and return a namespace object with known arguments at this point.

Parameters

- **args** (*Sequence[Union[str, os.PathLike[str]]]*) –
- **namespace** (*Optional[argparse.Namespace]*) –

Return type `argparse.Namespace`

parse_known_and_unknown_args (*args, namespace=None*)

Parse and return a namespace object with known arguments, and the remaining arguments unknown at this point.

Parameters

- **args** (*Sequence[Union[str, os.PathLike[str]]]*) –
- **namespace** (*Optional[argparse.Namespace]*) –

Return type `Tuple[argparse.Namespace, List[str]]`

addini (*name, help, type=None, default=None*)

Register an ini-file option.

Name Name of the ini-variable.

Type Type of the variable. Can be:

- `string`: a string
- `bool`: a boolean
- `args`: a list of strings, separated as in a shell
- `linelist`: a list of strings, separated by line breaks
- `paths`: a list of `pathlib.Path`, separated as in a shell
- `pathlist`: a list of `py.path`, separated as in a shell

New in version 6.3: The `paths` variable type.

Defaults to `string` if `None` or not passed.

Default Default value if no ini-file option exists but is queried.

Parameters

- **name** (*str*) –
- **help** (*str*) –
- **type** (*Optional[Literal['string', 'paths', 'pathlist', 'args', 'linelist', 'bool']]*) –

Return type `None`

The value of ini-variables can be retrieved via a call to `config.getini(name)`.

OptionGroup

class OptionGroup

A group of options shown in its own section.

Parameters

- **name** (*str*) –
- **description** (*str*) –
- **parser** (*Optional*[`pytest.config.argparsing.Parser`]) –
- **_ispytest** (*bool*) –

Return type `None`

addoption(*optnames, **attrs)

Add an option to this group.

If a shortened version of a long option is specified, it will be suppressed in the help. `addoption('--twowords', '--two-words')` results in help showing `--two-words` only, but `--twowords` gets accepted **and** the automatic destination is in `args.twowords`.

Parameters

- **optnames** (*str*) –
- **attrs** (*Any*) –

Return type `None`

PytestPluginManager

final class PytestPluginManager

Bases: `pluggy._manager.PluginManager`

A `pluggy.PluginManager` with additional pytest-specific functionality:

- Loading plugins from the command line, `PYTEST_PLUGINS` env variable and `pytest_plugins` global variables found in plugins being loaded.
- `conftest.py` loading during start-up.

Return type `None`

parse_hookimpl_opts(plugin, name)

Parameters

- **plugin** (*object*) –
- **name** (*str*) –

parse_hookspec_opts(module_or_class, name)

Parameters **name** (*str*) –

register(plugin, name=None)

Register a plugin and return its canonical name or `None` if the name is blocked from registering. Raise a `ValueError` if the plugin is already registered.

Parameters

- **plugin** (*object*) –
- **name** (*Optional[str]*) –

Return type `Optional[str]`

getplugin (*name*)

Parameters **name** (*str*) –

hasplugin (*name*)

Return whether a plugin with the given name is registered.

Parameters **name** (*str*) –

Return type `bool`

import_plugin (*modname, consider_entry_points=False*)

Import a plugin with modname.

If `consider_entry_points` is `True`, entry point names are also considered to find a plugin.

Parameters

- **modname** (*str*) –
- **consider_entry_points** (*bool*) –

Return type `None`

add_hookcall_monitoring (*before, after*)

add before/after tracing functions for all hooks and return an undo function which, when called, will remove the added tracers.

`before(hook_name, hook_impls, kwargs)` will be called ahead of all hook calls and receive a hookcaller instance, a list of `HookImpl` instances and the keyword arguments for the hook call.

`after(outcome, hook_name, hook_impls, kwargs)` receives the same arguments as `before` but also a `pluggy._callers._Result` object which represents the result of the overall hook call.

add_hookspecs (*module_or_class*)

add new hook specifications defined in the given `module_or_class`. Functions are recognized if they have been decorated accordingly.

check_pending ()

Verify that all hooks which have not been verified against a hook specification are optional, otherwise raise `PluginValidationError`.

enable_tracing ()

enable tracing of hook calls and return an undo function.

get_canonical_name (*plugin*)

Return canonical name for a plugin object. Note that a plugin may be registered under a different name which was specified by the caller of `register(plugin, name)`. To obtain the name of an registered plugin use `get_name(plugin)` instead.

get_hookcallers (*plugin*)

get all hook callers for the specified plugin.

get_name (*plugin*)

Return name for registered plugin or `None` if not registered.

get_plugin (*name*)
Return a plugin or None for the given name.

get_plugins ()
return the set of registered plugins.

has_plugin (*name*)
Return True if a plugin with the given name is registered.

is_blocked (*name*)
return True if the given plugin name is blocked.

is_registered (*plugin*)
Return True if the plugin is already registered.

list_name_plugin ()
return list of name/plugin pairs.

list_plugin_distinfo ()
return list of distinfo/plugin tuples for all setuptools registered plugins.

load_setuptools_entrypoints (*group, name=None*)
Load modules from querying the specified setuptools group.

Parameters

- **group** (*str*) – entry point group to load plugins
- **name** (*str*) – if given, loads only plugins with the given name.

Return type `int`

Returns return the number of loaded plugins by this call.

set_blocked (*name*)
block registrations of the given name, unregister if already registered.

subset_hook_caller (*name, remove_plugins*)
Return a new `_hooks._HookCaller` instance for the named method which manages calls to all registered plugins except the ones from `remove_plugins`.

unregister (*plugin=None, name=None*)
unregister a plugin object and all its contained hook implementations from internal data structures.

Session

final class Session

Bases: `_pytest.nodes.FSCollector`

exception Interrupted

Bases: `KeyboardInterrupt`

Signals that the test run was interrupted.

exception Failed

Bases: `Exception`

Signals a stop as failed test run.

property startpath: pathlib.Path

The path from which pytest was invoked.

New in version 6.3.0.

property stardir: `py._path.local.LocalPath`

The path from which pytest was invoked.

Prefer to use `startpath` which is a `pathlib.Path`.

Type `LEGACY_PATH`

perform_collect (*args: Optional[Sequence[str]] = None, genitems: Literal[True] = True*) → `Sequence[_pytest.nodes.Item]`

perform_collect (*args: Optional[Sequence[str]] = None, genitems: bool = True*) → `Sequence[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

Perform the collection phase for this session.

This is called by the default `pytest_collection` hook implementation; see the documentation of this hook for more details. For testing purposes, it may also be called directly on a fresh `Session`.

This function normally recursively expands any collectors collected from the session to their items, and only items are returned. For testing purposes, this may be suppressed by passing `genitems=False`, in which case the return value contains these collectors unexpanded, and `session.items` is empty.

for ... in collect()

Return a list of children (items and collectors) for this collection node.

Return type `Iterator[Union[_pytest.nodes.Item, _pytest.nodes.Collector]]`

TestReport

final class TestReport

Bases: `_pytest.reports.BaseReport`

Basic test report object (also used for setup and teardown calls if they fail).

Parameters

- **nodeid** (*str*) –
- **location** (*Tuple[str, Optional[int], str]*) –
- **outcome** (*Literal['passed', 'failed', 'skipped']*) –
- **longrepr** (*Union[None, _pytest._code.code.ExceptionInfo[BaseException], Tuple[str, int, str], str, _pytest._code.code.TerminalRepr]*) –
- **when** (*Literal['setup', 'call', 'teardown']*) –
- **sections** (*Iterable[Tuple[str, str]]*) –
- **duration** (*float*) –
- **user_properties** (*Optional[Iterable[Tuple[str, object]]]*) –

Return type `None`

nodeid: `str`

Normalized collection nodeid.

location: `Optional[Tuple[str, Optional[int], str]]`

A (filesystempath, lineno, domaininfo) tuple indicating the actual location of a test item - it might be different from the collected one e.g. if a method is inherited from a different module.

keywords

A name -> value dictionary containing all keywords and markers associated with a test invocation.

outcome: `Literal['passed', 'failed', 'skipped']`

Test outcome, always one of “passed”, “failed”, “skipped”.

longrepr: `Union[None, _pytest._code.code.ExceptionInfo[BaseException],`

`Tuple[str, int, str], str, _pytest._code.code.TerminalRepr]`

None or a failure representation.

when: `Optional[str]`

One of ‘setup’, ‘call’, ‘teardown’ to indicate runtest phase.

user_properties

User properties is a list of tuples (name, value) that holds user defined properties of the test.

sections: `List[Tuple[str, str]]`

Tuples of str (heading, content) with extra information for the test report. Used by pytest to add text captured from `stdout`, `stderr`, and intercepted logging events. May be used by other plugins to add arbitrary information to reports.

duration

Time it took to run just the test.

classmethod from_item_and_call (*item, call*)

Create and fill a `TestReport` with standard item and call info.

Parameters

- **item** (`_pytest.nodes.Item`) –
- **call** (`CallInfo [None]`) –

Return type `TestReport`

property caplog: `str`

Return captured log lines, if log capturing is enabled.

New in version 3.5.

property capstderr: `str`

Return captured text from `stderr`, if capturing is enabled.

New in version 3.0.

property capstdout: `str`

Return captured text from `stdout`, if capturing is enabled.

New in version 3.0.

property count_towards_summary: `bool`

Experimental Whether this report should be counted towards the totals shown at the end of the test session: “1 passed, 1 failure, etc”.

Note: This function is considered **experimental**, so beware that it is subject to changes even in patch releases.

property head_line: `Optional[str]`

Experimental The head line shown with longrepr output for this report, more commonly during traceback representation during failures:

```
_____ Test.foo _____
```

In the example above, the `head_line` is “Test.foo”.

Note: This function is considered **experimental**, so beware that it is subject to changes even in patch releases.

property longreprtext: `str`

Read-only property that returns the full string representation of `longrepr`.

New in version 3.0.

`_Result`

Result object used within *hook wrappers*, see `_Result` in the *pluggy* documentation for more information.

Stash

`class Stash`

Stash is a type-safe heterogeneous mutable mapping that allows keys and value types to be defined separately from where it (the `Stash`) is created.

Usually you will be given an object which has a `Stash`, for example `Config` or a `Node`:

```
stash: Stash = some_object.stash
```

If a module or plugin wants to store data in this `Stash`, it creates `StashKeys` for its keys (at the module level):

```
# At the top-level of the module
some_str_key = StashKey[str]()
some_bool_key = StashKey[bool]()
```

To store information:

```
# Value type must match the key.
stash[some_str_key] = "value"
stash[some_bool_key] = True
```

To retrieve the information:

```
# The static type of some_str is str.
some_str = stash[some_str_key]
# The static type of some_bool is bool.
some_bool = stash[some_bool_key]
```

Return type `None`

`__setitem__` (*key*, *value*)

Set a value for key.

Parameters

- **key** (`_pytest.stash.StashKey[_pytest.stash.T]`) –
- **value** (`_pytest.stash.T`) –

Return type `None`

`__getitem__` (*key*)

Get the value for key.

Raises `KeyError` if the key wasn't set before.

Parameters `key` (`_pytest.stash.StashKey[_pytest.stash.T]`) –

Return type `_pytest.stash.T`

`get` (*key*, *default*)

Get the value for key, or return default if the key wasn't set before.

Parameters

- `key` (`_pytest.stash.StashKey[_pytest.stash.T]`) –
- `default` (`_pytest.stash.D`) –

Return type `Union[_pytest.stash.T, _pytest.stash.D]`

`setdefault` (*key*, *default*)

Return the value of key if already set, otherwise set the value of key to default and return default.

Parameters

- `key` (`_pytest.stash.StashKey[_pytest.stash.T]`) –
- `default` (`_pytest.stash.T`) –

Return type `_pytest.stash.T`

`__delitem__` (*key*)

Delete the value for key.

Raises `KeyError` if the key wasn't set before.

Parameters `key` (`_pytest.stash.StashKey[_pytest.stash.T]`) –

Return type `None`

`__contains__` (*key*)

Return whether key was set.

Parameters `key` (`_pytest.stash.StashKey[_pytest.stash.T]`) –

Return type `bool`

`__len__` ()

Return how many items exist in the stash.

Return type `int`

class `StashKey` (*args, **kws)

Bases: `Generic[_pytest.stash.T]`

`StashKey` is an object used as a key to a `Stash`.

A `StashKey` is associated with the type `T` of the value of the key.

A `StashKey` is unique and cannot conflict with another key.

3.4.6 Global Variables

pytest treats some global variables in a special manner when defined in a test module or `conftest.py` files.

`collect_ignore`

Tutorial: *Customizing test collection*

Can be declared in *conftest.py* files to exclude test directories or modules. Needs to be a list of paths (`str`, `pathlib.Path` or any `os.PathLike`).

```
collect_ignore = ["setup.py"]
```

`collect_ignore_glob`

Tutorial: *Customizing test collection*

Can be declared in *conftest.py* files to exclude test directories or modules with Unix shell-style wildcards. Needs to be `list[str]` where `str` can contain glob patterns.

```
collect_ignore_glob = ["*_ignore.py"]
```

`pytest_plugins`

Tutorial: *Requiring/Loading plugins in a test module or conftest file*

Can be declared at the **global** level in *test modules* and *conftest.py* files to register additional plugins. Can be either a `str` or `Sequence[str]`.

```
pytest_plugins = "myapp.testsupport.myplugin"
```

```
pytest_plugins = ("myapp.testsupport.tools", "myapp.testsupport.regression")
```

`pytestmark`

Tutorial: *Marking whole classes or modules*

Can be declared at the **global** level in *test modules* to apply one or more *marks* to all test functions and methods. Can be either a single mark or a list of marks (applied in left-to-right order).

```
import pytest

pytestmark = pytest.mark.webtest
```

```
import pytest

pytestmark = [pytest.mark.integration, pytest.mark.slow]
```

3.4.7 Environment Variables

Environment variables that can be used to change pytest's behavior.

`PYTEST_ADDOPTS`

This contains a command-line (parsed by the `py:mod:shlex` module) that will be **prepended** to the command line given by the user, see *Builtin configuration file options* for more information.

`PYTEST_CURRENT_TEST`

This is not meant to be set by users, but is set by pytest internally with the name of the current test so other processes can inspect it, see `PYTEST_CURRENT_TEST` *environment variable* for more information.

PYTEST_DEBUG

When set, pytest will print tracing and debug information.

PYTEST_DISABLE_PLUGIN_AUTOLOAD

When set, disables plugin auto-loading through setuptools entrypoints. Only explicitly specified plugins will be loaded.

PYTEST_PLUGINS

Contains comma-separated list of modules that should be loaded as plugins:

```
export PYTEST_PLUGINS=mymodule.plugin,xdist
```

PYTEST_THEME

Sets a *pygment style* to use for the code output.

PYTEST_THEME_MODE

Sets the `PYTEST_THEME` to be either *dark* or *light*.

PY_COLORS

When set to 1, pytest will use color in terminal output. When set to 0, pytest will not use color. `PY_COLORS` takes precedence over `NO_COLOR` and `FORCE_COLOR`.

NO_COLOR

When set (regardless of value), pytest will not use color in terminal output. `PY_COLORS` takes precedence over `NO_COLOR`, which takes precedence over `FORCE_COLOR`. See no-color.org for other libraries supporting this community standard.

FORCE_COLOR

When set (regardless of value), pytest will use color in terminal output. `PY_COLORS` and `NO_COLOR` take precedence over `FORCE_COLOR`.

3.4.8 Exceptions

final class UsageError

Bases: `Exception`

Error in pytest usage or invocation.

3.4.9 Warnings

Custom warnings generated in some situations such as improper usage or deprecated features.

class PytestWarning

Bases: `UserWarning`

Base class for all warnings emitted by pytest.

class PytestAssertRewriteWarning

Bases: `pytest.PytestWarning`

Warning emitted by the pytest assert rewrite module.

class PytestCacheWarning

Bases: `pytest.PytestWarning`

Warning emitted by the cache plugin in various situations.

class PytestCollectionWarning

Bases: `pytest.PytestWarning`

Warning emitted when pytest is not able to collect a file or symbol in a module.

class PytestConfigWarning

Bases: `pytest.PytestWarning`

Warning emitted for configuration issues.

class PytestDeprecationWarning

Bases: `pytest.PytestWarning`, `DeprecationWarning`

Warning class for features that will be removed in a future version.

class PytestExperimentalApiWarning

Bases: `pytest.PytestWarning`, `FutureWarning`

Warning category used to denote experiments in pytest.

Use sparingly as the API might change or even be removed completely in a future version.

class PytestUnhandledCoroutineWarning

Bases: `pytest.PytestWarning`

Warning emitted for an unhandled coroutine.

A coroutine was encountered when collecting test functions, but was not handled by any async-aware plugin. Coroutine test functions are not natively supported.

class PytestUnknownMarkWarning

Bases: `pytest.PytestWarning`

Warning emitted on use of unknown markers.

See *How to mark test functions with attributes* for details.

class PytestUnraisableExceptionWarning

Bases: `pytest.PytestWarning`

An unraisable exception was reported.

Unraisable exceptions are exceptions raised in `__del__` implementations and similar situations when the exception cannot be raised as normal.

class PytestUnhandledThreadExceptionWarning

Bases: `pytest.PytestWarning`

An unhandled exception occurred in a `Thread`.

Such exceptions don't propagate normally.

Consult the *Internal pytest warnings* section in the documentation for more information.

3.4.10 Configuration Options

Here is a list of builtin configuration options that may be written in a `pytest.ini`, `pyproject.toml`, `tox.ini` or `setup.cfg` file, usually located at the root of your repository. To see each file format in details, see [Configuration file formats](#).

Warning: Usage of `setup.cfg` is not recommended except for very simple use cases. `.cfg` files use a different parser than `pytest.ini` and `tox.ini` which might cause hard to track down problems. When possible, it is recommended to use the latter files, or `pyproject.toml`, to hold your pytest configuration.

Configuration options may be overwritten in the command-line by using `-o/--override-ini`, which can also be passed multiple times. The expected format is `name=value`. For example:

```
pytest -o console_output_style=classic -o cache_dir=/tmp/mycache
```

addopts

Add the specified OPTS to the set of command line arguments as if they had been specified by the user. Example: if you have this ini file content:

```
# content of pytest.ini
[pytest]
addopts = --maxfail=2 -rf # exit after 2 failures, report fail info
```

issuing `pytest test_hello.py` actually means:

```
pytest --maxfail=2 -rf test_hello.py
```

Default is to add no options.

cache_dir

Sets a directory where stores content of cache plugin. Default directory is `.pytest_cache` which is created in `rootdir`. Directory may be relative or absolute path. If setting relative path, then directory is created relative to `rootdir`. Additionally path may contain environment variables, that will be expanded. For more information about cache plugin please refer to [How to re-run failed tests and maintain state between test runs](#).

console_output_style

Sets the console output style while running tests:

- `classic`: classic pytest output.
- `progress`: like classic pytest output, but with a progress indicator.
- `count`: like progress, but shows progress as the number of tests completed instead of a percent.

The default is `progress`, but you can fallback to `classic` if you prefer or the new mode is causing unexpected problems:

```
# content of pytest.ini
[pytest]
console_output_style = classic
```

doctest_encoding

Default encoding to use to decode text files with docstrings. [See how pytest handles doctests](#).

doctest_optionflags

One or more doctest flag names from the standard `doctest` module. [See how pytest handles doctests](#).

empty_parameter_set_mark

Allows to pick the action for empty parametersets in parameterization

- `skip` skips tests with an empty parameterset (default)
- `xfail` marks tests with an empty parameterset as `xfail(run=False)`
- `fail_at_collect` raises an exception if `parametrize` collects an empty parameter set

```
# content of pytest.ini
[pytest]
empty_parameter_set_mark = xfail
```

Note: The default value of this option is planned to change to `xfail` in future releases as this is considered less error prone, see [#3155](#) for more details.

faulthandler_timeout

Dumps the tracebacks of all threads if a test takes longer than X seconds to run (including fixture setup and tear-down). Implemented using the `faulthandler.dump_traceback_later()` function, so all caveats there apply.

```
# content of pytest.ini
[pytest]
faulthandler_timeout=5
```

For more information please refer to `faulthandler`.

filterwarnings

Sets a list of filters and actions that should be taken for matched warnings. By default all warnings emitted during the test session will be displayed in a summary at the end of the test session.

```
# content of pytest.ini
[pytest]
filterwarnings =
    error
    ignore::DeprecationWarning
```

This tells pytest to ignore deprecation warnings and turn all other warnings into errors. For more information please refer to [How to capture warnings](#).

junit_duration_report

New in version 4.1.

Configures how durations are recorded into the JUnit XML report:

- `total` (the default): duration times reported include setup, call, and teardown times.
- `call`: duration times reported include only call times, excluding setup and teardown.

```
[pytest]
junit_duration_report = call
```

junit_family

New in version 4.2.

Changed in version 6.1: Default changed to `xunit2`.

Configures the format of the generated JUnit XML file. The possible options are:

- `xunit1` (or `legacy`): produces old style output, compatible with the `xunit 1.0` format.

- `xunit2`: produces `xunit 2.0 style output`, which should be more compatible with latest Jenkins versions. **This is the default.**

```
[pytest]
junit_family = xunit2
```

`junit_logging`

New in version 3.5.

Changed in version 5.4: `log`, `all`, `out-err` options added.

Configures if captured output should be written to the JUnit XML file. Valid values are:

- `log`: write only logging captured output.
- `system-out`: write captured stdout contents.
- `system-err`: write captured stderr contents.
- `out-err`: write both captured stdout and stderr contents.
- `all`: write captured logging, stdout and stderr contents.
- `no` (the default): no captured output is written.

```
[pytest]
junit_logging = system-out
```

`junit_log_passing_tests`

New in version 4.6.

If `junit_logging != "no"`, configures if the captured output should be written to the JUnit XML file for **passing** tests. Default is `True`.

```
[pytest]
junit_log_passing_tests = False
```

`junit_suite_name`

To set the name of the root test suite xml item, you can configure the `junit_suite_name` option in your config file:

```
[pytest]
junit_suite_name = my_suite
```

`log_auto_indent`

Allow selective auto-indentation of multiline log messages.

Supports command line option `--log-auto-indent [value]` and config option `log_auto_indent = [value]` to set the auto-indentation behavior for all logging.

[value] can be:

- `True` or “On” - Dynamically auto-indent multiline log messages
- `False` or “Off” or `0` - Do not auto-indent multiline log messages (the default behavior)
- `[positive integer]` - auto-indent multiline log messages by `[value]` spaces

```
[pytest]
log_auto_indent = False
```

Supports passing kwarg `extra={"auto_indent": [value]}` to calls to `logging.log()` to specify auto-indentation behavior for a specific entry in the log. `extra` kwarg overrides the value specified on the command line or in the config.

log_cli

Enable log display during test run (also known as “*live logging*”). The default is `False`.

```
[pytest]
log_cli = True
```

log_cli_date_format

Sets a `time.strftime()`-compatible string that will be used when formatting dates for live logging.

```
[pytest]
log_cli_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *Live Logs*.

log_cli_format

Sets a `logging`-compatible string used to format live logging messages.

```
[pytest]
log_cli_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *Live Logs*.

log_cli_level

Sets the minimum log message level that should be captured for live logging. The integer value or the names of the levels can be used.

```
[pytest]
log_cli_level = INFO
```

For more information, see *Live Logs*.

log_date_format

Sets a `time.strftime()`-compatible string that will be used when formatting dates for logging capture.

```
[pytest]
log_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *How to manage logging*.

log_file

Sets a file name relative to the `pytest.ini` file where log messages should be written to, in addition to the other logging facilities that are active.

```
[pytest]
log_file = logs/pytest-logs.txt
```

For more information, see *How to manage logging*.

log_file_date_format

Sets a `time.strftime()`-compatible string that will be used when formatting dates for the logging file.

```
[pytest]
log_file_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *How to manage logging*.

log_file_format

Sets a logging-compatible string used to format logging messages redirected to the logging file.

```
[pytest]
log_file_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *How to manage logging*.

log_file_level

Sets the minimum log message level that should be captured for the logging file. The integer value or the names of the levels can be used.

```
[pytest]
log_file_level = INFO
```

For more information, see *How to manage logging*.

log_format

Sets a logging-compatible string used to format captured logging messages.

```
[pytest]
log_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *How to manage logging*.

log_level

Sets the minimum log message level that should be captured for logging capture. The integer value or the names of the levels can be used.

```
[pytest]
log_level = INFO
```

For more information, see *How to manage logging*.

markers

When the `--strict-markers` or `--strict` command-line arguments are used, only known markers - defined in code by core pytest or some plugin - are allowed.

You can list additional markers in this setting to add them to the whitelist, in which case you probably want to add `--strict-markers` to `addopts` to avoid future regressions:

```
[pytest]
addopts = --strict-markers
markers =
    slow
    serial
```

Note: The use of `--strict-markers` is highly preferred. `--strict` was kept for backward compatibility only and may be confusing for others as it only applies to markers and not to other options.

minversion

Specifies a minimal pytest version required for running tests.

```
# content of pytest.ini
[pytest]
minversion = 3.0 # will fail if we run with pytest-2.8
```

norecursedirs

Set the directory basename patterns to avoid when recursing for test discovery. The individual (fnmatch-style) patterns are applied to the basename of a directory to decide if to recurse into it. Pattern matching characters:

```
*      matches everything
?      matches any single character
[seq]  matches any character in seq
[!seq] matches any char not in seq
```

Default patterns are `'*.egg', '.*', '_darcs', 'build', 'CVS', 'dist', 'node_modules', 'venv', '{arch}'`. Setting a `norecursedirs` replaces the default. Here is an example of how to avoid certain directories:

```
[pytest]
norecursedirs = .svn _build tmp*
```

This would tell `pytest` to not look into typical subversion or sphinx-build directories or into any `tmp` prefixed directory.

Additionally, `pytest` will attempt to intelligently identify and ignore a virtualenv by the presence of an activation script. Any directory deemed to be the root of a virtual environment will not be considered during test collection unless `--collect-in-virtualenv` is given. Note also that `norecursedirs` takes precedence over `--collect-in-virtualenv`; e.g. if you intend to run tests in a virtualenv with a base directory that matches `'.*'` you *must* override `norecursedirs` in addition to using the `--collect-in-virtualenv` flag.

python_classes

One or more name prefixes or glob-style patterns determining which classes are considered for test collection. Search for multiple glob patterns by adding a space between patterns. By default, `pytest` will consider any class prefixed with `Test` as a test collection. Here is an example of how to collect tests from classes that end in `Suite`:

```
[pytest]
python_classes = *Suite
```

Note that `unittest.TestCase` derived classes are always collected regardless of this option, as `unittest`'s own collection framework is used to collect those tests.

python_files

One or more Glob-style file patterns determining which python files are considered as test modules. Search for multiple glob patterns by adding a space between patterns:

```
[pytest]
python_files = test_*.py check_*.py example_*.py
```

Or one per line:

```
[pytest]
python_files =
    test_*.py
    check_*.py
    example_*.py
```

By default, files matching `test_*.py` and `*_test.py` will be considered test modules.

python_functions

One or more name prefixes or glob-patterns determining which test functions and methods are considered tests. Search for multiple glob patterns by adding a space between patterns. By default, `pytest` will consider any function prefixed with `test` as a test. Here is an example of how to collect test functions and methods that end in `_test`:

```
[pytest]
python_functions = *_test
```

Note that this has no effect on methods that live on a `unittest.TestCase` derived class, as `unittest`'s own collection framework is used to collect those tests.

See *Changing naming conventions* for more detailed examples.

pythonpath

Sets list of directories that should be added to the python search path. Directories will be added to the head of `sys.path`. Similar to the `PYTHONPATH` environment variable, the directories will be included in where Python will look for imported modules. Paths are relative to the *rootdir* directory. Directories remain in path for the duration of the test session.

```
[pytest]
pythonpath = src1 src2
```

required_plugins

A space separated list of plugins that must be present for pytest to run. Plugins can be listed with or without version specifiers directly following their name. Whitespace between different version specifiers is not allowed. If any one of the plugins is not found, emit an error.

```
[pytest]
required_plugins = pytest-django>=3.0.0,<4.0.0 pytest-html pytest-xdist>=1.0.0
```

testpaths

Sets list of directories that should be searched for tests when no specific directories, files or test ids are given in the command line when executing pytest from the *rootdir* directory. Useful when all project tests are in a known location to speed up test collection and to avoid picking up undesired tests by accident.

```
[pytest]
testpaths = testing doc
```

This tells pytest to only look for tests in `testing` and `doc` directories when executing from the root directory.

usefixtures

List of fixtures that will be applied to all test functions; this is semantically the same to apply the `@pytest.mark.usefixtures` marker to all test functions.

```
[pytest]
usefixtures =
    clean_db
```

xfail_strict

If set to `True`, tests marked with `@pytest.mark.xfail` that actually succeed will by default fail the test suite. For more information, see *strict parameter*.

```
[pytest]
xfail_strict = True
```

3.4.11 Command-line Flags

All the command-line flags can be obtained by running `pytest --help`:

```
$ pytest --help
usage: pytest [options] [file_or_dir] [file_or_dir] [...]

positional arguments:
  file_or_dir

general:
  -k EXPRESSION          only run tests which match the given substring
                        expression. An expression is a python evaluatable
                        expression where all names are substring-matched
                        against test names and their parent classes.
                        Example: -k 'test_method or test_other' matches all
                        test functions and classes whose name contains
                        'test_method' or 'test_other', while -k 'not
                        test_method' matches those that don't contain
                        'test_method' in their names. -k 'not test_method
                        and not test_other' will eliminate the matches.
                        Additionally keywords are matched to classes and
                        functions containing extra names in their
                        'extra_keyword_matches' set, as well as functions
                        which have names assigned directly to them. The
                        matching is case-insensitive.
  -m MARKEXPR           only run tests matching given mark expression.
                        For example: -m 'mark1 and not mark2'.
  --markers             show markers (builtin, plugin and per-project ones).
  -x, --exitfirst      exit instantly on first error or failed test.
  --fixtures, --funcargs
                        show available fixtures, sorted by plugin appearance
                        (fixtures with leading '_' are only shown with '-v')
  --fixtures-per-test  show fixtures per test
  --pdb               start the interactive Python debugger on errors or
                        KeyboardInterrupt.
  --pdbcls=module:classname
                        specify a custom interactive Python debugger for use
                        with --pdb. For example:
                        --pdbcls=Python.terminal.debugger:TerminalPdb
  --trace             Immediately break when running each test.
  --capture=method    per-test capturing method: one of fd|sys|no|tee-sys.
  -s                 shortcut for --capture=no.
  --runxfail         report the results of xfail tests as if they were
                        not marked
  --lf, --last-failed
                        rerun only the tests that failed at the last run (or
                        all if none failed)
  --ff, --failed-first
                        run all tests, but run the last failures first.
                        This may re-order tests and thus lead to repeated
                        fixture setup/teardown.
  --nf, --new-first  run tests from new files first, then the rest of the
                        tests sorted by file mtime
  --cache-show=[CACHESHOW]
                        show cache contents, don't perform collection or
                        tests. Optional argument: glob (default: '*').
  --cache-clear      remove all cache contents at start of test run.
  --lfnf={all,none}, --last-failed-no-failures={all,none}
                        which tests to run with no previously (known)
```

(continues on next page)

(continued from previous page)

```

failures.
--sw, --stepwise      exit on test failure and continue from last failing
                      test next time
--sw-skip, --stepwise-skip
                      ignore the first failing test but stop on the next
                      failing test.
                      implicitly enables --stepwise.

reporting:
--durations=N        show N slowest setup/test durations (N=0 for all).
--durations-min=N    Minimal duration in seconds for inclusion in slowest
                      list. Default 0.005
-v, --verbose        increase verbosity.
--no-header          disable header
--no-summary         disable summary
-q, --quiet          decrease verbosity.
--verbosity=VERBOSE set verbosity. Default is 0.
-r chars             show extra test summary info as specified by chars:
                      (f)ailed, (E)rror, (s)kipped, (x)failed, (X)passed,
                      (p)assed, (P)assed with output, (a)ll except passed
                      (p/P), or (A)ll. (w)arnings are enabled by default
                      (see --disable-warnings), 'N' can be used to reset
                      the list. (default: 'fE').
--disable-warnings, --disable-pytest-warnings
                      disable warnings summary
-l, --showlocals     show locals in tracebacks (disabled by default).
--tb=style           traceback print mode
                      (auto/long/short/line/native/no).
--show-capture={no,stdout,stderr,log,all}
                      Controls how captured stdout/stderr/log is shown on
                      failed tests. Default is 'all'.
--full-trace         don't cut any tracebacks (default is to cut).
--color=color        color terminal output (yes/no/auto).
--code-highlight={yes,no}
                      Whether code should be highlighted (only if --color
                      is also enabled)
--pastebin=mode      send failed|all info to bpaste.net pastebin service.
--junit-xml=path     create junit-xml style report file at given path.
--junit-prefix=str   prepend prefix to classnames in junit-xml output

pytest-warnings:
-W PYTHONWARNINGS, --pythonwarnings=PYTHONWARNINGS
                      set which warnings to report, see -W option of
                      python itself.
--maxfail=num        exit after first num failures or errors.
--strict-config      any warnings encountered while parsing the `pytest`
                      section of the configuration file raise errors.
--strict-markers     markers not registered in the `markers` section of
                      the configuration file raise errors.
--strict             (deprecated) alias to --strict-markers.
-c file             load configuration from `file` instead of trying to
                      locate one of the implicit configuration files.
--continue-on-collection-errors
                      Force test execution even if collection errors
                      occur.
--rootdir=ROOTDIR   Define root directory for tests. Can be relative
                      path: 'root_dir', './root_dir',

```

(continues on next page)

(continued from previous page)

```

        'root_dir/another_dir/'; absolute path:
        '/home/user/root_dir'; path with variables:
        '$HOME/root_dir'.

collection:
  --collect-only, --co    only collect tests, don't execute them.
  --pyargs                try to interpret all arguments as python packages.
  --ignore=path           ignore path during collection (multi-allowed).
  --ignore-glob=path      ignore path pattern during collection (multi-
                          allowed).
  --deselect=nodeid_prefix
                          deselect item (via node id prefix) during collection
                          (multi-allowed).
  --confcutdir=dir        only load conftest.py's relative to specified dir.
  --noconftest            Don't load any conftest.py files.
  --keep-duplicates       Keep duplicate tests.
  --collect-in-virtualenv
                          Don't ignore tests in a local virtualenv directory
  --import-mode={prepend,append,importlib}
                          prepend/append to sys.path when importing test
                          modules and conftest files, default is to prepend.
  --doctest-modules       run doctests in all .py modules
  --doctest-report={none,cdiff,ndiff,udiff,only_first_failure}
                          choose another output format for diffs on doctest
                          failure
  --doctest-glob=pat      doctests file matching pattern, default: test*.txt
  --doctest-ignore-import-errors
                          ignore doctest ImportError
  --doctest-continue-on-failure
                          for a given doctest, continue to run after the first
                          failure

test session debugging and configuration:
  --basetemp=dir          base temporary directory for this test run. (warning:
                          this directory is removed if it exists)
  -V, --version           display pytest version and information about
                          plugins. When given twice, also display information
                          about plugins.
  -h, --help              show help message and configuration info
  -p name                 early-load given plugin module name or entry point
                          (multi-allowed).
                          To avoid loading of plugins, use the `no:` prefix,
                          e.g. `no:doctest`.
  --trace-config           trace considerations of conftest.py files.
  --debug=[DEBUG_FILE_NAME]
                          store internal tracing debug information in this log
                          file.
                          This file is opened with 'w' and truncated as a
                          result, care advised.
                          Defaults to 'pytestdebug.log'.
  -o OVERRIDE_INI, --override-ini=OVERRIDE_INI
                          override ini option with "option=value" style, e.g.
                          `-o xfail_strict=True -o cache_dir=cache`.
  --assert=MODE           Control assertion debugging tools.
                          'plain' performs no assertion debugging.
                          'rewrite' (the default) rewrites assert statements
                          in test modules on import to provide assert

```

(continues on next page)

(continued from previous page)

```

expression information.
--setup-only          only setup fixtures, do not execute tests.
--setup-show         show setup of fixtures while executing tests.
--setup-plan         show what fixtures and tests would be executed but
                    don't execute anything.

logging:
--log-level=LEVEL    level of messages to catch/display.
                    Not set by default, so it depends on the root/parent
                    log handler's effective level, where it is "WARNING"
                    by default.
--log-format=LOG_FORMAT
                    log format as used by the logging module.
--log-date-format=LOG_DATE_FORMAT
                    log date format as used by the logging module.
--log-cli-level=LOG_CLI_LEVEL
                    cli logging level.
--log-cli-format=LOG_CLI_FORMAT
                    log format as used by the logging module.
--log-cli-date-format=LOG_CLI_DATE_FORMAT
                    log date format as used by the logging module.
--log-file=LOG_FILE  path to a file when logging will be written to.
--log-file-level=LOG_FILE_LEVEL
                    log file logging level.
--log-file-format=LOG_FILE_FORMAT
                    log format as used by the logging module.
--log-file-date-format=LOG_FILE_DATE_FORMAT
                    log date format as used by the logging module.
--log-auto-indent=LOG_AUTO_INDENT
                    Auto-indent multiline messages passed to the logging
                    module. Accepts true|on, false|off or an integer.

[pytest] ini-options in the first pytest.ini|tox.ini|setup.cfg file found:

markers (linelist):  markers for test functions
empty_parameter_set_mark (string):
                    default marker for empty parametersets
norecursedirs (args): directory patterns to avoid for recursion
testpaths (args):    directories to search for tests when no files or
                    directories are given in the command line.
filterwarnings (linelist):
                    Each line specifies a pattern for
                    warnings.filterwarnings. Processed after
                    -W/--pythonwarnings.
usefixtures (args):  list of default fixtures to be used with this
                    project
python_files (args): glob-style file patterns for Python test module
                    discovery
python_classes (args):
                    prefixes or glob names for Python test class
                    discovery
python_functions (args):
                    prefixes or glob names for Python test function and
                    method discovery
disable_test_id_escaping_and_forfeit_all_rights_to_community_support (bool):
                    disable string escape non-ascii characters, might
                    cause unwanted side effects(use at your own risk)

```

(continues on next page)

(continued from previous page)

```

console_output_style (string):
    console output: "classic", or with additional
    progress information ("progress" (percentage) |
    "count").
xfail_strict (bool): default for the strict parameter of xfail markers
    when not given explicitly (default: False)
enable_assertion_pass_hook (bool):
    Enables the pytest_assertion_pass hook. Make sure to
    delete any previously generated pyc cache files.
junit_suite_name (string):
    Test suite name for JUnit report
junit_logging (string):
    Write captured log messages to JUnit report: one of
    no|log|system-out|system-err|out-err|all
junit_log_passing_tests (bool):
    Capture log information for passing tests to JUnit
    report:
junit_duration_report (string):
    Duration time to report: one of total|call
junit_family (string):
    Emit XML for schema: one of legacy|xunit1|xunit2
doctest_optionflags (args):
    option flags for doctests
doctest_encoding (string):
    encoding used for doctest files
cache_dir (string): cache directory path.
log_level (string): default value for --log-level
log_format (string): default value for --log-format
log_date_format (string):
    default value for --log-date-format
log_cli (bool): enable log display during test run (also known as
    "live logging").
log_cli_level (string):
    default value for --log-cli-level
log_cli_format (string):
    default value for --log-cli-format
log_cli_date_format (string):
    default value for --log-cli-date-format
log_file (string): default value for --log-file
log_file_level (string):
    default value for --log-file-level
log_file_format (string):
    default value for --log-file-format
log_file_date_format (string):
    default value for --log-file-date-format
log_auto_indent (string):
    default value for --log-auto-indent
faulthandler_timeout (string):
    Dump the traceback of all threads if a test takes
    more than TIMEOUT seconds to finish.
addopts (args): extra command line options
minversion (string): minimally required pytest version
required_plugins (args):
    plugins that must be present for pytest to run

environment variables:
PYTEST_ADDOPTS          extra command line options

```

(continues on next page)

(continued from previous page)

```
PYTEST_PLUGINS          comma-separated plugins to load during startup
PYTEST_DISABLE_PLUGIN_AUTOLOAD set to disable plugin auto-loading
PYTEST_DEBUG           set to enable debug tracing of pytest's internals
```

```
to see available markers type: pytest --markers
to see available fixtures type: pytest --fixtures
(shown according to specified file_or_dir or current dir if not specified; fixtures_
↳with leading '_' are only shown with the '-v' option
```


EXPLANATION

4.1 Anatomy of a test

In the simplest terms, a test is meant to look at the result of a particular behavior, and make sure that result aligns with what you would expect. Behavior is not something that can be empirically measured, which is why writing tests can be challenging.

“Behavior” is the way in which some system **acts in response** to a particular situation and/or stimuli. But exactly *how* or *why* something is done is not quite as important as *what* was done.

You can think of a test as being broken down into four steps:

1. **Arrange**
2. **Act**
3. **Assert**
4. **Cleanup**

Arrange is where we prepare everything for our test. This means pretty much everything except for the “**act**”. It’s lining up the dominoes so that the **act** can do its thing in one, state-changing step. This can mean preparing objects, starting/killing services, entering records into a database, or even things like defining a URL to query, generating some credentials for a user that doesn’t exist yet, or just waiting for some process to finish.

Act is the singular, state-changing action that kicks off the **behavior** we want to test. This behavior is what carries out the changing of the state of the system under test (SUT), and it’s the resulting changed state that we can look at to make a judgement about the behavior. This typically takes the form of a function/method call.

Assert is where we look at that resulting state and check if it looks how we’d expect after the dust has settled. It’s where we gather evidence to say the behavior does or does not aligns with what we expect. The `assert` in our test is where we take that measurement/observation and apply our judgement to it. If something should be green, we’d say `assert thing == "green"`.

Cleanup is where the test picks up after itself, so other tests aren’t being accidentally influenced by it.

At its core, the test is ultimately the **act** and **assert** steps, with the **arrange** step only providing the context. **Behavior** exists between **act** and **assert**.

4.2 About fixtures

See also:

How to use fixtures

See also:

Fixtures reference

pytest fixtures are designed to be explicit, modular and scalable.

4.2.1 What fixtures are

In testing, a *fixture* provides a defined, reliable and consistent context for the tests. This could include environment (for example a database configured with known parameters) or content (such as a dataset).

Fixtures define the steps and data that constitute the *arrange* phase of a test (see *Anatomy of a test*). In pytest, they are functions you define that serve this purpose. They can also be used to define a test's *act* phase; this is a powerful technique for designing more complex tests.

The services, state, or other operating environments set up by fixtures are accessed by test functions through arguments. For each fixture used by a test function there is typically a parameter (named after the fixture) in the test function's definition.

We can tell pytest that a particular function is a fixture by decorating it with `@pytest.fixture`. Here's a simple example of what a fixture in pytest might look like:

```
import pytest

class Fruit:
    def __init__(self, name):
        self.name = name

    def __eq__(self, other):
        return self.name == other.name

@pytest.fixture
def my_fruit():
    return Fruit("apple")

@pytest.fixture
def fruit_basket(my_fruit):
    return [Fruit("banana"), my_fruit]

def test_my_fruit_in_basket(my_fruit, fruit_basket):
    assert my_fruit in fruit_basket
```

Tests don't have to be limited to a single fixture, either. They can depend on as many fixtures as you want, and fixtures can use other fixtures, as well. This is where pytest's fixture system really shines.

4.2.2 Improvements over xUnit-style setup/teardown functions

pytest fixtures offer dramatic improvements over the classic xUnit style of setup/teardown functions:

- fixtures have explicit names and are activated by declaring their use from test functions, modules, classes or whole projects.
- fixtures are implemented in a modular manner, as each fixture name triggers a *fixture function* which can itself use other fixtures.
- fixture management scales from simple unit to complex functional testing, allowing to parametrize fixtures and tests according to configuration and component options, or to re-use fixtures across function, class, module or whole test session scopes.
- teardown logic can be easily, and safely managed, no matter how many fixtures are used, without the need to carefully handle errors by hand or micromanage the order that cleanup steps are added.

In addition, pytest continues to support *How to implement xunit-style set-up*. You can mix both styles, moving incrementally from classic to new style, as you prefer. You can also start out from existing *unittest.TestCase style* or *nose based* projects.

4.2.3 Fixture errors

pytest does its best to put all the fixtures for a given test in a linear order so that it can see which fixture happens first, second, third, and so on. If an earlier fixture has a problem, though, and raises an exception, pytest will stop executing fixtures for that test and mark the test as having an error.

When a test is marked as having an error, it doesn't mean the test failed, though. It just means the test couldn't even be attempted because one of the things it depends on had a problem.

This is one reason why it's a good idea to cut out as many unnecessary dependencies as possible for a given test. That way a problem in something unrelated isn't causing us to have an incomplete picture of what may or may not have issues.

Here's a quick example to help explain:

```
import pytest

@pytest.fixture
def order():
    return []

@pytest.fixture
def append_first(order):
    order.append(1)

@pytest.fixture
def append_second(order, append_first):
    order.extend([2])

@pytest.fixture(autouse=True)
def append_third(order, append_second):
    order += [3]

def test_order(order):
    assert order == [1, 2, 3]
```

If, for whatever reason, `order.append(1)` had a bug and it raises an exception, we wouldn't be able to know if `order.extend([2])` or `order += [3]` would also have problems. After `append_first` throws an exception, pytest won't run any more fixtures for `test_order`, and it won't even try to run `test_order` itself. The only things that would've run would be `order` and `append_first`.

4.2.4 Sharing test data

If you want to make test data from files available to your tests, a good way to do this is by loading these data in a fixture for use by your tests. This makes use of the automatic caching mechanisms of pytest.

Another good approach is by adding the data files in the `tests` folder. There are also community plugins available to help to manage this aspect of testing, e.g. `pytest-datadir` and `pytest-datafiles`.

4.2.5 A note about fixture cleanup

pytest does not do any special processing for `SIGTERM` and `SIGQUIT` signals (`SIGINT` is handled naturally by the Python runtime via `KeyboardInterrupt`), so fixtures that manage external resources which are important to be cleared when the Python process is terminated (by those signals) might leak resources.

The reason pytest does not handle those signals to perform fixture cleanup is that signal handlers are global, and changing them might interfere with the code under execution.

If fixtures in your suite need special care regarding termination in those scenarios, see [this comment](#) in the issue tracker for a possible workaround.

4.3 Good Integration Practices

4.3.1 Install package with pip

For development, we recommend you use `venv` for virtual environments and `pip` for installing your application and any dependencies, as well as the `pytest` package itself. This ensures your code and dependencies are isolated from your system Python installation.

Next, place a `pyproject.toml` file in the root of your package:

```
[build-system]
requires = ["setuptools>=42", "wheel"]
build-backend = "setuptools.build_meta"
```

and a `setup.cfg` file containing your package's metadata with the following minimum content:

```
[metadata]
name = PACKAGENAME

[options]
packages = find:
```

where `PACKAGENAME` is the name of your package.

Note: If your pip version is older than 21.3, you'll also need a `setup.py` file:

```
from setuptools import setup

setup()
```

You can then install your package in “editable” mode by running from the same directory:

```
pip install -e .
```

which lets you change your source code (both tests and application) and rerun tests at will.

4.3.2 Conventions for Python test discovery

pytest implements the following standard test discovery:

- If no arguments are specified then collection starts from `testpaths` (if configured) or the current directory. Alternatively, command line arguments can be used in any combination of directories, file names or node ids.
- Recurse into directories, unless they match `norecursedirs`.
- In those directories, search for `test_*.py` or `*_test.py` files, imported by their *test package name*.
- From those files, collect test items:
 - test prefixed test functions or methods outside of class
 - test prefixed test functions or methods inside Test prefixed test classes (without an `__init__` method)

For examples of how to customize your test discovery [Changing standard \(Python\) test discovery](#).

Within Python modules, pytest also discovers tests using the standard `unittest.TestCase` subclassing technique.

4.3.3 Choosing a test layout / import rules

pytest supports two common test layouts:

Tests outside application code

Putting tests into an extra directory outside your actual application code might be useful if you have many functional tests or for other reasons want to keep tests separate from actual application code (often a good idea):

```
pyproject.toml
setup.cfg
mypkg/
  __init__.py
  app.py
  view.py
tests/
  test_app.py
  test_view.py
  ...
```

This has the following benefits:

- Your tests can run against an installed version after executing `pip install ..`

- Your tests can run against the local copy with an editable install after executing `pip install --editable ..`
- If you don't use an editable install and are relying on the fact that Python by default puts the current directory in `sys.path` to import your package, you can execute `python -m pytest` to execute the tests against the local copy directly, without using `pip`.

Note: See *Invoking `pytest` versus `python -m pytest`* for more information about the difference between calling `pytest` and `python -m pytest`.

Note that this scheme has a drawback if you are using `prepend import mode` (which is the default): your test files must have **unique names**, because `pytest` will import them as *top-level* modules since there are no packages to derive a full package name from. In other words, the test files in the example above will be imported as `test_app` and `test_view` top-level modules by adding `tests/` to `sys.path`.

If you need to have test modules with the same name, you might add `__init__.py` files to your `tests` folder and subfolders, changing them to packages:

```
pyproject.toml
setup.cfg
mypkg/
...
tests/
  __init__.py
  foo/
    __init__.py
    test_view.py
  bar/
    __init__.py
    test_view.py
```

Now `pytest` will load the modules as `tests.foo.test_view` and `tests.bar.test_view`, allowing you to have modules with the same name. But now this introduces a subtle problem: in order to load the test modules from the `tests` directory, `pytest` prepends the root of the repository to `sys.path`, which adds the side-effect that now `mypkg` is also importable.

This is problematic if you are using a tool like `tox` to test your package in a virtual environment, because you want to test the *installed* version of your package, not the local code from the repository.

In this situation, it is **strongly** suggested to use a `src` layout where application root package resides in a sub-directory of your root:

```
pyproject.toml
setup.cfg
src/
  mypkg/
    __init__.py
    app.py
    view.py
tests/
  __init__.py
  foo/
    __init__.py
    test_view.py
  bar/
    __init__.py
    test_view.py
```


This layout prevents a lot of common pitfalls and has many benefits, which are better explained in this excellent [blog post](#) by Ionel Cristian Mărieș.

Note: The new `--import-mode=importlib` (see *Import modes*) doesn't have any of the drawbacks above because `sys.path` is not changed when importing test modules, so users that run into this issue are strongly encouraged to try it and report if the new option works well for them.

The `src` directory layout is still strongly recommended however.

Tests as part of application code

Inlining test directories into your application package is useful if you have direct relation between tests and application modules and want to distribute them along with your application:

```
pyproject.toml
setup.cfg
mypkg/
  __init__.py
  app.py
  view.py
  test/
    __init__.py
    test_app.py
    test_view.py
    ...
```

In this scheme, it is easy to run your tests using the `--pyargs` option:

```
pytest --pyargs mypkg
```

`pytest` will discover where `mypkg` is installed and collect tests from there.

Note that this layout also works in conjunction with the `src` layout mentioned in the previous section.

Note: You can use namespace packages (PEP420) for your application but `pytest` will still perform *test package name* discovery based on the presence of `__init__.py` files. If you use one of the two recommended file system layouts above but leave away the `__init__.py` files from your directories, it should just work. From “inlined tests”, however, you will need to use absolute imports for getting at your application code.

Note: In `prepend` and `append` import-modes, if `pytest` finds a `"a/b/test_module.py"` test file while recursing into the filesystem it determines the import name as follows:

- determine `basedir`: this is the first “upward” (towards the root) directory not containing an `__init__.py`. If e.g. both `a` and `b` contain an `__init__.py` file then the parent directory of `a` will become the `basedir`.
- perform `sys.path.insert(0, basedir)` to make the test module importable under the fully qualified import name.
- import `a.b.test_module` where the path is determined by converting path separators `/` into `.”` characters. This means you must follow the convention of having directory and file names map directly to the import names.

The reason for this somewhat evolved importing technique is that in larger projects multiple test modules might import from each other and thus deriving a canonical import name helps to avoid surprises such as a test module getting imported twice.

With `--import-mode=importlib` things are less convoluted because pytest doesn't need to change `sys.path` or `sys.modules`, making things much less surprising.

4.3.4 tox

Once you are done with your work and want to make sure that your actual package passes all tests you may want to look into `tox`, the virtualenv test automation tool and its [pytest support](#). `tox` helps you to setup virtualenv environments with pre-defined dependencies and then executing a pre-configured test command with options. It will run tests against the installed package and not against your source code checkout, helping to detect packaging glitches.

4.3.5 Do not run via setuptools

Integration with setuptools is **not recommended**, i.e. you should not be using `python setup.py test` or `pytest-runner`, and may stop working in the future.

This is deprecated since it depends on deprecated features of setuptools and relies on features that break security mechanisms in pip. For example 'setup_requires' and 'tests_require' bypass `pip --require-hashes`. For more information and migration instructions, see the [pytest-runner notice](#). See also [pypa/setuptools#1684](#).

setuptools intends to [remove the test command](#).

4.4 Flaky tests

A “flaky” test is one that exhibits intermittent or sporadic failure, that seems to have non-deterministic behaviour. Sometimes it passes, sometimes it fails, and it's not clear why. This page discusses pytest features that can help and other general strategies for identifying, fixing or mitigating them.

4.4.1 Why flaky tests are a problem

Flaky tests are particularly troublesome when a continuous integration (CI) server is being used, so that all tests must pass before a new code change can be merged. If the test result is not a reliable signal – that a test failure means the code change broke the test – developers can become mistrustful of the test results, which can lead to overlooking genuine failures. It is also a source of wasted time as developers must re-run test suites and investigate spurious failures.

4.4.2 Potential root causes

System state

Broadly speaking, a flaky test indicates that the test relies on some system state that is not being appropriately controlled - the test environment is not sufficiently isolated. Higher level tests are more likely to be flaky as they rely on more state.

Flaky tests sometimes appear when a test suite is run in parallel (such as use of `pytest-xdist`). This can indicate a test is reliant on test ordering.

- Perhaps a different test is failing to clean up after itself and leaving behind data which causes the flaky test to fail.
- The flaky test is reliant on data from a previous test that doesn't clean up after itself, and in parallel runs that previous test is not always present
- Tests that modify global state typically cannot be run in parallel.

Overly strict assertion

Overly strict assertions can cause problems with floating point comparison as well as timing issues. `pytest.approx()` is useful here.

4.4.3 Pytest features

Xfail strict

`pytest.mark.xfail` with `strict=False` can be used to mark a test so that its failure does not cause the whole build to break. This could be considered like a manual quarantine, and is rather dangerous to use permanently.

PYTEST_CURRENT_TEST

`PYTEST_CURRENT_TEST` may be useful for figuring out “which test got stuck”. See `PYTEST_CURRENT_TEST` *environment variable* for more details.

Plugins

Rerunning any failed tests can mitigate the negative effects of flaky tests by giving them additional chances to pass, so that the overall build does not fail. Several pytest plugins support this:

- `flaky`
- `pytest-flakefinder` - [blog post](#)
- `pytest-rerunfailures`
- `pytest-replay`: This plugin helps to reproduce locally crashes or flaky tests observed during CI runs.

Plugins to deliberately randomize tests can help expose tests with state problems:

- `pytest-random-order`
- `pytest-randomly`

4.4.4 Other general strategies

Split up test suites

It can be common to split a single test suite into two, such as unit vs integration, and only use the unit test suite as a CI gate. This also helps keep build times manageable as high level tests tend to be slower. However, it means it does become possible for code that breaks the build to be merged, so extra vigilance is needed for monitoring the integration test results.

Video/screenshot on failure

For UI tests these are important for understanding what the state of the UI was when the test failed. `pytest-splinter` can be used with plugins like `pytest-bdd` and can [save a screenshot on test failure](#), which can help to isolate the cause.

Delete or rewrite the test

If the functionality is covered by other tests, perhaps the test can be removed. If not, perhaps it can be rewritten at a lower level which will remove the flakiness or make its source more apparent.

Quarantine

Mark Lapierre discusses the [Pros and Cons of Quarantined Tests](#) in a post from 2018.

CI tools that rerun on failure

Azure Pipelines (the Azure cloud CI/CD tool, formerly Visual Studio Team Services or VSTS) has a feature to [identify flaky tests](#) and rerun failed tests.

4.4.5 Research

This is a limited list, please submit an issue or pull request to expand it!

- Gao, Zebao, Yalan Liang, Myra B. Cohen, Atif M. Memon, and Zhen Wang. “Making system user interactive tests repeatable: When and what should we control?.” In *Software Engineering (ICSE), 2015 IEEE/ACM 37th IEEE International Conference on*, vol. 1, pp. 55-65. IEEE, 2015. [PDF](#)
- Palomba, Fabio, and Andy Zaidman. “Does refactoring of test smells induce fixing flaky tests?.” In *Software Maintenance and Evolution (ICSME), 2017 IEEE International Conference on*, pp. 1-12. IEEE, 2017. [PDF in Google Drive](#)
- Bell, Jonathan, Owolabi Legunsen, Michael Hilton, Lamyaa Eloussi, Tiffany Yung, and Darko Marinov. “DeFlaker: Automatically detecting flaky tests.” In *Proceedings of the 2018 International Conference on Software Engineering*. 2018. [PDF](#)

4.4.6 Resources

- [Eradicating Non-Determinism in Tests](#) by Martin Fowler, 2011
- [No more flaky tests on the Go team](#) by Pavan Sudarshan, 2012
- [The Build That Cried Broken: Building Trust in your Continuous Integration Tests talk \(video\)](#) by Angie Jones at SeleniumConf Austin 2017
- [Test and Code Podcast: Flaky Tests and How to Deal with Them](#) by Brian Okken and Anthony Shaw, 2018
- Microsoft:
 - [How we approach testing VSTS to enable continuous delivery](#) by Brian Harry MS, 2017
 - [Eliminating Flaky Tests blog and talk \(video\)](#) by Munil Shah, 2017
- Google:
 - [Flaky Tests at Google and How We Mitigate Them](#) by John Micco, 2016

– Where do Google’s flaky tests come from? by Jeff Listfield, 2017

4.5 pytest import mechanisms and `sys.path/PYTHONPATH`

4.5.1 Import modes

pytest as a testing framework needs to import test modules and `conftest.py` files for execution.

Importing files in Python (at least until recently) is a non-trivial processes, often requiring changing `sys.path`. Some aspects of the import process can be controlled through the `--import-mode` command-line flag, which can assume these values:

- `prepend` (default): the directory path containing each module will be inserted into the *beginning* of `sys.path` if not already there, and then imported with the `__import__` builtin.

This requires test module names to be unique when the test directory tree is not arranged in packages, because the modules will put in `sys.modules` after importing.

This is the classic mechanism, dating back from the time Python 2 was still supported.

- `append`: the directory containing each module is appended to the end of `sys.path` if not already there, and imported with `__import__`.

This better allows to run test modules against installed versions of a package even if the package under test has the same import root. For example:

```
testing/__init__.py
testing/test_pkg_under_test.py
pkg_under_test/
```

the tests will run against the installed version of `pkg_under_test` when `--import-mode=append` is used whereas with `prepend` they would pick up the local version. This kind of confusion is why we advocate for using *src* layouts.

Same as `prepend`, requires test module names to be unique when the test directory tree is not arranged in packages, because the modules will put in `sys.modules` after importing.

- `importlib`: new in `pytest-6.0`, this mode uses `importlib` to import test modules. This gives full control over the import process, and doesn’t require changing `sys.path`.

For this reason this doesn’t require test module names to be unique, but also makes test modules non-importable by each other.

We intend to make `importlib` the default in future releases, depending on feedback.

4.5.2 `prepend` and `append` import modes scenarios

Here’s a list of scenarios when using `prepend` or `append` import modes where `pytest` needs to change `sys.path` in order to import test modules or `conftest.py` files, and the issues users might encounter because of that.

Test modules / `conftest.py` files inside packages

Consider this file and directory layout:

```
root/
|- foo/
  |- __init__.py
  |- conftest.py
  |- bar/
    |- __init__.py
    |- tests/
      |- __init__.py
      |- test_foo.py
```

When executing:

```
pytest root/
```

pytest will find `foo/bar/tests/test_foo.py` and realize it is part of a package given that there's an `__init__.py` file in the same folder. It will then search upwards until it can find the last folder which still contains an `__init__.py` file in order to find the package `root` (in this case `foo/`). To load the module, it will insert `root/` to the front of `sys.path` (if not there already) in order to load `test_foo.py` as the *module* `foo.bar.tests.test_foo`.

The same logic applies to the `conftest.py` file: it will be imported as `foo.conftest` module.

Preserving the full package name is important when tests live in a package to avoid problems and allow test modules to have duplicated names. This is also discussed in details in *Conventions for Python test discovery*.

Standalone test modules / `conftest.py` files

Consider this file and directory layout:

```
root/
|- foo/
  |- conftest.py
  |- bar/
    |- tests/
      |- test_foo.py
```

When executing:

```
pytest root/
```

pytest will find `foo/bar/tests/test_foo.py` and realize it is NOT part of a package given that there's no `__init__.py` file in the same folder. It will then add `root/foo/bar/tests` to `sys.path` in order to import `test_foo.py` as the *module* `test_foo`. The same is done with the `conftest.py` file by adding `root/foo` to `sys.path` to import it as `conftest`.

For this reason this layout cannot have test modules with the same name, as they all will be imported in the global import namespace.

This is also discussed in details in *Conventions for Python test discovery*.

4.5.3 Invoking `pytest` versus `python -m pytest`

Running `pytest` with `pytest [...]` instead of `python -m pytest [...]` yields nearly equivalent behaviour, except that the latter will add the current directory to `sys.path`, which is standard `python` behavior.

See also *Calling `pytest` through `python -m pytest`*.

FURTHER TOPICS

5.1 Examples and customization tricks

Here is a (growing) list of examples. *Contact* us if you need more examples or have questions. Also take a look at the *comprehensive documentation* which contains many example snippets as well. Also, [pytest on stackoverflow.com](#) often comes with example answers.

For basic examples, see

- *Get Started* for basic introductory examples
- *How to write and report assertions in tests* for basic assertion examples
- *Fixtures* for basic fixture/setup examples
- *How to parametrize fixtures and test functions* for basic test function parametrization
- *How to use unittest-based tests with pytest* for basic unittest integration
- *How to run tests written for nose* for basic nosetests integration

The following examples aim at various use cases you might encounter.

5.1.1 Demo of Python failure reports with pytest

Here is a nice run of several failures and how `pytest` presents things:

```
assertion $ pytest failure_demo.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project/assertion
collected 44 items

failure_demo.py FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF [100%]

===== FAILURES =====
_____ test_generative[3-6] _____

param1 = 3, param2 = 6

    @pytest.mark.parametrize("param1, param2", [(3, 6)])
    def test_generative(param1, param2):
>     assert param1 * 2 < param2
E         assert (3 * 2) < 6
```

(continues on next page)

(continued from previous page)

```

failure_demo.py:19: AssertionError
_____ TestFailing.test_simple _____

self = <failure_demo.TestFailing object at 0xdeadbeef0001>

    def test_simple(self):
        def f():
            return 42

        def g():
            return 43

>     assert f() == g()
E     assert 42 == 43
E     + where 42 = <function TestFailing.test_simple.<locals>.f at 0xdeadbeef0002>
→ ()
E     + and 43 = <function TestFailing.test_simple.<locals>.g at 0xdeadbeef0003>
→ ()

failure_demo.py:30: AssertionError
_____ TestFailing.test_simple_multiline _____

self = <failure_demo.TestFailing object at 0xdeadbeef0004>

    def test_simple_multiline(self):
>     otherfunc_multi(42, 6 * 9)

failure_demo.py:33:
-----
a = 42, b = 54

    def otherfunc_multi(a, b):
>     assert a == b
E     assert 42 == 54

failure_demo.py:14: AssertionError
_____ TestFailing.test_not _____

self = <failure_demo.TestFailing object at 0xdeadbeef0005>

    def test_not(self):
        def f():
            return 42

>     assert not f()
E     assert not 42
E     + where 42 = <function TestFailing.test_not.<locals>.f at 0xdeadbeef0006>()

failure_demo.py:39: AssertionError
_____ TestSpecialisedExplanations.test_eq_text _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0007>

    def test_eq_text(self):
>     assert "spam" == "eggs"

```

(continues on next page)

(continued from previous page)

```

E      AssertionError: assert 'spam' == 'eggs'
E          - eggs
E          + spam

failure_demo.py:44: AssertionError
_____ TestSpecialisedExplanations.test_eq_similar_text _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0008>

    def test_eq_similar_text(self):
>     assert "foo 1 bar" == "foo 2 bar"
E      AssertionError: assert 'foo 1 bar' == 'foo 2 bar'
E          - foo 2 bar
E          ?     ^
E          + foo 1 bar
E          ?     ^

failure_demo.py:47: AssertionError
_____ TestSpecialisedExplanations.test_eq_multiline_text _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0009>

    def test_eq_multiline_text(self):
>     assert "foo\nspam\nbar" == "foo\neggs\nbar"
E      AssertionError: assert 'foo\nspam\nbar' == 'foo\neggs\nbar'
E          foo
E          - eggs
E          + spam
E          bar

failure_demo.py:50: AssertionError
_____ TestSpecialisedExplanations.test_eq_long_text _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef000a>

    def test_eq_long_text(self):
        a = "1" * 100 + "a" + "2" * 100
        b = "1" * 100 + "b" + "2" * 100
>     assert a == b
E      AssertionError: assert '111111111111...222222222222' == '111111111111...
↪222222222222'
E          Skipping 90 identical leading characters in diff, use -v to show
E          Skipping 91 identical trailing characters in diff, use -v to show
E          - 11111111111b222222222
E            ?             ^
E          + 11111111111a222222222
E            ?             ^

failure_demo.py:55: AssertionError
_____ TestSpecialisedExplanations.test_eq_long_text_multiline _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef000b>

    def test_eq_long_text_multiline(self):
        a = "1\n" * 100 + "a" + "2\n" * 100
        b = "1\n" * 100 + "b" + "2\n" * 100
>     assert a == b

```

(continues on next page)

(continued from previous page)

```

E      AssertionError: assert '1\n1\n1\n1\n...\n2\n2\n2\n2\n' == '1\n1\n1\n1\n...\n2\n
↪n2\n2\n2\n2\n'
E      Skipping 190 identical leading characters in diff, use -v to show
E      Skipping 191 identical trailing characters in diff, use -v to show
E      1
E      1
E      1
E      1
E      1...
E
E      ...Full output truncated (7 lines hidden), use '-vv' to show

failure_demo.py:60: AssertionError
_____ TestSpecialisedExplanations.test_eq_list _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef000c>

    def test_eq_list(self):
>     assert [0, 1, 2] == [0, 1, 3]
E     assert [0, 1, 2] == [0, 1, 3]
E     At index 2 diff: 2 != 3
E     Use -v to get the full diff

failure_demo.py:63: AssertionError
_____ TestSpecialisedExplanations.test_eq_list_long _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef000d>

    def test_eq_list_long(self):
        a = [0] * 100 + [1] + [3] * 100
        b = [0] * 100 + [2] + [3] * 100
>     assert a == b
E     assert [0, 0, 0, 0, 0, 0, ...] == [0, 0, 0, 0, 0, 0, ...]
E     At index 100 diff: 1 != 2
E     Use -v to get the full diff

failure_demo.py:68: AssertionError
_____ TestSpecialisedExplanations.test_eq_dict _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef000e>

    def test_eq_dict(self):
>     assert {"a": 0, "b": 1, "c": 0} == {"a": 0, "b": 2, "d": 0}
E     AssertionError: assert {'a': 0, 'b': 1, 'c': 0} == {'a': 0, 'b': 2, 'd': 0}
E     Omitting 1 identical items, use -vv to show
E     Differing items:
E     {'b': 1} != {'b': 2}
E     Left contains 1 more item:
E     {'c': 0}
E     Right contains 1 more item:
E     {'d': 0}...
E
E     ...Full output truncated (2 lines hidden), use '-vv' to show

failure_demo.py:71: AssertionError
_____ TestSpecialisedExplanations.test_eq_set _____

```

(continues on next page)

(continued from previous page)

```

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef000f>

    def test_eq_set(self):
>     assert {0, 10, 11, 12} == {0, 20, 21}
E     AssertionError: assert {0, 10, 11, 12} == {0, 20, 21}
E       Extra items in the left set:
E       10
E       11
E       12
E     Extra items in the right set:
E     20
E     21...
E
E     ...Full output truncated (2 lines hidden), use '-vv' to show

failure_demo.py:74: AssertionError
_____ TestSpecialisedExplanations.test_eq_longer_list _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0010>

    def test_eq_longer_list(self):
>     assert [1, 2] == [1, 2, 3]
E     assert [1, 2] == [1, 2, 3]
E       Right contains one more item: 3
E       Use -v to get the full diff

failure_demo.py:77: AssertionError
_____ TestSpecialisedExplanations.test_in_list _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0011>

    def test_in_list(self):
>     assert 1 in [0, 2, 3, 4, 5]
E     assert 1 in [0, 2, 3, 4, 5]

failure_demo.py:80: AssertionError
_____ TestSpecialisedExplanations.test_not_in_text_multiline _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0012>

    def test_not_in_text_multiline(self):
        text = "some multiline\ntext\nwhich\nincludes foo\nand a\ntail"
>     assert "foo" not in text
E     AssertionError: assert 'foo' not in 'some multil...and a\ntail'
E       'foo' is contained here:
E         some multiline
E         text
E         which
E         includes foo
E         ?           +++
E         and a...
E
E     ...Full output truncated (2 lines hidden), use '-vv' to show

failure_demo.py:84: AssertionError
_____ TestSpecialisedExplanations.test_not_in_text_single _____

```

(continues on next page)

(continued from previous page)

```

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0013>

    def test_not_in_text_single(self):
        text = "single foo line"
>       assert "foo" not in text
E       AssertionError: assert 'foo' not in 'single foo line'
E         'foo' is contained here:
E           single foo line
E         ?          +++

failure_demo.py:88: AssertionError
_____ TestSpecialisedExplanations.test_not_in_text_single_long _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0014>

    def test_not_in_text_single_long(self):
        text = "head " * 50 + "foo " + "tail " * 20
>       assert "foo" not in text
E       AssertionError: assert 'foo' not in 'head head h...l tail tail '
E         'foo' is contained here:
E           head head foo tail tail tail tail tail tail tail tail tail tail
↪tail tail tail tail tail tail tail tail
E         ?          +++

failure_demo.py:92: AssertionError
_____ TestSpecialisedExplanations.test_not_in_text_single_long_term _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0015>

    def test_not_in_text_single_long_term(self):
        text = "head " * 50 + "f" * 70 + "tail " * 20
>       assert "f" * 70 not in text
E       AssertionError: assert 'ffffffffffff...ffffffffffff' not in 'head head h...l
↪tail tail '
E         'ffffffffffff...ffffffffffff' is contained here:
E           head head
↪fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffftail tail
↪tail tail tail tail tail tail tail tail tail tail tail tail tail tail
↪tail tail
E         ?          _

↪+++++

failure_demo.py:96: AssertionError
_____ TestSpecialisedExplanations.test_eq_dataclass _____

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0016>

    def test_eq_dataclass(self):
        from dataclasses import dataclass

        @dataclass
        class Foo:
            a: int
            b: str

        left = Foo(1, "b")
        right = Foo(1, "c")

```

(continues on next page)

(continued from previous page)

```
>     assert left == right
E     AssertionError: assert TestSpecialis...oo(a=1, b='b') == TestSpecialis...
↳oo(a=1, b='c')
E
E     Omitting 1 identical items, use -vv to show
E     Differing attributes:
E     ['b']
E
E     Drill down into differing attribute b:
E     b: 'b' != 'c'...
E
E     ...Full output truncated (3 lines hidden), use '-vv' to show
```

```
failure_demo.py:108: AssertionError
_____ TestSpecialisedExplanations.test_eq_attrs _____
```

```
self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0017>
```

```
def test_eq_attrs(self):
    import attr
```

```
    @attr.s
    class Foo:
        a = attr.ib()
        b = attr.ib()
```

```
    left = Foo(1, "b")
    right = Foo(1, "c")
```

```
>     assert left == right
E     AssertionError: assert Foo(a=1, b='b') == Foo(a=1, b='c')
```

```
E
E     Omitting 1 identical items, use -vv to show
E     Differing attributes:
E     ['b']
E
E     Drill down into differing attribute b:
E     b: 'b' != 'c'...
E
E     ...Full output truncated (3 lines hidden), use '-vv' to show
```

```
failure_demo.py:120: AssertionError
_____ test_attribute _____
```

```
def test_attribute():
    class Foo:
        b = 1
```

```
    i = Foo()
```

```
>     assert i.b == 2
```

```
E     assert 1 == 2
```

```
E     + where 1 = <failure_demo.test_attribute.<locals>.Foo object at 0xdeadbeef0018>.b
↳0xdeadbeef0018>.b
```

```
failure_demo.py:128: AssertionError
_____ test_attribute_instance _____
```

```
def test_attribute_instance():
```

(continues on next page)

(continued from previous page)

```

class Foo:
    b = 1

>     assert Foo().b == 2
E     AssertionError: assert 1 == 2
E     + where 1 = <failure_demo.test_attribute_instance.<locals>.Foo object at 0xdeadbeef0019>.b
E     + where <failure_demo.test_attribute_instance.<locals>.Foo object at 0xdeadbeef0019> = <class 'failure_demo.test_attribute_instance.<locals>.Foo'>()

failure_demo.py:135: AssertionError
_____ test_attribute_failure _____

def test_attribute_failure():
    class Foo:
        def _get_b(self):
            raise Exception("Failed to get attrib")

        b = property(_get_b)

    i = Foo()
>     assert i.b == 2

failure_demo.py:146:
-----
self = <failure_demo.test_attribute_failure.<locals>.Foo object at 0xdeadbeef001a>

def _get_b(self):
>     raise Exception("Failed to get attrib")
E     Exception: Failed to get attrib

failure_demo.py:141: Exception
_____ test_attribute_multiple _____

def test_attribute_multiple():
    class Foo:
        b = 1

    class Bar:
        b = 2

>     assert Foo().b == Bar().b
E     AssertionError: assert 1 == 2
E     + where 1 = <failure_demo.test_attribute_multiple.<locals>.Foo object at 0xdeadbeef001b>.b
E     + where <failure_demo.test_attribute_multiple.<locals>.Foo object at 0xdeadbeef001b> = <class 'failure_demo.test_attribute_multiple.<locals>.Foo'>()
E     + and 2 = <failure_demo.test_attribute_multiple.<locals>.Bar object at 0xdeadbeef001c>.b
E     + where <failure_demo.test_attribute_multiple.<locals>.Bar object at 0xdeadbeef001c> = <class 'failure_demo.test_attribute_multiple.<locals>.Bar'>()

failure_demo.py:156: AssertionError
_____ TestRaises.test_raises _____

self = <failure_demo.TestRaises object at 0xdeadbeef001d>

```

(continues on next page)

(continued from previous page)

```

    def test_raises(self):
        s = "qwe"
>       raises(TypeError, int, s)
E       ValueError: invalid literal for int() with base 10: 'qwe'

failure_demo.py:166: ValueError
_____ TestRaises.test_raises_doesnt _____

self = <failure_demo.TestRaises object at 0xdeadbeef001e>

    def test_raises_doesnt(self):
>       raises(OSEError, int, "3")
E       Failed: DID NOT RAISE <class 'OSEError'>

failure_demo.py:169: Failed
_____ TestRaises.test_raise _____

self = <failure_demo.TestRaises object at 0xdeadbeef001f>

    def test_raise(self):
>       raise ValueError("demo error")
E       ValueError: demo error

failure_demo.py:172: ValueError
_____ TestRaises.test_tupleerror _____

self = <failure_demo.TestRaises object at 0xdeadbeef0020>

    def test_tupleerror(self):
>       a, b = [1] # NOQA
E       ValueError: not enough values to unpack (expected 2, got 1)

failure_demo.py:175: ValueError
_____ TestRaises.test_reinterpret_fails_with_print_for_the_fun_of_it _____

self = <failure_demo.TestRaises object at 0xdeadbeef0021>

    def test_reinterpret_fails_with_print_for_the_fun_of_it(self):
        items = [1, 2, 3]
        print(f"items is {items!r}")
>       a, b = items.pop()
E       TypeError: cannot unpack non-iterable int object

failure_demo.py:180: TypeError
----- Captured stdout call -----
items is [1, 2, 3]
_____ TestRaises.test_some_error _____

self = <failure_demo.TestRaises object at 0xdeadbeef0022>

    def test_some_error(self):
>       if namenotexi: # NOQA
E       NameError: name 'namenotexi' is not defined

failure_demo.py:183: NameError
_____ test_dynamic_compile_shows_nicely _____

```

(continues on next page)

(continued from previous page)

```

def test_dynamic_compile_shows_nicely():
    import importlib.util
    import sys

    src = "def foo():\n assert 1 == 0\n"
    name = "abc-123"
    spec = importlib.util.spec_from_loader(name, loader=None)
    module = importlib.util.module_from_spec(spec)
    code = compile(src, name, "exec")
    exec(code, module.__dict__)
    sys.modules[name] = module
>     module.foo()

failure_demo.py:202:
-----
>     ???
E     AssertionError

abc-123:2: AssertionError
----- TestMoreErrors.test_complex_error -----

self = <failure_demo.TestMoreErrors object at 0xdeadbeef0023>

    def test_complex_error(self):
        def f():
            return 44

            def g():
                return 43
>         somefunc(f(), g())

failure_demo.py:213:
-----
failure_demo.py:10: in somefunc
    otherfunc(x, y)
-----

a = 44, b = 43

    def otherfunc(a, b):
>         assert a == b
E         assert 44 == 43

failure_demo.py:6: AssertionError
----- TestMoreErrors.test_z1_unpack_error -----

self = <failure_demo.TestMoreErrors object at 0xdeadbeef0024>

    def test_z1_unpack_error(self):
        items = []
>         a, b = items
E         ValueError: not enough values to unpack (expected 2, got 0)

failure_demo.py:217: ValueError

```

(continues on next page)

(continued from previous page)

```

_____ TestMoreErrors.test_z2_type_error _____
self = <failure_demo.TestMoreErrors object at 0xdeadbeef0025>

    def test_z2_type_error(self):
        items = 3
>         a, b = items
E         TypeError: cannot unpack non-iterable int object

failure_demo.py:221: TypeError
_____ TestMoreErrors.test_startswith _____
self = <failure_demo.TestMoreErrors object at 0xdeadbeef0026>

    def test_startswith(self):
        s = "123"
        g = "456"
>         assert s.startswith(g)
E         AssertionError: assert False
E         + where False = <built-in method startswith of str object at 0xdeadbeef0027>
E         ↪ ('456')
E         + where <built-in method startswith of str object at 0xdeadbeef0027> =
E         ↪ '123'.startswith

failure_demo.py:226: AssertionError
_____ TestMoreErrors.test_startswith_nested _____
self = <failure_demo.TestMoreErrors object at 0xdeadbeef0028>

    def test_startswith_nested(self):
        def f():
            return "123"

        def g():
            return "456"

>         assert f().startswith(g())
E         AssertionError: assert False
E         + where False = <built-in method startswith of str object at 0xdeadbeef0027>
E         ↪ ('456')
E         + where <built-in method startswith of str object at 0xdeadbeef0027> =
E         ↪ '123'.startswith
E         + where '123' = <function TestMoreErrors.test_startswith_nested.<locals>
E         ↪ .f at 0xdeadbeef0029>()
E         + and '456' = <function TestMoreErrors.test_startswith_nested.<locals>
E         ↪ .g at 0xdeadbeef002a>()

failure_demo.py:235: AssertionError
_____ TestMoreErrors.test_global_func _____
self = <failure_demo.TestMoreErrors object at 0xdeadbeef002b>

    def test_global_func(self):
>         assert isinstance(globf(42), float)
E         assert False
E         + where False = isinstance(43, float)
E         + where 43 = globf(42)

```

(continues on next page)

(continued from previous page)

```
failure_demo.py:238: AssertionError
_____ TestMoreErrors.test_instance _____

self = <failure_demo.TestMoreErrors object at 0xdeadbeef002c>

    def test_instance(self):
        self.x = 6 * 7
>       assert self.x != 42
E       assert 42 != 42
E       + where 42 = <failure_demo.TestMoreErrors object at 0xdeadbeef002c>.x

failure_demo.py:242: AssertionError
_____ TestMoreErrors.test_compare _____

self = <failure_demo.TestMoreErrors object at 0xdeadbeef002d>

    def test_compare(self):
>       assert globf(10) < 5
E       assert 11 < 5
E       + where 11 = globf(10)

failure_demo.py:245: AssertionError
_____ TestMoreErrors.test_try_finally _____

self = <failure_demo.TestMoreErrors object at 0xdeadbeef002e>

    def test_try_finally(self):
        x = 1
        try:
>           assert x == 0
E           assert 1 == 0

failure_demo.py:250: AssertionError
_____ TestCustomAssertMsg.test_single_line _____

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef002f>

    def test_single_line(self):
        class A:
            a = 1

            b = 2
>           assert A.a == b, "A.a appears not to be b"
E           AssertionError: A.a appears not to be b
E           assert 1 == 2
E           + where 1 = <class 'failure_demo.TestCustomAssertMsg.test_single_line.
↪<locals>.A'>.a

failure_demo.py:261: AssertionError
_____ TestCustomAssertMsg.test_multiline _____

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef0030>

    def test_multiline(self):
        class A:
            a = 1
```

(continues on next page)

(continued from previous page)

```

    b = 2
>    assert (
        A.a == b
    ), "A.a appears not to be b\nor does not appear to be b\none of those"
E    AssertionError: A.a appears not to be b
E        or does not appear to be b
E        one of those
E    assert 1 == 2
E    + where 1 = <class 'failure_demo.TestCustomAssertMsg.test_multiline.<locals>
↪ .A'>.a

failure_demo.py:268: AssertionError
_____ TestCustomAssertMsg.test_custom_repr _____

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef0031>

    def test_custom_repr(self):
        class JSON:
            a = 1

            def __repr__(self):
                return "This is JSON\n{\n 'foo': 'bar'\n}"

        a = JSON()
        b = 2
>    assert a.a == b, a
E    AssertionError: This is JSON
E    {
E    'foo': 'bar'
E    }
E    assert 1 == 2
E    + where 1 = This is JSON\n{\n 'foo': 'bar'\n}.a

failure_demo.py:281: AssertionError
===== short test summary info =====
FAILED failure_demo.py::test_generative[3-6] - assert (3 * 2) < 6
FAILED failure_demo.py::TestFailing::test_simple - assert 42 == 43
FAILED failure_demo.py::TestFailing::test_simple_multiline - assert 42 == 54
FAILED failure_demo.py::TestFailing::test_not - assert not 42
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_text - Asser...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_similar_text
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_multiline_text
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_long_text - ...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_long_text_multiline
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_list - asser...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_list_long - ...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_dict - Asser...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_set - Assert...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_longer_list
FAILED failure_demo.py::TestSpecialisedExplanations::test_in_list - asser...
FAILED failure_demo.py::TestSpecialisedExplanations::test_not_in_text_multiline
FAILED failure_demo.py::TestSpecialisedExplanations::test_not_in_text_single
FAILED failure_demo.py::TestSpecialisedExplanations::test_not_in_text_single_long
FAILED failure_demo.py::TestSpecialisedExplanations::test_not_in_text_single_long_term
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_dataclass - ...
FAILED failure_demo.py::TestSpecialisedExplanations::test_eq_attrs - Asse...

```

(continues on next page)

(continued from previous page)

```

FAILED failure_demo.py::test_attribute - assert 1 == 2
FAILED failure_demo.py::test_attribute_instance - AssertionError: assert ...
FAILED failure_demo.py::test_attribute_failure - Exception: Failed to get...
FAILED failure_demo.py::test_attribute_multiple - AssertionError: assert ...
FAILED failure_demo.py::TestRaises::test_raises - ValueError: invalid lit...
FAILED failure_demo.py::TestRaises::test_raises_doesnt - Failed: DID NOT ...
FAILED failure_demo.py::TestRaises::test_raise - ValueError: demo error
FAILED failure_demo.py::TestRaises::test_tupleerror - ValueError: not eno...
FAILED failure_demo.py::TestRaises::test_reinterpret_fails_with_print_for_the_fun_of_
↪it
FAILED failure_demo.py::TestRaises::test_some_error - NameError: name 'na...
FAILED failure_demo.py::test_dynamic_compile_shows_nicely - AssertionError
FAILED failure_demo.py::TestMoreErrors::test_complex_error - assert 44 == 43
FAILED failure_demo.py::TestMoreErrors::test_z1_unpack_error - ValueError...
FAILED failure_demo.py::TestMoreErrors::test_z2_type_error - TypeError: c...
FAILED failure_demo.py::TestMoreErrors::test_startswith - AssertionError:...
FAILED failure_demo.py::TestMoreErrors::test_startswith_nested - Assertio...
FAILED failure_demo.py::TestMoreErrors::test_global_func - assert False
FAILED failure_demo.py::TestMoreErrors::test_instance - assert 42 != 42
FAILED failure_demo.py::TestMoreErrors::test_compare - assert 11 < 5
FAILED failure_demo.py::TestMoreErrors::test_try_finally - assert 1 == 0
FAILED failure_demo.py::TestCustomAssertMsg::test_single_line - Assertion...
FAILED failure_demo.py::TestCustomAssertMsg::test_multiline - AssertionEr...
FAILED failure_demo.py::TestCustomAssertMsg::test_custom_repr - Assertion...
===== 44 failed in 0.12s =====

```

5.1.2 Basic patterns and examples

How to change command line options defaults

It can be tedious to type the same series of command line options every time you use `pytest`. For example, if you always want to see detailed info on skipped and xfailed tests, as well as have terser “dot” progress output, you can write it into a configuration file:

```

# content of pytest.ini
[pytest]
addopts = -ra -q

```

Alternatively, you can set a `PYTEST_ADDOPTS` environment variable to add command line options while the environment is in use:

```

export PYTEST_ADDOPTS="-v"

```

Here’s how the command-line is built in the presence of `addopts` or the environment variable:

```

<pytest.ini:addopts> $PYTEST_ADDOPTS <extra command-line arguments>

```

So if the user executes in the command-line:

```

pytest -m slow

```

The actual command line executed is:

```

pytest -ra -q -v -m slow

```

Note that as usual for other command-line applications, in case of conflicting options the last one wins, so the example above will show verbose output because `-v` overwrites `-q`.

Pass different values to a test function, depending on command line options

Suppose we want to write a test that depends on a command line option. Here is a basic pattern to achieve this:

```
# content of test_sample.py
def test_answer(cmdopt):
    if cmdopt == "type1":
        print("first")
    elif cmdopt == "type2":
        print("second")
    assert 0 # to see what was printed
```

For this to work we need to add a command line option and provide the `cmdopt` through a *fixture function*:

```
# content of confest.py
import pytest

def pytest_addoption(parser):
    parser.addoption(
        "--cmdopt", action="store", default="type1", help="my option: type1 or type2"
    )

@pytest.fixture
def cmdopt(request):
    return request.config.getoption("--cmdopt")
```

Let's run this without supplying our new option:

```
$ pytest -q test_sample.py
F [100%]
===== FAILURES =====
_____ test_answer _____

cmdopt = 'type1'

    def test_answer(cmdopt):
        if cmdopt == "type1":
            print("first")
        elif cmdopt == "type2":
            print("second")
>     assert 0 # to see what was printed
E     assert 0

test_sample.py:6: AssertionError
----- Captured stdout call -----
first
===== short test summary info =====
FAILED test_sample.py::test_answer - assert 0
1 failed in 0.12s
```

And now with supplying a command line option:

```

$ pytest -q --cmdopt=type2
F [100%]
===== FAILURES =====
_____ test_answer _____

cmdopt = 'type2'

    def test_answer(cmdopt):
        if cmdopt == "type1":
            print("first")
        elif cmdopt == "type2":
            print("second")
>       assert 0 # to see what was printed
E       assert 0

test_sample.py:6: AssertionError
----- Captured stdout call -----
second
===== short test summary info =====
FAILED test_sample.py::test_answer - assert 0
1 failed in 0.12s

```

You can see that the command line option arrived in our test.

We could add simple validation for the input by listing the choices:

```

# content of confptest.py
import pytest

def pytest_addoption(parser):
    parser.addoption(
        "--cmdopt",
        action="store",
        default="type1",
        help="my option: type1 or type2",
        choices=("type1", "type2"),
    )

```

Now we'll get feedback on a bad argument:

```

$ pytest -q --cmdopt=type3
ERROR: usage: pytest [options] [file_or_dir] [file_or_dir] [...]
pytest: error: argument --cmdopt: invalid choice: 'type3' (choose from 'type1', 'type2'
→)

```

If you need to provide more detailed error messages, you can use the `type` parameter and raise `pytest.UsageError`:

```

# content of confptest.py
import pytest

def type_checker(value):
    msg = "cmdopt must specify a numeric type as typeNNN"
    if not value.startswith("type"):
        raise pytest.UsageError(msg)
    try:
        int(value[4:])

```

(continues on next page)

(continued from previous page)

```

except ValueError:
    raise pytest.UsageError(msg)

return value

def pytest_addoption(parser):
    parser.addoption(
        "--cmdopt",
        action="store",
        default="type1",
        help="my option: type1 or type2",
        type=type_checker,
    )

```

This completes the basic pattern. However, one often rather wants to process command line options outside of the test and rather pass in different or more complex objects.

Dynamically adding command line options

Through *addopts* you can statically add command line options for your project. You can also dynamically modify the command line arguments before they get processed:

```

# setuptools plugin
import sys

def pytest_load_initial_conftests(args):
    if "xdist" in sys.modules: # pytest-xdist plugin
        import multiprocessing

        num = max(multiprocessing.cpu_count() / 2, 1)
        args[:] = ["-n", str(num)] + args

```

If you have the *xdist* plugin installed you will now always perform test runs using a number of subprocesses close to your CPU. Running in an empty directory with the above *conftest.py*:

```

$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 0 items

===== no tests ran in 0.12s =====

```

Control skipping of tests according to command line option

Here is a `confptest.py` file adding a `--runslow` command line option to control skipping of `pytest.mark.slow` marked tests:

```
# content of confptest.py

import pytest

def pytest_addoption(parser):
    parser.addoption(
        "--runslow", action="store_true", default=False, help="run slow tests"
    )

def pytest_configure(config):
    config.addinvalue_line("markers", "slow: mark test as slow to run")

def pytest_collection_modifyitems(config, items):
    if config.getoption("--runslow"):
        # --runslow given in cli: do not skip slow tests
        return
    skip_slow = pytest.mark.skip(reason="need --runslow option to run")
    for item in items:
        if "slow" in item.keywords:
            item.add_marker(skip_slow)
```

We can now write a test module like this:

```
# content of test_module.py
import pytest

def test_func_fast():
    pass

@pytest.mark.slow
def test_func_slow():
    pass
```

and when running it will see a skipped “slow” test:

```
$ pytest -rs # "-rs" means report details on the little 's'
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items

test_module.py .s [100%]

===== short test summary info =====
SKIPPED [1] test_module.py:8: need --runslow option to run
===== 1 passed, 1 skipped in 0.12s =====
```

Or run it including the slow marked test:

```
$ pytest --runslow
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items

test_module.py .. [100%]

===== 2 passed in 0.12s =====
```

Writing well integrated assertion helpers

If you have a test helper function called from a test you can use the `pytest.fail` marker to fail a test with a certain message. The test support function will not show up in the traceback if you set the `__tracebackhide__` option somewhere in the helper function. Example:

```
# content of test_checkconfig.py
import pytest

def checkconfig(x):
    __tracebackhide__ = True
    if not hasattr(x, "config"):
        pytest.fail("not configured: {}".format(x))

def test_something():
    checkconfig(42)
```

The `__tracebackhide__` setting influences pytest showing of tracebacks: the `checkconfig` function will not be shown unless the `--full-trace` command line option is specified. Let's run our little function:

```
$ pytest -q test_checkconfig.py
F [100%]
===== FAILURES =====
_____ test_something _____

    def test_something():
>     checkconfig(42)
E     Failed: not configured: 42

test_checkconfig.py:11: Failed
===== short test summary info =====
FAILED test_checkconfig.py::test_something - Failed: not configured: 42
1 failed in 0.12s
```

If you only want to hide certain exceptions, you can set `__tracebackhide__` to a callable which gets the `ExceptionInfo` object. You can for example use this to make sure unexpected exception types aren't hidden:

```
import operator
import pytest
```

(continues on next page)

(continued from previous page)

```
class ConfigException(Exception):
    pass

def checkconfig(x):
    __tracebackhide__ = operator.methodcaller("errisinstance", ConfigException)
    if not hasattr(x, "config"):
        raise ConfigException("not configured: {}".format(x))

def test_something():
    checkconfig(42)
```

This will avoid hiding the exception traceback on unrelated exceptions (i.e. bugs in assertion helpers).

Detect if running from within a pytest run

Usually it is a bad idea to make application code behave differently if called from a test. But if you absolutely must find out if your application code is running from a test you can do something like this:

```
# content of your_module.py

_called_from_test = False
```

```
# content of conftest.py

def pytest_configure(config):
    your_module._called_from_test = True
```

and then check for the `your_module._called_from_test` flag:

```
if your_module._called_from_test:
    # called from within a test run
    ...
else:
    # called "normally"
    ...
```

accordingly in your application.

Adding info to test report header

It's easy to present extra information in a pytest run:

```
# content of conftest.py

def pytest_report_header(config):
    return "project deps: mylib-1.1"
```

which will add the string to the test header accordingly:

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
project deps: mylib-1.1
rootdir: /home/sweet/project
collected 0 items

===== no tests ran in 0.12s =====
```

It is also possible to return a list of strings which will be considered as several lines of information. You may consider `config.getoption('verbose')` in order to display more information if applicable:

```
# content of conftest.py

def pytest_report_header(config):
    if config.getoption("verbose") > 0:
        return ["info1: did you know that ...", "did you?"]
```

which will add info only when run with “-v”:

```
$ pytest -v
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
info1: did you know that ...
did you?
rootdir: /home/sweet/project
collecting ... collected 0 items

===== no tests ran in 0.12s =====
```

and nothing when run plainly:

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 0 items

===== no tests ran in 0.12s =====
```

Profiling test duration

If you have a slow running large test suite you might want to find out which tests are the slowest. Let’s make an artificial test suite:

```
# content of test_some_are_slow.py
import time

def test_funcfast():
```

(continues on next page)

(continued from previous page)

```

time.sleep(0.1)

def test_funcslow1():
    time.sleep(0.2)

def test_funcslow2():
    time.sleep(0.3)

```

Now we can profile which test functions execute the slowest:

```

$ pytest --durations=3
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 3 items

test_some_are_slow.py ... [100%]

===== slowest 3 durations =====
0.30s call     test_some_are_slow.py::test_funcslow2
0.20s call     test_some_are_slow.py::test_funcslow1
0.10s call     test_some_are_slow.py::test_funcfast
===== 3 passed in 0.12s =====

```

Incremental testing - test steps

Sometimes you may have a testing situation which consists of a series of test steps. If one step fails it makes no sense to execute further steps as they are all expected to fail anyway and their tracebacks add no insight. Here is a simple `conftest.py` file which introduces an incremental marker which is to be used on classes:

```

# content of conftest.py

from typing import Dict, Tuple
import pytest

# store history of failures per test class name and per index in parametrize (if_
↳parametrize used)
_test_failed_incremental: Dict[str, Dict[Tuple[int, ...], str]] = {}

def pytest_runtest_makereport(item, call):
    if "incremental" in item.keywords:
        # incremental marker is used
        if call.excinfo is not None:
            # the test has failed
            # retrieve the class name of the test
            cls_name = str(item.cls)
            # retrieve the index of the test (if parametrize is used in combination_
↳with incremental)
            parametrize_index = (
                tuple(item.callspec.indices.values())
                if hasattr(item, "callspec")
            )

```

(continues on next page)

(continued from previous page)

```

        else ()
    )
    # retrieve the name of the test function
    test_name = item.originalname or item.name
    # store in _test_failed_incremental the original name of the failed test
    _test_failed_incremental.setdefault(cls_name, {}).setdefault(
        parametrize_index, test_name
    )

def pytest_runtest_setup(item):
    if "incremental" in item.keywords:
        # retrieve the class name of the test
        cls_name = str(item.cls)
        # check if a previous test has failed for this class
        if cls_name in _test_failed_incremental:
            # retrieve the index of the test (if parametrize is used in combination_
↳with incremental)
            parametrize_index = (
                tuple(item.callspec.indices.values())
                if hasattr(item, "callspec")
                else ()
            )
            # retrieve the name of the first test function to fail for this class_
↳name and index
            test_name = _test_failed_incremental[cls_name].get(parametrize_index, _
↳None)
            # if name found, test has failed for the combination of class name & test_
↳name

            if test_name is not None:
                pytest.xfail("previous test failed ({})" .format(test_name))

```

These two hook implementations work together to abort incremental-marked tests in a class. Here is a test module example:

```

# content of test_step.py

import pytest

@pytest.mark.incremental
class TestUserHandling:
    def test_login(self):
        pass

    def test_modification(self):
        assert 0

    def test_deletion(self):
        pass

def test_normal():
    pass

```

If we run this:

```

$ pytest -rx
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 4 items

test_step.py .Fx. [100%]

===== FAILURES =====
_____ TestUserHandling.test_modification _____

self = <test_step.TestUserHandling object at 0xdeadbeef0001>

    def test_modification(self):
>     assert 0
E     assert 0

test_step.py:11: AssertionError
===== short test summary info =====
XFAIL test_step.py::TestUserHandling::test_deletion
  reason: previous test failed (test_modification)
===== 1 failed, 2 passed, 1 xfailed in 0.12s =====

```

We'll see that `test_deletion` was not executed because `test_modification` failed. It is reported as an “expected failure”.

Package/Directory-level fixtures (setups)

If you have nested test directories, you can have per-directory fixture scopes by placing fixture functions in a `conftest.py` file in that directory. You can use all types of fixtures including *autouse fixtures* which are the equivalent of xUnit's setup/teardown concept. It's however recommended to have explicit fixture references in your tests or test classes rather than relying on implicitly executing setup/teardown functions, especially if they are far away from the actual tests.

Here is an example for making a `db` fixture available in a directory:

```

# content of a/conftest.py
import pytest

class DB:
    pass

@pytest.fixture(scope="session")
def db():
    return DB()

```

and then a test module in that directory:

```

# content of a/test_db.py
def test_a1(db):
    assert 0, db # to show value

```

another test module:


```
# content of a/test_db2.py
def test_a2(db):
    assert 0, db # to show value
```

and then a module in a sister directory which will not see the db fixture:

```
# content of b/test_error.py
def test_root(db): # no db here, will error out
    pass
```

We can run this:

```
$ pytest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 7 items

test_step.py .Fx. [ 57%]
a/test_db.py F [ 71%]
a/test_db2.py F [ 85%]
b/test_error.py E [100%]

===== ERRORS =====
_____ ERROR at setup of test_root _____
file /home/sweet/project/b/test_error.py, line 1
  def test_root(db): # no db here, will error out
E     fixture 'db' not found
>     available fixtures: cache, capfd, capfdbinary, caplog, capsys, capsysbinary,
↳ doctest_namespace, monkeypatch, pytestconfig, record_property, record_testsuite_
↳ property, record_xml_attribute, recwarn, tmp_path, tmp_path_factory, tmpdir, tmpdir_
↳ factory
>     use 'pytest --fixtures [testpath]' for help on them.

/home/sweet/project/b/test_error.py:1
===== FAILURES =====
_____ TestUserHandling.test_modification _____

self = <test_step.TestUserHandling object at 0xdeadbeef0002>

    def test_modification(self):
>     assert 0
E     assert 0

test_step.py:11: AssertionError
_____ test_a1 _____

db = <confstest.DB object at 0xdeadbeef0003>

    def test_a1(db):
>     assert 0, db # to show value
E     AssertionError: <confstest.DB object at 0xdeadbeef0003>
E     assert 0

a/test_db.py:2: AssertionError
_____ test_a2 _____
```

(continues on next page)

(continued from previous page)

```

db = <confptest.DB object at 0xdeadbeef0003>

    def test_a2(db):
>     assert 0, db # to show value
E     AssertionError: <confptest.DB object at 0xdeadbeef0003>
E     assert 0

a/test_db2.py:2: AssertionError
===== short test summary info =====
FAILED test_step.py::TestUserHandling::test_modification - assert 0
FAILED a/test_db.py::test_a1 - AssertionError: <confptest.DB object at 0x7...
FAILED a/test_db2.py::test_a2 - AssertionError: <confptest.DB object at 0x...
ERROR b/test_error.py::test_root
===== 3 failed, 2 passed, 1 xfailed, 1 error in 0.12s =====

```

The two test modules in the `a` directory see the same `db` fixture instance while the one test in the sister-directory `b` doesn't see it. We could of course also define a `db` fixture in that sister directory's `confptest.py` file. Note that each fixture is only instantiated if there is a test actually needing it (unless you use "autouse" fixture which are always executed ahead of the first test executing).

Post-process test reports / failures

If you want to postprocess test reports and need access to the executing environment you can implement a hook that gets called when the test "report" object is about to be created. Here we write out all failing test calls and also access a fixture (if it was used by the test) in case you want to query/look at it during your post processing. In our case we just write some information out to a `failures` file:

```

# content of confptest.py

import pytest
import os.path

@pytest.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    # execute all other hooks to obtain the report object
    outcome = yield
    rep = outcome.get_result()

    # we only look at actual failing test calls, not setup/teardown
    if rep.when == "call" and rep.failed:
        mode = "a" if os.path.exists("failures") else "w"
        with open("failures", mode) as f:
            # let's also access a fixture for the fun of it
            if "tmp_path" in item.fixturenames:
                extra = " ({}).format(item.funcargs["tmp_path"])
            else:
                extra = ""

            f.write(rep.nodeid + extra + "\n")

```

if you then have failing tests:

```
# content of test_module.py
def test_fail1(tmp_path):
    assert 0

def test_fail2():
    assert 0
```

and run them:

```
$ pytest test_module.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items

test_module.py FF [100%]

===== FAILURES =====
_____ test_fail1 _____

tmp_path = PosixPath('PYTEST_TMPDIR/test_fail10')

    def test_fail1(tmp_path):
>     assert 0
E     assert 0

test_module.py:2: AssertionError
_____ test_fail2 _____

    def test_fail2():
>     assert 0
E     assert 0

test_module.py:6: AssertionError
===== short test summary info =====
FAILED test_module.py::test_fail1 - assert 0
FAILED test_module.py::test_fail2 - assert 0
===== 2 failed in 0.12s =====
```

you will have a “failures” file which contains the failing test ids:

```
$ cat failures
test_module.py::test_fail1 (PYTEST_TMPDIR/test_fail10)
test_module.py::test_fail2
```

Making test result information available in fixtures

If you want to make test result reports available in fixture finalizers here is a little example implemented via a local plugin:

```
# content of conftest.py

import pytest

@pytest.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    # execute all other hooks to obtain the report object
    outcome = yield
    rep = outcome.get_result()

    # set a report attribute for each phase of a call, which can
    # be "setup", "call", "teardown"

    setattr(item, "rep_" + rep.when, rep)

@pytest.fixture
def something(request):
    yield
    # request.node is an "item" because we use the default
    # "function" scope
    if request.node.rep_setup.failed:
        print("setting up a test failed!", request.node.nodeid)
    elif request.node.rep_setup.passed:
        if request.node.rep_call.failed:
            print("executing test failed", request.node.nodeid)
```

if you then have failing tests:

```
# content of test_module.py

import pytest

@pytest.fixture
def other():
    assert 0

def test_setup_fails(something, other):
    pass

def test_call_fails(something):
    assert 0

def test_fail2():
    assert 0
```

and run it:

```

$ pytest -s test_module.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 3 items

test_module.py Esetting up a test failed! test_module.py::test_setup_fails
Executing test failed test_module.py::test_call_fails
F

===== ERRORS =====
_____ ERROR at setup of test_setup_fails _____

    @pytest.fixture
    def other():
>         assert 0
E         assert 0

test_module.py:7: AssertionError
===== FAILURES =====
_____ test_call_fails _____

something = None

    def test_call_fails(something):
>         assert 0
E         assert 0

test_module.py:15: AssertionError
_____ test_fail2 _____

    def test_fail2():
>         assert 0
E         assert 0

test_module.py:19: AssertionError
===== short test summary info =====
FAILED test_module.py::test_call_fails - assert 0
FAILED test_module.py::test_fail2 - assert 0
ERROR test_module.py::test_setup_fails - assert 0
===== 2 failed, 1 error in 0.12s =====

```

You'll see that the fixture finalizers could use the precise reporting information.

PYTEST_CURRENT_TEST environment variable

Sometimes a test session might get stuck and there might be no easy way to figure out which test got stuck, for example if pytest was run in quiet mode (`-q`) or you don't have access to the console output. This is particularly a problem if the problem happens only sporadically, the famous "flaky" kind of tests.

pytest sets the `PYTEST_CURRENT_TEST` environment variable when running tests, which can be inspected by process monitoring utilities or libraries like `psutil` to discover which test got stuck if necessary:

```
import psutil
```

(continues on next page)

(continued from previous page)

```

for pid in psutil.pids():
    environ = psutil.Process(pid).environ()
    if "PYTEST_CURRENT_TEST" in environ:
        print(f'pytest process {pid} running: {environ["PYTEST_CURRENT_TEST"]}')

```

During the test session pytest will set `PYTEST_CURRENT_TEST` to the current test *nodeid* and the current stage, which can be `setup`, `call`, or `teardown`.

For example, when running a single test function named `test_foo` from `foo_module.py`, `PYTEST_CURRENT_TEST` will be set to:

1. `foo_module.py::test_foo (setup)`
2. `foo_module.py::test_foo (call)`
3. `foo_module.py::test_foo (teardown)`

In that order.

Note: The contents of `PYTEST_CURRENT_TEST` is meant to be human readable and the actual format can be changed between releases (even bug fixes) so it shouldn't be relied on for scripting or automation.

Freezing pytest

If you freeze your application using a tool like [PyInstaller](#) in order to distribute it to your end-users, it is a good idea to also package your test runner and run your tests using the frozen application. This way packaging errors such as dependencies not being included into the executable can be detected early while also allowing you to send test files to users so they can run them in their machines, which can be useful to obtain more information about a hard to reproduce bug.

Fortunately recent `PyInstaller` releases already have a custom hook for `pytest`, but if you are using another tool to freeze executables such as `cx_freeze` or `py2exe`, you can use `pytest.freeze_includes()` to obtain the full list of internal `pytest` modules. How to configure the tools to find the internal modules varies from tool to tool, however.

Instead of freezing the `pytest` runner as a separate executable, you can make your frozen program work as the `pytest` runner by some clever argument handling during program startup. This allows you to have a single executable, which is usually more convenient. Please note that the mechanism for plugin discovery used by `pytest` (`setuptools` entry points) doesn't work with frozen executables so `pytest` can't find any third party plugins automatically. To include third party plugins like `pytest-timeout` they must be imported explicitly and passed on to `pytest.main`.

```

# contents of app_main.py
import sys
import pytest_timeout # Third party plugin

if len(sys.argv) > 1 and sys.argv[1] == "--pytest":
    import pytest

    sys.exit(pytest.main(sys.argv[2:], plugins=[pytest_timeout]))
else:
    # normal application execution: at this point argv can be parsed
    # by your argument-parsing library of choice as usual
    ...

```

This allows you to execute tests using the frozen application with standard `pytest` command-line options:

```
./app_main --pytest --verbose --tb=long --junitxml=results.xml test-suite/
```

5.1.3 Parametrizing tests

pytest allows to easily parametrize test functions. For basic docs, see *How to parametrize fixtures and test functions*.

In the following we provide some examples using the builtin mechanisms.

Generating parameters combinations, depending on command line

Let's say we want to execute a test with different computation parameters and the parameter range shall be determined by a command line argument. Let's first write a simple (do-nothing) computation test:

```
# content of test_compute.py

def test_compute(param1):
    assert param1 < 4
```

Now we add a test configuration like this:

```
# content of conftest.py

def pytest_addoption(parser):
    parser.addoption("--all", action="store_true", help="run all combinations")

def pytest_generate_tests(metafunc):
    if "param1" in metafunc.fixturenames:
        if metafunc.config.getoption("all"):
            end = 5
        else:
            end = 2
        metafunc.parametrize("param1", range(end))
```

This means that we only run 2 tests if we do not pass `--all`:

```
$ pytest -q test_compute.py
.. [100%]
2 passed in 0.12s
```

We run only two computations, so we see two dots. let's run the full monty:

```
$ pytest -q --all
....F [100%]
===== FAILURES =====
_____ test_compute[4] _____

param1 = 4

    def test_compute(param1):
>     assert param1 < 4
E     assert 4 < 4

test_compute.py:4: AssertionError
===== short test summary info =====
FAILED test_compute.py::test_compute[4] - assert 4 < 4
1 failed, 4 passed in 0.12s
```

As expected when running the full range of `param1` values we'll get an error on the last one.

Different options for test IDs

pytest will build a string that is the test ID for each set of values in a parametrized test. These IDs can be used with `-k` to select specific cases to run, and they will also identify the specific case when one is failing. Running pytest with `--collect-only` will show the generated IDs.

Numbers, strings, booleans and `None` will have their usual string representation used in the test ID. For other objects, pytest will make a string based on the argument name:

```
# content of test_time.py

from datetime import datetime, timedelta

import pytest

testdata = [
    (datetime(2001, 12, 12), datetime(2001, 12, 11), timedelta(1)),
    (datetime(2001, 12, 11), datetime(2001, 12, 12), timedelta(-1)),
]

@pytest.mark.parametrize("a,b,expected", testdata)
def test_timedistance_v0(a, b, expected):
    diff = a - b
    assert diff == expected

@pytest.mark.parametrize("a,b,expected", testdata, ids=["forward", "backward"])
def test_timedistance_v1(a, b, expected):
    diff = a - b
    assert diff == expected

def idfn(val):
    if isinstance(val, (datetime,)):
        # note this wouldn't show any hours/minutes/seconds
        return val.strftime("%Y%m%d")

@pytest.mark.parametrize("a,b,expected", testdata, ids=idfn)
def test_timedistance_v2(a, b, expected):
    diff = a - b
    assert diff == expected

@pytest.mark.parametrize(
    "a,b,expected",
    [
        pytest.param(
            datetime(2001, 12, 12), datetime(2001, 12, 11), timedelta(1), id="forward"
        ),
        pytest.param(
            datetime(2001, 12, 11), datetime(2001, 12, 12), timedelta(-1), id=
↪ "backward"
        ),
    ],
)
```

(continues on next page)

(continued from previous page)

```

    ],
)
def test_timedistance_v3(a, b, expected):
    diff = a - b
    assert diff == expected

```

In `test_timedistance_v0`, we let pytest generate the test IDs.

In `test_timedistance_v1`, we specified `ids` as a list of strings which were used as the test IDs. These are succinct, but can be a pain to maintain.

In `test_timedistance_v2`, we specified `ids` as a function that can generate a string representation to make part of the test ID. So our `datetime` values use the label generated by `idfn`, but because we didn't generate a label for `timedelta` objects, they are still using the default pytest representation:

```

$ pytest test_time.py --collect-only
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 8 items

<Module test_time.py>
  <Function test_timedistance_v0[a0-b0-expected0]>
  <Function test_timedistance_v0[a1-b1-expected1]>
  <Function test_timedistance_v1[forward]>
  <Function test_timedistance_v1[backward]>
  <Function test_timedistance_v2[20011212-20011211-expected0]>
  <Function test_timedistance_v2[20011211-20011212-expected1]>
  <Function test_timedistance_v3[forward]>
  <Function test_timedistance_v3[backward]>

===== 8 tests collected in 0.12s =====

```

In `test_timedistance_v3`, we used `pytest.param` to specify the test IDs together with the actual data, instead of listing them separately.

A quick port of “testscenarios”

Here is a quick port to run tests configured with `testscenarios`, an add-on from Robert Collins for the standard `unittest` framework. We only have to work a bit to construct the correct arguments for `pytest's` `Metafunc.parametrize()`:

```

# content of test_scenarios.py

def pytest_generate_tests(metafunc):
    idlist = []
    argvalues = []
    for scenario in metafunc.cls.scenarios:
        idlist.append(scenario[0])
        items = scenario[1].items()
        argnames = [x[0] for x in items]
        argvalues.append([x[1] for x in items])
    metafunc.parametrize(argnames, argvalues, ids=idlist, scope="class")

```

(continues on next page)

(continued from previous page)

```

scenario1 = ("basic", {"attribute": "value"})
scenario2 = ("advanced", {"attribute": "value2"})

class TestSampleWithScenarios:
    scenarios = [scenario1, scenario2]

    def test_demo1(self, attribute):
        assert isinstance(attribute, str)

    def test_demo2(self, attribute):
        assert isinstance(attribute, str)

```

this is a fully self-contained example which you can run with:

```

$ pytest test_scenarios.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 4 items

test_scenarios.py .... [100%]

===== 4 passed in 0.12s =====

```

If you just collect tests you'll also nicely see 'advanced' and 'basic' as variants for the test function:

```

$ pytest --collect-only test_scenarios.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 4 items

<Module test_scenarios.py>
  <Class TestSampleWithScenarios>
    <Function test_demo1[basic]>
    <Function test_demo2[basic]>
    <Function test_demo1[advanced]>
    <Function test_demo2[advanced]>

===== 4 tests collected in 0.12s =====

```

Note that we told `metafunc.parametrize()` that your scenario values should be considered class-scoped. With `pytest-2.3` this leads to a resource-based ordering.

Deferring the setup of parametrized resources

The parametrization of test functions happens at collection time. It is a good idea to setup expensive resources like DB connections or subprocess only when the actual test is run. Here is a simple example how you can achieve that. This test requires a db object fixture:

```
# content of test_backends.py

import pytest

def test_db_initialized(db):
    # a dummy test
    if db.__class__.__name__ == "DB2":
        pytest.fail("deliberately failing for demo purposes")
```

We can now add a test configuration that generates two invocations of the `test_db_initialized` function and also implements a factory that creates a database object for the actual test invocations:

```
# content of conftest.py
import pytest

def pytest_generate_tests(metafunc):
    if "db" in metafunc.fixturenames:
        metafunc.parametrize("db", ["d1", "d2"], indirect=True)

class DB1:
    "one database object"

class DB2:
    "alternative database object"

@pytest.fixture
def db(request):
    if request.param == "d1":
        return DB1()
    elif request.param == "d2":
        return DB2()
    else:
        raise ValueError("invalid internal test config")
```

Let's first see how it looks like at collection time:

```
$ pytest test_backends.py --collect-only
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items

<Module test_backends.py>
  <Function test_db_initialized[d1]>
  <Function test_db_initialized[d2]>
```

(continues on next page)

(continued from previous page)

```
===== 2 tests collected in 0.12s =====
```

And then when we run the test:

```
$ pytest -q test_backends.py
.F [100%]
===== FAILURES =====
_____ test_db_initialized[d2] _____

db = <conftest.DB2 object at 0xdeadbeef0001>

    def test_db_initialized(db):
        # a dummy test
        if db.__class__.__name__ == "DB2":
>         pytest.fail("deliberately failing for demo purposes")
E         Failed: deliberately failing for demo purposes

test_backends.py:8: Failed
===== short test summary info =====
FAILED test_backends.py::test_db_initialized[d2] - Failed: deliberately f...
1 failed, 1 passed in 0.12s
```

The first invocation with `db == "DB1"` passed while the second with `db == "DB2"` failed. Our `db` fixture function has instantiated each of the DB values during the setup phase while the `pytest_generate_tests` generated two according calls to the `test_db_initialized` during the collection phase.

Indirect parametrization

Using the `indirect=True` parameter when parametrizing a test allows to parametrize a test with a fixture receiving the values before passing them to a test:

```
import pytest

@pytest.fixture
def fixt(request):
    return request.param * 3

@pytest.mark.parametrize("fixt", ["a", "b"], indirect=True)
def test_indirect(fixt):
    assert len(fixt) == 3
```

This can be used, for example, to do more expensive setup at test run time in the fixture, rather than having to run those setup steps at collection time.

Apply indirect on particular arguments

Very often parametrization uses more than one argument name. There is opportunity to apply `indirect` parameter on particular arguments. It can be done by passing list or tuple of arguments' names to `indirect`. In the example below there is a function `test_indirect` which uses two fixtures: `x` and `y`. Here we give to `indirect` the list, which contains the name of the fixture `x`. The `indirect` parameter will be applied to this argument only, and the value `a` will be passed to respective fixture function:

```
# content of test_indirect_list.py

import pytest

@pytest.fixture(scope="function")
def x(request):
    return request.param * 3

@pytest.fixture(scope="function")
def y(request):
    return request.param * 2

@pytest.mark.parametrize("x, y", [("a", "b")], indirect=["x"])
def test_indirect(x, y):
    assert x == "aaa"
    assert y == "b"
```

The result of this test will be successful:

```
$ pytest -v test_indirect_list.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 1 item

test_indirect_list.py::test_indirect[a-b] PASSED [100%]

===== 1 passed in 0.12s =====
```

Parametrizing test methods through per-class configuration

Here is an example `pytest_generate_tests` function implementing a parametrization scheme similar to Michael Foord's `unittest parametrizer` but in a lot less code:

```
# content of ./test_parametrize.py
import pytest

def pytest_generate_tests(metafunc):
    # called once per each test function
    funcarglist = metafunc.cls.params[metafunc.function.__name__]
    argnames = sorted(funcarglist[0])
    metafunc.parametrize(
```

(continues on next page)

(continued from previous page)

```

    argnames, [[funcargs[name] for name in argnames] for funcargs in funcarglist]
)

class TestClass:
    # a map specifying multiple argument sets for a test method
    params = {
        "test_equals": [dict(a=1, b=2), dict(a=3, b=3)],
        "test_zerodivision": [dict(a=1, b=0)],
    }

    def test_equals(self, a, b):
        assert a == b

    def test_zerodivision(self, a, b):
        with pytest.raises(ZeroDivisionError):
            a / b

```

Our test generator looks up a class-level definition which specifies which argument sets to use for each test function. Let's run it:

```

$ pytest -q
F.. [100%]
===== FAILURES =====
_____ TestClass.test_equals[1-2] _____

self = <test_parametrize.TestClass object at 0xdeadbeef0002>, a = 1, b = 2

    def test_equals(self, a, b):
>     assert a == b
E     assert 1 == 2

test_parametrize.py:21: AssertionError
===== short test summary info =====
FAILED test_parametrize.py::TestClass::test_equals[1-2] - assert 1 == 2
1 failed, 2 passed in 0.12s

```

Indirect parametrization with multiple fixtures

Here is a stripped down real-life example of using parametrized testing for testing serialization of objects between different python interpreters. We define a `test_basic_objects` function which is to be run with different sets of arguments for its three arguments:

- `python1`: first python interpreter, run to pickle-dump an object to a file
- `python2`: second interpreter, run to pickle-load an object from a file
- `obj`: object to be dumped/loaded

```

"""
module containing a parametrized tests testing cross-python
serialization via the pickle module.
"""
import shutil
import subprocess
import textwrap

```

(continues on next page)

(continued from previous page)

```

import pytest

pythonlist = ["python3.5", "python3.6", "python3.7"]

@pytest.fixture(params=pythonlist)
def python1(request, tmp_path):
    picklefile = tmp_path / "data.pickle"
    return Python(request.param, picklefile)

@pytest.fixture(params=pythonlist)
def python2(request, python1):
    return Python(request.param, python1.picklefile)

class Python:
    def __init__(self, version, picklefile):
        self.pythonpath = shutil.which(version)
        if not self.pythonpath:
            pytest.skip(f"{version!r} not found")
        self.picklefile = picklefile

    def dumps(self, obj):
        dumpfile = self.picklefile.with_name("dump.py")
        dumpfile.write_text(
            textwrap.dedent(
                r"""
import pickle
f = open({!r}, 'wb')
s = pickle.dump({!r}, f, protocol=2)
f.close()
""".format(
                    str(self.picklefile), obj
                )
            )
        )
        subprocess.check_call((self.pythonpath, str(dumpfile)))

    def load_and_is_true(self, expression):
        loadfile = self.picklefile.with_name("load.py")
        loadfile.write_text(
            textwrap.dedent(
                r"""
import pickle
f = open({!r}, 'rb')
obj = pickle.load(f)
f.close()
res = eval({!r})
if not res:
    raise SystemExit(1)
""".format(
                    str(self.picklefile), expression
                )
            )
        )

```

(continues on next page)

(continued from previous page)

```

        print(loadfile)
        subprocess.check_call((self.pythonpath, str(loadfile)))

@pytest.mark.parametrize("obj", [42, {}, {1: 3}])
def test_basic_objects(python1, python2, obj):
    python1.dumps(obj)
    python2.load_and_is_true(f"obj == {obj}")

```

Running it results in some skips if we don't have all the python interpreters installed and otherwise runs all combinations (3 interpreters times 3 interpreters times 3 objects to serialize/deserialize):

```

. $ pytest -rs -q multipython.py
..... [100%]
27 passed in 0.12s

```

Indirect parametrization of optional implementations/imports

If you want to compare the outcomes of several implementations of a given API, you can write test functions that receive the already imported implementations and get skipped in case the implementation is not importable/available. Let's say we have a "base" implementation and the other (possibly optimized ones) need to provide similar results:

```

# content of conftest.py

import pytest

@pytest.fixture(scope="session")
def basemod(request):
    return pytest.importorskip("base")

@pytest.fixture(scope="session", params=["opt1", "opt2"])
def optmod(request):
    return pytest.importorskip(request.param)

```

And then a base implementation of a simple function:

```

# content of base.py
def func1():
    return 1

```

And an optimized version:

```

# content of opt1.py
def func1():
    return 1.0001

```

And finally a little test module:

```

# content of test_module.py

def test_func1(basemod, optmod):
    assert round(basemod.func1(), 3) == round(optmod.func1(), 3)

```


If you run this with reporting for skips enabled:

```
$ pytest -rs test_module.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 2 items

test_module.py .s [100%]

===== short test summary info =====
SKIPPED [1] conftest.py:12: could not import 'opt2': No module named 'opt2'
===== 1 passed, 1 skipped in 0.12s =====
```

You'll see that we don't have an `opt2` module and thus the second test run of our `test_func1` was skipped. A few notes:

- the fixture functions in the `conftest.py` file are “session-scoped” because we don't need to import more than once
- if you have multiple test functions and a skipped import, you will see the `[1]` count increasing in the report
- you can put `@pytest.mark.parametrize` style parametrization on the test functions to parametrize input/output values as well.

Set marks or test ID for individual parametrized test

Use `pytest.param` to apply marks or set test ID to individual parametrized test. For example:

```
# content of test_pytest_param_example.py
import pytest

@pytest.mark.parametrize(
    "test_input, expected",
    [
        ("3+5", 8),
        pytest.param("1+7", 8, marks=pytest.mark.basic),
        pytest.param("2+4", 6, marks=pytest.mark.basic, id="basic_2+4"),
        pytest.param(
            "6*9", 42, marks=[pytest.mark.basic, pytest.mark.xfail], id="basic_6*9"
        ),
    ],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

In this example, we have 4 parametrized tests. Except for the first test, we mark the rest three parametrized tests with the custom marker `basic`, and for the fourth test we also use the built-in mark `xfail` to indicate this test is expected to fail. For explicitness, we set test ids for some tests.

Then run `pytest` with verbose mode and with only the `basic` marker:

```
$ pytest -v -m basic
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
```

(continues on next page)

(continued from previous page)

```
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 24 items / 21 deselected / 3 selected

test_pytest_param_example.py::test_eval[1+7-8] PASSED           [ 33%]
test_pytest_param_example.py::test_eval[basic_2+4] PASSED      [ 66%]
test_pytest_param_example.py::test_eval[basic_6*9] XFAIL       [100%]

===== 2 passed, 21 deselected, 1 xfailed in 0.12s =====
```

As the result:

- Four tests were collected
- One test was deselected because it doesn't have the `basic` mark.
- Three tests with the `basic` mark was selected.
- The test `test_eval[1+7-8]` passed, but the name is autogenerated and confusing.
- The test `test_eval[basic_2+4]` passed.
- The test `test_eval[basic_6*9]` was expected to fail and did fail.

Parametrizing conditional raising

Use `pytest.raises()` with the `pytest.mark.parametrize` decorator to write parametrized tests in which some tests raise exceptions and others do not.

It is helpful to define a no-op context manager `does_not_raise` to serve as a complement to `raises`. For example:

```
from contextlib import contextmanager
import pytest

@contextmanager
def does_not_raise():
    yield

@pytest.mark.parametrize(
    "example_input, expectation",
    [
        (3, does_not_raise()),
        (2, does_not_raise()),
        (1, does_not_raise()),
        (0, pytest.raises(ZeroDivisionError)),
    ],
)
def test_division(example_input, expectation):
    """Test how much I know division."""
    with expectation:
        assert (6 / example_input) is not None
```

In the example above, the first three test cases should run unexceptionally, while the fourth should raise `ZeroDivisionError`.

If you're only supporting Python 3.7+, you can simply use `nullcontext` to define `does_not_raise`:

```
from contextlib import nullcontext as does_not_raise
```

Or, if you're supporting Python 3.3+ you can use:

```
from contextlib import ExitStack as does_not_raise
```

Or, if desired, you can `pip install contextlib2` and use:

```
from contextlib2 import nullcontext as does_not_raise
```

5.1.4 Working with custom markers

Here are some examples using the *How to mark test functions with attributes* mechanism.

Marking test functions and selecting them for a run

You can “mark” a test function with custom metadata like this:

```
# content of test_server.py

import pytest

@pytest.mark.webtest
def test_send_http():
    pass # perform some webtest test for your app

def test_something_quick():
    pass

def test_another():
    pass

class TestClass:
    def test_method(self):
        pass
```

You can then restrict a test run to only run tests marked with `webtest`:

```
$ pytest -v -m webtest
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 4 items / 3 deselected / 1 selected

test_server.py::test_send_http PASSED [100%]

===== 1 passed, 3 deselected in 0.12s =====
```

Or the inverse, running all tests except the `webtest` ones:

```

$ pytest -v -m "not webtest"
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 4 items / 1 deselected / 3 selected

test_server.py::test_something_quick PASSED [ 33%]
test_server.py::test_another PASSED [ 66%]
test_server.py::TestClass::test_method PASSED [100%]

===== 3 passed, 1 deselected in 0.12s =====

```

Selecting tests based on their node ID

You can provide one or more *node IDs* as positional arguments to select only specified tests. This makes it easy to select tests based on their module, class, method, or function name:

```

$ pytest -v test_server.py::TestClass::test_method
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED [100%]

===== 1 passed in 0.12s =====

```

You can also select on the class:

```

$ pytest -v test_server.py::TestClass
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED [100%]

===== 1 passed in 0.12s =====

```

Or select multiple nodes:

```

$ pytest -v test_server.py::TestClass test_server.py::test_send_http
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 2 items

test_server.py::TestClass::test_method PASSED [ 50%]

```

(continues on next page)

(continued from previous page)

```
test_server.py::test_send_http PASSED [100%]
===== 2 passed in 0.12s =====
```

Note: Node IDs are of the form `module.py::class::method` or `module.py::function`. Node IDs control which tests are collected, so `module.py::class` will select all test methods on the class. Nodes are also created for each parameter of a parametrized fixture or test, so selecting a parametrized test must include the parameter value, e.g. `module.py::function[param]`.

Node IDs for failing tests are displayed in the test summary info when running `pytest` with the `-rf` option. You can also construct Node IDs from the output of `pytest --collectonly`.

Using `-k expr` to select tests based on their name

New in version 2.0/2.3.4.

You can use the `-k` command line option to specify an expression which implements a substring match on the test names instead of the exact match on markers that `-m` provides. This makes it easy to select tests based on their names:

Changed in version 5.4.

The expression matching is now case-insensitive.

```
$ pytest -v -k http # running with the above defined example module
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 4 items / 3 deselected / 1 selected

test_server.py::test_send_http PASSED [100%]
===== 1 passed, 3 deselected in 0.12s =====
```

And you can also run all tests except the ones that match the keyword:

```
$ pytest -k "not send_http" -v
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 4 items / 1 deselected / 3 selected

test_server.py::test_something_quick PASSED [ 33%]
test_server.py::test_another PASSED [ 66%]
test_server.py::TestClass::test_method PASSED [100%]
===== 3 passed, 1 deselected in 0.12s =====
```

Or to select “http” and “quick” tests:

```
$ pytest -k "http or quick" -v
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 4 items / 2 deselected / 2 selected

test_server.py::test_send_http PASSED [ 50%]
test_server.py::test_something_quick PASSED [100%]

===== 2 passed, 2 deselected in 0.12s =====
```

You can use `and`, `or`, `not` and parentheses.

In addition to the test's name, `-k` also matches the names of the test's parents (usually, the name of the file and class it's in), attributes set on the test function, markers applied to it or its parents and any *extra keywords* explicitly added to it or its parents.

Registering markers

Registering markers for your test suite is simple:

```
# content of pytest.ini
[pytest]
markers =
    webtest: mark a test as a webtest.
    slow: mark test as slow.
```

Multiple custom markers can be registered, by defining each one in its own line, as shown in above example.

You can ask which markers exist for your test suite - the list includes our just defined `webtest` and `slow` markers:

```
$ pytest --markers
@pytest.mark.webtest: mark a test as a webtest.

@pytest.mark.slow: mark test as slow.

@pytest.mark.filterwarnings(warning): add a warning filter to the given test. see_
↳https://docs.pytest.org/en/stable/how-to/capture-warnings.html#pytest-mark-
↳filterwarnings

@pytest.mark.skip(reason=None): skip the given test function with an optional reason._
↳Example: skip(reason="no way of currently testing this") skips the test.

@pytest.mark.skipif(condition, ..., *, reason=...): skip the given test function if_
↳any of the conditions evaluate to True. Example: skipif(sys.platform == 'win32')_
↳skips the test if we are on the win32 platform. See https://docs.pytest.org/en/
↳stable/reference/reference.html#pytest-mark-skipif

@pytest.mark.xfail(condition, ..., *, reason=..., run=True, raises=None, strict=xfail_
↳strict): mark the test function as an expected failure if any of the conditions_
↳evaluate to True. Optionally specify a reason for better reporting and run=False if_
↳you don't even want to execute the test function. If only specific exception(s) are_
↳expected, you can list them in raises, and if the test fails in other ways, it will_
↳be reported as a true failure. See https://docs.pytest.org/en/stable/reference/
↳reference.html#pytest-mark-xfail
```

(continues on next page)

(continued from previous page)

```
@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times
↳ passing in different arguments in turn. argvalues generally needs to be a list of
↳ values if argnames specifies only one name or a list of tuples of values if
↳ argnames specifies multiple names. Example: @parametrize('arg1', [1,2]) would lead
↳ to two calls of the decorated test function, one with arg1=1 and another with
↳ arg1=2. see https://docs.pytest.org/en/stable/how-to/parametrize.html for more info
↳ and examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all
↳ of the specified fixtures. see https://docs.pytest.org/en/stable/explanation/
↳ fixtures.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin
↳ machinery will try to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin
↳ machinery will try to call it last/as late as possible.
```

For an example on how to add and work with markers from a plugin, see *Custom marker and command line option to control test runs*.

Note: It is recommended to explicitly register markers so that:

- There is one place in your test suite defining your markers
- Asking for existing markers via `pytest --markers` gives good output
- Typos in function markers are treated as an error if you use the `--strict-markers` option.

Marking whole classes or modules

You may use `pytest.mark` decorators with classes to apply markers to all of its test methods:

```
# content of test_mark_classlevel.py
import pytest

@pytest.mark.webtest
class TestClass:
    def test_startup(self):
        pass

    def test_startup_and_more(self):
        pass
```

This is equivalent to directly applying the decorator to the two test functions.

To apply marks at the module level, use the `pytestmark` global variable:

```
import pytest
pytestmark = pytest.mark.webtest
```

or multiple markers:

```
pytestmark = [pytest.mark.webtest, pytest.mark.slowtest]
```

Due to legacy reasons, before class decorators were introduced, it is possible to set the `pytestmark` attribute on a test class like this:

```
import pytest

class TestClass:
    pytestmark = pytest.mark.webtest
```

Marking individual tests when using parametrize

When using `parametrize`, applying a mark will make it apply to each individual test. However it is also possible to apply a marker to an individual test instance:

```
import pytest

@pytest.mark.foo
@pytest.mark.parametrize(
    ("n", "expected"), [(1, 2), pytest.param(1, 3, marks=pytest.mark.bar), (2, 3)]
)
def test_increment(n, expected):
    assert n + 1 == expected
```

In this example the mark “foo” will apply to each of the three tests, whereas the “bar” mark is only applied to the second test. Skip and xfail marks can also be applied in this way, see [Skip/xfail with parametrize](#).

Custom marker and command line option to control test runs

Plugins can provide custom markers and implement specific behaviour based on it. This is a self-contained example which adds a command line option and a parametrized test function marker to run tests specifies via named environments:

```
# content of conftest.py

import pytest

def pytest_addoption(parser):
    parser.addoption(
        "-E",
        action="store",
        metavar="NAME",
        help="only run tests matching the environment NAME.",
    )

def pytest_configure(config):
    # register an additional marker
    config.addinvalue_line(
        "markers", "env(name): mark test to run only on named environment"
    )
```

(continues on next page)

(continued from previous page)

```
def pytest_runtest_setup(item):
    envnames = [mark.args[0] for mark in item.iter_markers(name="env")]
    if envnames:
        if item.config.getoption("-E") not in envnames:
            pytest.skip("test requires env in {!r}".format(envnames))
```

A test file using this local plugin:

```
# content of test_someenv.py

import pytest

@pytest.mark.env("stage1")
def test_basic_db_operation():
    pass
```

and an example invocations specifying a different environment than what the test needs:

```
$ pytest -E stage2
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_someenv.py s [100%]

===== 1 skipped in 0.12s =====
```

and here is one that specifies exactly the environment needed:

```
$ pytest -E stage1
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 1 item

test_someenv.py . [100%]

===== 1 passed in 0.12s =====
```

The `--markers` option always gives you a list of available markers:

```
$ pytest --markers
@pytest.mark.env(name): mark test to run only on named environment

@pytest.mark.filterwarnings(warning): add a warning filter to the given test. see_
↳ https://docs.pytest.org/en/stable/how-to/capture-warnings.html#pytest-mark-
↳ filterwarnings

@pytest.mark.skip(reason=None): skip the given test function with an optional reason._
↳ Example: skip(reason="no way of currently testing this") skips the test.
```

(continues on next page)

(continued from previous page)

```

@pytest.mark.skipif(condition, ..., *, reason=...): skip the given test function if
↳ any of the conditions evaluate to True. Example: skipif(sys.platform == 'win32')
↳ skips the test if we are on the win32 platform. See https://docs.pytest.org/en/
↳ stable/reference/reference.html#pytest-mark-skipif

@pytest.mark.xfail(condition, ..., *, reason=..., run=True, raises=None, strict=xfail_
↳ strict): mark the test function as an expected failure if any of the conditions
↳ evaluate to True. Optionally specify a reason for better reporting and run=False if
↳ you don't even want to execute the test function. If only specific exception(s) are
↳ expected, you can list them in raises, and if the test fails in other ways, it will
↳ be reported as a true failure. See https://docs.pytest.org/en/stable/reference/
↳ reference.html#pytest-mark-xfail

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times
↳ passing in different arguments in turn. argvalues generally needs to be a list of
↳ values if argnames specifies only one name or a list of tuples of values if
↳ argnames specifies multiple names. Example: @parametrize('arg1', [1,2]) would lead
↳ to two calls of the decorated test function, one with arg1=1 and another with
↳ arg1=2. see https://docs.pytest.org/en/stable/how-to/parametrize.html for more info
↳ and examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all
↳ of the specified fixtures. see https://docs.pytest.org/en/stable/explanation/
↳ fixtures.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin
↳ machinery will try to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin
↳ machinery will try to call it last/as late as possible.

```

Passing a callable to custom markers

Below is the config file that will be used in the next examples:

```

# content of conftest.py
import sys

def pytest_runtest_setup(item):
    for marker in item.iter_markers(name="my_marker"):
        print(marker)
        sys.stdout.flush()

```

A custom marker can have its argument set, i.e. `args` and `kwargs` properties, defined by either invoking it as a callable or using `pytest.mark.MARKER_NAME.with_args`. These two methods achieve the same effect most of the time.

However, if there is a callable as the single positional argument with no keyword arguments, using the `pytest.mark.MARKER_NAME(c)` will not pass `c` as a positional argument but decorate `c` with the custom marker (see [MarkDecorator](#)). Fortunately, `pytest.mark.MARKER_NAME.with_args` comes to the rescue:

```

# content of test_custom_marker.py
import pytest

```

(continues on next page)

(continued from previous page)

```
def hello_world(*args, **kwargs):
    return "Hello World"

@pytest.mark.my_marker.with_args(hello_world)
def test_with_args():
    pass
```

The output is as follows:

```
$ pytest -q -s
Mark(name='my_marker', args=(<function hello_world at 0xdeadbeef0001>,), kwargs={})
.
1 passed in 0.12s
```

We can see that the custom marker has its argument set extended with the function `hello_world`. This is the key difference between creating a custom marker as a callable, which invokes `__call__` behind the scenes, and using `with_args`.

Reading markers which were set from multiple places

If you are heavily using markers in your test suite you may encounter the case where a marker is applied several times to a test function. From plugin code you can read over all such settings. Example:

```
# content of test_mark_three_times.py
import pytest

pytestmark = pytest.mark.glob("module", x=1)

@pytest.mark.glob("class", x=2)
class TestClass:
    @pytest.mark.glob("function", x=3)
    def test_something(self):
        pass
```

Here we have the marker “glob” applied three times to the same test function. From a conftest file we can read it like this:

```
# content of conftest.py
import sys

def pytest_runtest_setup(item):
    for mark in item.iter_markers(name="glob"):
        print("glob args={} kwargs={}".format(mark.args, mark.kwargs))
        sys.stdout.flush()
```

Let’s run this without capturing output and see what we get:

```
$ pytest -q -s
glob args=('function',) kwargs={'x': 3}
glob args=('class',) kwargs={'x': 2}
glob args=('module',) kwargs={'x': 1}
.
1 passed in 0.12s
```

Marking platform specific tests with pytest

Consider you have a test suite which marks tests for particular platforms, namely `pytest.mark.darwin`, `pytest.mark.win32` etc. and you also have tests that run on all platforms and have no specific marker. If you now want to have a way to only run the tests for your particular platform, you could use the following plugin:

```
# content of conftest.py
#
import sys
import pytest

ALL = set("darwin linux win32".split())

def pytest_runtest_setup(item):
    supported_platforms = ALL.intersection(mark.name for mark in item.iter_markers())
    plat = sys.platform
    if supported_platforms and plat not in supported_platforms:
        pytest.skip("cannot run on platform {}".format(plat))
```

then tests will be skipped if they were specified for a different platform. Let's do a little test file to show how this looks like:

```
# content of test_plat.py

import pytest

@pytest.mark.darwin
def test_if_apple_is_evil():
    pass

@pytest.mark.linux
def test_if_linux_works():
    pass

@pytest.mark.win32
def test_if_win32_crashes():
    pass

def test_runs_everywhere():
    pass
```

then you will see two tests skipped and two executed tests as expected:

```
$ pytest -rs # this option reports skip reasons
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 4 items

test_plat.py s.s. [100%]
```

(continues on next page)

(continued from previous page)

```
===== short test summary info =====
SKIPPED [2] conftest.py:12: cannot run on platform linux
===== 2 passed, 2 skipped in 0.12s =====
```

Note that if you specify a platform via the marker-command line option like this:

```
$ pytest -m linux
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 4 items / 3 deselected / 1 selected

test_plat.py . [100%]

===== 1 passed, 3 deselected in 0.12s =====
```

then the unmarked-tests will not be run. It is thus a way to restrict the run to the specific tests.

Automatically adding markers based on test names

If you have a test suite where test function names indicate a certain type of test, you can implement a hook that automatically defines markers so that you can use the `-m` option with it. Let's look at this test module:

```
# content of test_module.py

def test_interface_simple():
    assert 0

def test_interface_complex():
    assert 0

def test_event_simple():
    assert 0

def test_something_else():
    assert 0
```

We want to dynamically define two markers and can do it in a `conftest.py` plugin:

```
# content of conftest.py

import pytest

def pytest_collection_modifyitems(items):
    for item in items:
        if "interface" in item.nodeid:
            item.add_marker(pytest.mark.interface)
        elif "event" in item.nodeid:
            item.add_marker(pytest.mark.event)
```

We can now use the `-m` option to select one set:

```
$ pytest -m interface --tb=short
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 4 items / 2 deselected / 2 selected

test_module.py FF [100%]

===== FAILURES =====
_____ test_interface_simple _____
test_module.py:4: in test_interface_simple
    assert 0
E   assert 0
_____ test_interface_complex _____
test_module.py:8: in test_interface_complex
    assert 0
E   assert 0
===== short test summary info =====
FAILED test_module.py::test_interface_simple - assert 0
FAILED test_module.py::test_interface_complex - assert 0
===== 2 failed, 2 deselected in 0.12s =====
```

or to select both “event” and “interface” tests:

```
$ pytest -m "interface or event" --tb=short
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project
collected 4 items / 1 deselected / 3 selected

test_module.py FFF [100%]

===== FAILURES =====
_____ test_interface_simple _____
test_module.py:4: in test_interface_simple
    assert 0
E   assert 0
_____ test_interface_complex _____
test_module.py:8: in test_interface_complex
    assert 0
E   assert 0
_____ test_event_simple _____
test_module.py:12: in test_event_simple
    assert 0
E   assert 0
===== short test summary info =====
FAILED test_module.py::test_interface_simple - assert 0
FAILED test_module.py::test_interface_complex - assert 0
FAILED test_module.py::test_event_simple - assert 0
===== 3 failed, 1 deselected in 0.12s =====
```

5.1.5 A session-fixture which can look at all collected tests

A session-scoped fixture effectively has access to all collected test items. Here is an example of a fixture function which walks all collected tests and looks if their test class defines a `callme` method and calls it:

```
# content of conftest.py

import pytest

@pytest.fixture(scope="session", autouse=True)
def callattr_ahead_of_alltests(request):
    print("callattr_ahead_of_alltests called")
    seen = {None}
    session = request.node
    for item in session.items:
        cls = item.getparent(pytest.Class)
        if cls not in seen:
            if hasattr(cls.obj, "callme"):
                cls.obj.callme()
            seen.add(cls)
```

test classes may now define a `callme` method which will be called ahead of running any tests:

```
# content of test_module.py

class TestHello:
    @classmethod
    def callme(cls):
        print("callme called!")

    def test_method1(self):
        print("test_method1 called")

    def test_method2(self):
        print("test_method1 called")

class TestOther:
    @classmethod
    def callme(cls):
        print("callme other called")

    def test_other(self):
        print("test other")

# works with unittest as well ...
import unittest

class SomeTest(unittest.TestCase):
    @classmethod
    def callme(self):
        print("SomeTest callme called")

    def test_unit1(self):
```

(continues on next page)

(continued from previous page)

```
print("test_unit1 method called")
```

If you run this without output capturing:

```
$ pytest -q -s test_module.py
callattr_ahead_of_alltests called
callme called!
callme other called
SomeTest callme called
test_method1 called
.test_method1 called
.test other
.test_unit1 method called
.
4 passed in 0.12s
```

5.1.6 Changing standard (Python) test discovery

Ignore paths during test collection

You can easily ignore certain test directories and modules during collection by passing the `--ignore=path` option on the cli. `pytest` allows multiple `--ignore` options. Example:

```
tests/
|-- example
|   |-- test_example_01.py
|   |-- test_example_02.py
|   '-- test_example_03.py
|-- foobar
|   |-- test_foobar_01.py
|   |-- test_foobar_02.py
|   '-- test_foobar_03.py
'-- hello
    '-- world
        |-- test_world_01.py
        |-- test_world_02.py
        '-- test_world_03.py
```

Now if you invoke `pytest` with `--ignore=tests/foobar/test_foobar_03.py --ignore=tests/hello/`, you will see that `pytest` only collects test-modules, which do not match the patterns specified:

```
===== test session starts =====
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 5 items

tests/example/test_example_01.py .           [ 20%]
tests/example/test_example_02.py .           [ 40%]
tests/example/test_example_03.py .           [ 60%]
tests/foobar/test_foobar_01.py .             [ 80%]
tests/foobar/test_foobar_02.py .            [100%]

===== 5 passed in 0.02 seconds =====
```


The `--ignore-glob` option allows to ignore test file paths based on Unix shell-style wildcards. If you want to exclude test-modules that end with `_01.py`, execute `pytest` with `--ignore-glob='*_01.py'`.

Deselect tests during test collection

Tests can individually be deselected during collection by passing the `--deselect=item` option. For example, say `tests/foobar/test_foobar_01.py` contains `test_a` and `test_b`. You can run all of the tests within `tests/` *except* for `tests/foobar/test_foobar_01.py::test_a` by invoking `pytest` with `--deselect tests/foobar/test_foobar_01.py::test_a`. `pytest` allows multiple `--deselect` options.

Keeping duplicate paths specified from command line

Default behavior of `pytest` is to ignore duplicate paths specified from the command line. Example:

```
pytest path_a path_a
...
collected 1 item
...
```

Just collect tests once.

To collect duplicate tests, use the `--keep-duplicates` option on the cli. Example:

```
pytest --keep-duplicates path_a path_a
...
collected 2 items
...
```

As the collector just works on directories, if you specify twice a single test file, `pytest` will still collect it twice, no matter if the `--keep-duplicates` is not specified. Example:

```
pytest test_a.py test_a.py
...
collected 2 items
...
```

Changing directory recursion

You can set the `norecursedirs` option in an ini-file, for example your `pytest.ini` in the project root directory:

```
# content of pytest.ini
[pytest]
norecursedirs = .svn _build tmp*
```

This would tell `pytest` to not recurse into typical subversion or sphinx-build directories or into any `tmp` prefixed directory.

Changing naming conventions

You can configure different naming conventions by setting the `python_files`, `python_classes` and `python_functions` in your *configuration file*. Here is an example:

```
# content of pytest.ini
# Example 1: have pytest look for "check" instead of "test"
[pytest]
python_files = check_*.py
python_classes = Check
python_functions = *_check
```

This would make `pytest` look for tests in files that match the `check_*` .py glob-pattern, `Check` prefixes in classes, and functions and methods that match `*_check`. For example, if we have:

```
# content of check_myapp.py
class CheckMyApp:
    def simple_check(self):
        pass

    def complex_check(self):
        pass
```

The test collection would look like this:

```
$ pytest --collect-only
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project, configfile: pytest.ini
collected 2 items

<Module check_myapp.py>
  <Class CheckMyApp>
    <Function simple_check>
    <Function complex_check>

===== 2 tests collected in 0.12s =====
```

You can check for multiple glob patterns by adding a space between the patterns:

```
# Example 2: have pytest look for files with "test" and "example"
# content of pytest.ini
[pytest]
python_files = test_*.py example_*.py
```

Note: the `python_functions` and `python_classes` options has no effect for `unittest.TestCase` test discovery because `pytest` delegates discovery of test case methods to `unittest` code.

Interpreting cmdline arguments as Python packages

You can use the `--pyargs` option to make `pytest` try interpreting arguments as python package names, deriving their file system path and then running the test. For example if you have `unittest2` installed you can type:

```
pytest --pyargs unittest2.test.test_skipping -q
```

which would run the respective test module. Like with other options, through an ini-file and the `addopts` option you can make this change more permanently:

```
# content of pytest.ini
[pytest]
addopts = --pyargs
```

Now a simple invocation of `pytest NAME` will check if `NAME` exists as an importable package/module and otherwise treat it as a filesystem path.

Finding out what is collected

You can always peek at the collection tree without running tests like this:

```
. $ pytest --collect-only pythoncollection.py
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project, configfile: pytest.ini
collected 3 items

<Module CWD/pythoncollection.py>
  <Function test_function>
  <Class TestClass>
    <Function test_method>
    <Function test_anothermethod>

===== 3 tests collected in 0.12s =====
```

Customizing test collection

You can easily instruct `pytest` to discover tests from every Python file:

```
# content of pytest.ini
[pytest]
python_files = *.py
```

However, many projects will have a `setup.py` which they don't want to be imported. Moreover, there may files only importable by a specific python version. For such cases you can dynamically define files to be ignored by listing them in a `conftest.py` file:

```
# content of conftest.py
import sys

collect_ignore = ["setup.py"]
if sys.version_info[0] > 2:
    collect_ignore.append("pkg/module_py2.py")
```

and then if you have a module file like this:

```
# content of pkg/module_py2.py
def test_only_on_python2():
    try:
        assert 0
    except Exception, e:
        pass
```

and a setup.py dummy file like this:

```
# content of setup.py
0 / 0 # will raise exception if imported
```

If you run with a Python 2 interpreter then you will find the one test and will leave out the setup.py file:

```
#$ pytest --collect-only
===== test session starts =====
platform linux2 -- Python 2.7.10, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 1 items
<Module 'pkg/module_py2.py'>
  <Function 'test_only_on_python2'>

===== 1 tests found in 0.04 seconds =====
```

If you run with a Python 3 interpreter both the one test and the setup.py file will be left out:

```
$ pytest --collect-only
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project, configfile: pytest.ini
collected 0 items

===== no tests collected in 0.12s =====
```

It's also possible to ignore files based on Unix shell-style wildcards by adding patterns to `collect_ignore_glob`.

The following example `confptest.py` ignores the file `setup.py` and in addition all files that end with `*_py2.py` when executed with a Python 3 interpreter:

```
# content of confptest.py
import sys

collect_ignore = ["setup.py"]
if sys.version_info[0] > 2:
    collect_ignore_glob = ["*_py2.py"]
```

Since Pytest 2.6, users can prevent pytest from discovering classes that start with `Test` by setting a boolean `__test__` attribute to `False`.

```
# Will not be discovered as a test
class TestClass:
    __test__ = False
```

5.1.7 Working with non-python tests

A basic example for specifying tests in Yaml files

Here is an example `confptest.py` (extracted from Ali Afshar's special purpose `pytest-yamlwsgi` plugin). This `confptest.py` will collect `test*.yaml` files and will execute the yaml-formatted content as custom tests:

```
# content of confptest.py
import pytest

def pytest_collect_file(parent, fspath):
    if fspath.suffix == ".yaml" and fspath.name.startswith("test"):
        return YamlFile.from_parent(parent, path=fspath)

class YamlFile(pytest.File):
    def collect(self):
        # We need a yaml parser, e.g. PyYAML.
        import yaml

        raw = yaml.safe_load(self.path.open())
        for name, spec in sorted(raw.items()):
            yield YamlItem.from_parent(self, name=name, spec=spec)

class YamlItem(pytest.Item):
    def __init__(self, name, parent, spec):
        super().__init__(name, parent)
        self.spec = spec

    def runtest(self):
        for name, value in sorted(self.spec.items()):
            # Some custom test execution (dumb example follows).
            if name != value:
                raise YamlException(self, name, value)

    def repr_failure(self, excinfo):
        """Called when self.runtest() raises an exception."""
        if isinstance(excinfo.value, YamlException):
            return "\n".join(
                [
                    "usecase execution failed",
                    "  spec failed: {1!r}: {2!r}".format(*excinfo.value.args),
                    "  no further details known at this point.",
                ]
            )

    def reportinfo(self):
        return self.path, 0, f"usecase: {self.name}"

class YamlException(Exception):
    """Custom exception for error reporting."""
```

You can create a simple example file:

```
# test_simple.yaml
ok:
  sub1: sub1

hello:
  world: world
  some: other
```

and if you installed [PyYAML](#) or a compatible YAML-parser you can now execute the test specification:

```
nonpython $ pytest test_simple.yaml
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project/nonpython
collected 2 items

test_simple.yaml F. [100%]

===== FAILURES =====
_____ usecase: hello _____
usecase execution failed
  spec failed: 'some': 'other'
  no further details known at this point.
===== short test summary info =====
FAILED test_simple.yaml::hello
===== 1 failed, 1 passed in 0.12s =====
```

You get one dot for the passing `sub1: sub1` check and one failure. Obviously in the above `conftest.py` you'll want to implement a more interesting interpretation of the `yaml-values`. You can easily write your own domain specific testing language this way.

Note: `repr_failure(excinfo)` is called for representing test failures. If you create custom collection nodes you can return an error representation string of your choice. It will be reported as a (red) string.

`reportinfo()` is used for representing the test location and is also consulted when reporting in verbose mode:

```
nonpython $ pytest -v
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y -- $PYTHON_
↳PREFIX/bin/python
cachedir: .pytest_cache
rootdir: /home/sweet/project/nonpython
collecting ... collected 2 items

test_simple.yaml::hello FAILED [ 50%]
test_simple.yaml::ok PASSED [100%]

===== FAILURES =====
_____ usecase: hello _____
usecase execution failed
  spec failed: 'some': 'other'
  no further details known at this point.
===== short test summary info =====
FAILED test_simple.yaml::hello
===== 1 failed, 1 passed in 0.12s =====
```

While developing your custom test collection and execution it's also interesting to just look at the collection tree:

```

nonpython $ pytest --collect-only
===== test session starts =====
platform linux -- Python 3.x.y, pytest-6.x.y, py-1.x.y, pluggy-1.x.y
cachedir: .pytest_cache
rootdir: /home/sweet/project/nonpython
collected 2 items

<Package nonpython>
  <YamlFile test_simple.yaml>
    <YamlItem hello>
    <YamlItem ok>

===== 2 tests collected in 0.12s =====

```

5.2 Backwards Compatibility Policy

pytest is actively evolving and is a project that has been decades in the making, we keep learning about new and better structures to express different details about testing.

While we implement those modifications we try to ensure an easy transition and don't want to impose unnecessary churn on our users and community/plugin authors.

As of now, pytest considers multiple types of backward compatibility transitions:

- a) trivial: APIs which trivially translate to the new mechanism, and do not cause problematic changes.

We try to support those indefinitely while encouraging users to switch to newer/better mechanisms through documentation.

- b) transitional: the old and new API don't conflict and we can help users transition by using warnings, while supporting both for a prolonged time.

We will only start the removal of deprecated functionality in major releases (e.g. if we deprecate something in 3.0 we will start to remove it in 4.0), and keep it around for at least two minor releases (e.g. if we deprecate something in 3.9 and 4.0 is the next release, we start to remove it in 5.0, not in 4.0).

When the deprecation expires (e.g. 4.0 is released), we won't remove the deprecated functionality immediately, but will use the standard warning filters to turn them into **errors** by default. This approach makes it explicit that removal is imminent, and still gives you time to turn the deprecated feature into a warning instead of an error so it can be dealt with in your own time. In the next minor release (e.g. 4.1), the feature will be effectively removed.

- c) true breakage: should only be considered when normal transition is unreasonably unsustainable and would offset important development/features by years. In addition, they should be limited to APIs where the number of actual users is very small (for example only impacting some plugins), and can be coordinated with the community in advance.

Examples for such upcoming changes:

- removal of `pytest_runttest_protocol/nextitem` - #895
- rearranging of the node tree to include `FunctionDefinition`
- rearranging of `SetupState` #895

True breakages must be announced first in an issue containing:

- Detailed description of the change

- Rationale
- Expected impact on users and plugin authors (example in #895)

After there's no hard *-I* on the issue it should be followed up by an initial proof-of-concept Pull Request.

This POC serves as both a coordination point to assess impact and potential inspiration to come up with a transitional solution after all.

After a reasonable amount of time the PR can be merged to base a new major release.

For the PR to mature from POC to acceptance, it must contain: * Setup of deprecation errors/warnings that help users fix and port their code. If it is possible to introduce a deprecation period under the current series, before the true breakage, it should be introduced in a separate PR and be part of the current release stream. * Detailed description of the rationale and examples on how to port code in `doc/en/deprecations.rst`.

5.3 History

5.3.1 Focus primary on smooth transition - stance (pre 6.0)

Keeping backwards compatibility has a very high priority in the pytest project. Although we have deprecated functionality over the years, most of it is still supported. All deprecations in pytest were done because simpler or more efficient ways of accomplishing the same tasks have emerged, making the old way of doing things unnecessary.

With the pytest 3.0 release we introduced a clear communication scheme for when we will actually remove the old busted joint and politely ask you to use the new hotness instead, while giving you enough time to adjust your tests or raise concerns if there are valid reasons to keep deprecated functionality around.

To communicate changes we issue deprecation warnings using a custom warning hierarchy (see *Internal pytest warnings*). These warnings may be suppressed using the standard means: `-W` command-line flag or `filterwarnings` ini options (see *How to capture warnings*), but we suggest to use these sparingly and temporarily, and heed the warnings when possible.

We will only start the removal of deprecated functionality in major releases (e.g. if we deprecate something in 3.0 we will start to remove it in 4.0), and keep it around for at least two minor releases (e.g. if we deprecate something in 3.9 and 4.0 is the next release, we start to remove it in 5.0, not in 4.0).

When the deprecation expires (e.g. 4.0 is released), we won't remove the deprecated functionality immediately, but will use the standard warning filters to turn them into **errors** by default. This approach makes it explicit that removal is imminent, and still gives you time to turn the deprecated feature into a warning instead of an error so it can be dealt with in your own time. In the next minor release (e.g. 4.1), the feature will be effectively removed.

Deprecation Roadmap

Features currently deprecated and removed in previous releases can be found in *Deprecations and Removals*.

We track future deprecation and removal of features using milestones and the `deprecation` and `removal` labels on GitHub.

5.4 Deprecations and Removals

This page lists all pytest features that are currently deprecated or have been removed in past major releases. The objective is to give users a clear rationale why a certain feature has been removed, and what alternatives should be used instead.

- *Deprecated Features*
 - *py.path.local arguments for hooks replaced with pathlib.Path*
 - *Implementing the pytest_cmdline_preparse hook*
 - *Diamond inheritance between pytest.File and pytest.Item*
 - *Backward compatibilities in Parser.addoption*
 - *Raising unittest.SkipTest during collection*
 - *The --strict command-line option*
 - *The yield_fixture function/decorator*
 - *The pytest_warning_captured hook*
 - *The pytest.collect module*
 - *The pytest._fillfuncargs function*
- *Removed Features*
 - *--no-print-logs command-line option*
 - *Result log (--result-log)*
 - *pytest_collect_directory hook*
 - *TerminalReporter.writer*
 - *junit_family default value change to "xunit2"*
 - *Node Construction changed to Node.from_parent*
 - *pytest.fixture arguments are keyword only*
 - *funcargnames alias for fixturenames*
 - *pytest.config.global*
 - *"message" parameter of pytest.raises*
 - *raises / warns with a string as the second argument*
 - *Using Class in custom Collectors*
 - *marks in pytest.mark.parametrize*
 - *pytest_funcarg__ prefix*
 - *[pytest] section in setup.cfg files*
 - *Metafunc.addcall*
 - *cached_setup*
 - *pytest_plugins in non-top-level conftest files*
 - *Config.warn and Node.warn*

- *record_xml_property*
- *Passing command-line string to `pytest.main()`*
- *Calling fixtures directly*
- *yield tests*
- *Internal classes accessed through `Node`*
- *`Node.get_marker`*
- *`somefunction.markname`*
- *`pytest_namespace`*

5.4.1 Deprecated Features

Below is a complete list of all pytest features which are considered deprecated. Using those features will issue `PytestWarning` or subclasses, which can be filtered using *standard warning filters*.

`py.path.local` arguments for hooks replaced with `pathlib.Path`

In order to support the transition to `pathlib`, the following hooks now receive additional arguments:

- `pytest_ignore_collect(fspath: pathlib.Path)`
- `pytest_collect_file(fspath: pathlib.Path)`
- `pytest_pycollect_makemodule(fspath: pathlib.Path)`
- `pytest_report_header(startpath: pathlib.Path)`
- `pytest_report_collectionfinish(startpath: pathlib.Path)`

The accompanying `py.path.local` based paths have been deprecated: plugins which manually invoke those hooks should only pass the new `pathlib.Path` arguments, and users should change their hook implementations to use the new `pathlib.Path` arguments.

Implementing the `pytest_cmdline_preparse` hook

Deprecated since version 7.0.

Implementing the `pytest_cmdline_preparse` hook has been officially deprecated. Implement the `pytest_load_initial_conftests` hook instead.

```
def pytest_cmdline_preparse(config: Config, args: List[str]) -> None:
    ...

# becomes:

def pytest_load_initial_conftests(
    early_config: Config, parser: Parser, args: List[str]
) -> None:
    ...
```

Diamond inheritance between `pytest.File` and `pytest.Item`

Deprecated since version 6.3.

Inheriting from both `Item` and `file` at once has never been supported officially, however some plugins providing linting/code analysis have been using this as a hack.

This practice is now officially deprecated and a common way to fix this is [example pr fixing inheritance](#).

Backward compatibilities in `Parser.adoption`

Deprecated since version 2.4.

Several behaviors of `Parser.adoption` are now scheduled for removal in pytest 7 (deprecated since pytest 2.4.0):

- `parser.adoption(..., help=".. %default ..")` - use `%(default)s` instead.
- `parser.adoption(..., type="int/string/float/complex")` - use `type=int` etc. instead.

Raising `unittest.SkipTest` during collection

Deprecated since version 6.3.

Raising `unittest.SkipTest` to skip collection of tests during the pytest collection phase is deprecated. Use `pytest.skip()` instead.

Note: This deprecation only relates to using `unittest.SkipTest` during test collection. You are probably not doing that. Ordinary usage of `unittest.SkipTest` / `unittest.TestCase.skipTest()` / `unittest.skip()` in `unittest` test cases is fully supported.

The `--strict` command-line option

Deprecated since version 6.2.

The `--strict` command-line option has been deprecated in favor of `--strict-markers`, which better conveys what the option does.

We have plans to maybe in the future to reintroduce `--strict` and make it an encompassing flag for all strictness related options (`--strict-markers` and `--strict-config` at the moment, more might be introduced in the future).

The `yield_fixture` function/decorator

Deprecated since version 6.2.

`pytest.yield_fixture` is a deprecated alias for `pytest.fixture()`.

It has been so for a very long time, so can be search/replaced safely.

The `pytest_warning_captured` hook

Deprecated since version 6.0.

This hook has an `item` parameter which cannot be serialized by `pytest-xdist`.

Use the `pytest_warning_recored` hook instead, which replaces the `item` parameter by a `nodeid` parameter.

The `pytest.collect` module

Deprecated since version 6.0.

The `pytest.collect` module is no longer part of the public API, all its names should now be imported from `pytest` directly instead.

The `pytest._fillfuncargs` function

Deprecated since version 6.0.

This function was kept for backward compatibility with an older plugin.

It's functionality is not meant to be used directly, but if you must replace it, use `function._request._fillfixtures()` instead, though note this is not a public API and may break in the future.

5.4.2 Removed Features

As stated in our *Backwards Compatibility Policy* policy, deprecated features are removed only in major releases after an appropriate period of deprecation has passed.

`--no-print-logs` command-line option

Deprecated since version 5.4.

Removed in version 6.0.

The `--no-print-logs` option and `log_print` ini setting are removed. If you used them, please use `--show-capture` instead.

A `--show-capture` command-line option was added in `pytest 3.5.0` which allows to specify how to display captured output when tests fail: `no`, `stdout`, `stderr`, `log` or `all` (the default).

Result log (`--result-log`)

Deprecated since version 4.0.

Removed in version 6.0.

The `--result-log` option produces a stream of test reports which can be analysed at runtime, but it uses a custom format which requires users to implement their own parser.

The `pytest-reportlog` plugin provides a `--report-log` option, a more standard and extensible alternative, producing one JSON object per-line, and should cover the same use cases. Please try it out and provide feedback.

The `pytest-reportlog` plugin might even be merged into the core at some point, depending on the plans for the plugins and number of users using it.

pytest_collect_directory hook

Removed in version 6.0.

The `pytest_collect_directory` has not worked properly for years (it was called but the results were ignored). Users may consider using `pytest_collection_modifyitems` instead.

TerminalReporter.writer

Removed in version 6.0.

The `TerminalReporter.writer` attribute has been deprecated and should no longer be used. This was inadvertently exposed as part of the public API of that plugin and ties it too much with `py.io.TerminalWriter`.

Plugins that used `TerminalReporter.writer` directly should instead use `TerminalReporter` methods that provide the same functionality.

junit_family default value change to “xunit2”

Changed in version 6.0.

The default value of `junit_family` option will change to `xunit2` in pytest 6.0, which is an update of the old `xunit1` format and is supported by default in modern tools that manipulate this type of file (for example, Jenkins, Azure Pipelines, etc.).

Users are recommended to try the new `xunit2` format and see if their tooling that consumes the JUnit XML file supports it.

To use the new format, update your `pytest.ini`:

```
[pytest]
junit_family=xunit2
```

If you discover that your tooling does not support the new format, and want to keep using the legacy version, set the option to `legacy` instead:

```
[pytest]
junit_family=legacy
```

By using `legacy` you will keep using the `legacy/xunit1` format when upgrading to pytest 6.0, where the default format will be `xunit2`.

In order to let users know about the transition, pytest will issue a warning in case the `--junitxml` option is given in the command line but `junit_family` is not explicitly configured in `pytest.ini`.

Services known to support the `xunit2` format:

- Jenkins with the JUnit plugin.
- Azure Pipelines.

Node Construction changed to `Node.from_parent`

Changed in version 6.0.

The construction of nodes now should use the named constructor `from_parent`. This limitation in api surface intends to enable better/simpler refactoring of the collection tree.

This means that instead of `MyItem(name="foo", parent=collector, obj=42)` one now has to invoke `MyItem.from_parent(collector, name="foo")`.

Plugins that wish to support older versions of pytest and suppress the warning can use `hasattr` to check if `from_parent` exists in that version:

```
def pytest_pycollect_makeitem(collector, name, obj):
    if hasattr(MyItem, "from_parent"):
        item = MyItem.from_parent(collector, name="foo")
        item.obj = 42
        return item
    else:
        return MyItem(name="foo", parent=collector, obj=42)
```

Note that `from_parent` should only be called with keyword arguments for the parameters.

`pytest.fixture` arguments are keyword only

Removed in version 6.0.

Passing arguments to `pytest.fixture()` as positional arguments has been removed - pass them by keyword instead.

`funcargnames` alias for `fixturenames`

Removed in version 6.0.

The `FixtureRequest`, `Metafunc`, and `Function` classes track the names of their associated fixtures, with the aptly-named `fixturenames` attribute.

Prior to pytest 2.3, this attribute was named `funcargnames`, and we have kept that as an alias since. It is finally due for removal, as it is often confusing in places where we or plugin authors must distinguish between fixture names and names supplied by non-fixture things such as `pytest.mark.parametrize`.

`pytest.config.global`

Removed in version 5.0.

The `pytest.config.global` object is deprecated. Instead use `request.config` (via the `request` fixture) or if you are a plugin author use the `pytest_configure(config)` hook. Note that many hooks can also access the `config` object indirectly, through `session.config` or `item.config` for example.

"message" parameter of `pytest.raises`

Removed in version 5.0.

It is a common mistake to think this parameter will match the exception message, while in fact it only serves to provide a custom message in case the `pytest.raises` check fails. To prevent users from making this mistake, and because it is believed to be little used, `pytest` is deprecating it without providing an alternative for the moment.

If you have a valid use case for this parameter, consider that to obtain the same results you can just call `pytest.fail` manually at the end of the `with` statement.

For example:

```
with pytest.raises(TimeoutError, message="Client got unexpected message"):
    wait_for(websocket.recv(), 0.5)
```

Becomes:

```
with pytest.raises(TimeoutError):
    wait_for(websocket.recv(), 0.5)
    pytest.fail("Client got unexpected message")
```

If you still have concerns about this deprecation and future removal, please comment on [issue #3974](#).

`raises` / `warns` with a string as the second argument

Removed in version 5.0.

Use the context manager form of these instead. When necessary, invoke `exec` directly.

Example:

```
pytest.raises(ZeroDivisionError, "1 / 0")
pytest.raises(SyntaxError, "a $ b")

pytest.warns(DeprecationWarning, "my_function()")
pytest.warns(SyntaxWarning, "assert(1, 2)")
```

Becomes:

```
with pytest.raises(ZeroDivisionError):
    1 / 0
with pytest.raises(SyntaxError):
    exec("a $ b") # exec is required for invalid syntax

with pytest.warns(DeprecationWarning):
    my_function()
with pytest.warns(SyntaxWarning):
    exec("assert(1, 2)") # exec is used to avoid a top-level warning
```

Using Class in custom Collectors

Removed in version 4.0.

Using objects named "Class" as a way to customize the type of nodes that are collected in Collector subclasses has been deprecated. Users instead should use `pytest_pycollect_makeitem` to customize node types during collection.

This issue should affect only advanced plugins who create new collection types, so if you see this warning message please contact the authors so they can change the code.

marks in `pytest.mark.parametrize`

Removed in version 4.0.

Applying marks to values of a `pytest.mark.parametrize` call is now deprecated. For example:

```
@pytest.mark.parametrize(
    "a, b",
    [
        (3, 9),
        pytest.mark.xfail(reason="flaky")(6, 36),
        (10, 100),
        (20, 200),
        (40, 400),
        (50, 500),
    ],
)
def test_foo(a, b):
    ...
```

This code applies the `pytest.mark.xfail(reason="flaky")` mark to the (6, 36) value of the above parametrization call.

This was considered hard to read and understand, and also its implementation presented problems to the code preventing further internal improvements in the marks architecture.

To update the code, use `pytest.param`:

```
@pytest.mark.parametrize(
    "a, b",
    [
        (3, 9),
        pytest.param(6, 36, marks=pytest.mark.xfail(reason="flaky")),
        (10, 100),
        (20, 200),
        (40, 400),
        (50, 500),
    ],
)
def test_foo(a, b):
    ...
```


pytest_funcarg__ prefix

Removed in version 4.0.

In very early pytest versions fixtures could be defined using the `pytest_funcarg__` prefix:

```
def pytest_funcarg__data():
    return SomeData()
```

Switch over to the `@pytest.fixture` decorator:

```
@pytest.fixture
def data():
    return SomeData()
```

[pytest] section in setup.cfg files

Removed in version 4.0.

`[pytest]` sections in `setup.cfg` files should now be named `[tool:pytest]` to avoid conflicts with other distutils commands.

Metafunc.addcall

Removed in version 4.0.

`Metafunc.addcall` was a precursor to the current parametrized mechanism. Users should use `pytest.Metafunc.parametrize()` instead.

Example:

```
def pytest_generate_tests(metafunc):
    metafunc.addcall({"i": 1}, id="1")
    metafunc.addcall({"i": 2}, id="2")
```

Becomes:

```
def pytest_generate_tests(metafunc):
    metafunc.parametrize("i", [1, 2], ids=["1", "2"])
```

cached_setup

Removed in version 4.0.

`request.cached_setup` was the precursor of the `setup/teardown` mechanism available to fixtures.

Example:

```
@pytest.fixture
def db_session():
    return request.cached_setup(
        setup=Session.create, teardown=lambda session: session.close(), scope="module"
    )
```

This should be updated to make use of standard fixture mechanisms:

```
@pytest.fixture(scope="module")
def db_session():
    session = Session.create()
    yield session
    session.close()
```

You can consult funcarg comparison section in the docs for more information.

pytest_plugins in non-top-level conftest files

Removed in version 4.0.

Defining `pytest_plugins` is now deprecated in non-top-level `conftest.py` files because they will activate referenced plugins *globally*, which is surprising because for all other pytest features `conftest.py` files are only *active* for tests at or below it.

Config.warn and Node.warn

Removed in version 4.0.

Those methods were part of the internal pytest warnings system, but since 3.8 pytest is using the builtin warning system for its own warnings, so those two functions are now deprecated.

`Config.warn` should be replaced by calls to the standard `warnings.warn`, example:

```
config.warn("C1", "some warning")
```

Becomes:

```
warnings.warn(pytest.PytestWarning("some warning"))
```

`Node.warn` now supports two signatures:

- `node.warn(PytestWarning("some message"))`: is now the **recommended** way to call this function. The warning instance must be a `PytestWarning` or subclass.
- `node.warn("CI", "some message")`: this code/message form has been **removed** and should be converted to the warning instance form above.

record_xml_property

Removed in version 4.0.

The `record_xml_property` fixture is now deprecated in favor of the more generic `record_property`, which can be used by other consumers (for example `pytest-html`) to obtain custom information about the test run.

This is just a matter of renaming the fixture as the API is the same:

```
def test_foo(record_xml_property):
    ...
```

Change to:

```
def test_foo(record_property):
    ...
```

Passing command-line string to `pytest.main()`

Removed in version 4.0.

Passing a command-line string to `pytest.main()` is deprecated:

```
pytest.main("-v -s")
```

Pass a list instead:

```
pytest.main(["-v", "-s"])
```

By passing a string, users expect that `pytest` will interpret that command-line using the shell rules they are working on (for example `bash` or `Powershell`), but this is very hard/impossible to do in a portable way.

Calling fixtures directly

Removed in version 4.0.

Calling a fixture function directly, as opposed to request them in a test function, is deprecated.

For example:

```
@pytest.fixture
def cell():
    return ...

@pytest.fixture
def full_cell():
    cell = cell()
    cell.make_full()
    return cell
```

This is a great source of confusion to new users, which will often call the fixture functions and request them from test functions interchangeably, which breaks the fixture resolution model.

In those cases just request the function directly in the dependent fixture:

```
@pytest.fixture
def cell():
    return ...

@pytest.fixture
def full_cell(cell):
    cell.make_full()
    return cell
```

Alternatively if the fixture function is called multiple times inside a test (making it hard to apply the above pattern) or if you would like to make minimal changes to the code, you can create a fixture which calls the original function together with the name parameter:

```
def cell():
    return ...
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture(name="cell")
def cell_fixture():
    return cell()
```

yield tests

Removed in version 4.0.

pytest supported `yield`-style tests, where a test function actually `yield` functions and values that are then turned into proper test methods. Example:

```
def check(x, y):
    assert x ** x == y

def test_squared():
    yield check, 2, 4
    yield check, 3, 9
```

This would result into two actual test functions being generated.

This form of test function doesn't support fixtures properly, and users should switch to `pytest.mark.parametrize`:

```
@pytest.mark.parametrize("x, y", [(2, 4), (3, 9)])
def test_squared(x, y):
    assert x ** x == y
```

Internal classes accessed through Node

Removed in version 4.0.

Access of `Module`, `Function`, `Class`, `Instance`, `File` and `Item` through `Node` instances now issue this warning:

```
usage of Function.Module is deprecated, please use pytest.Module instead
```

Users should just import `pytest` and access those objects using the `pytest` module.

This has been documented as deprecated for years, but only now we are actually emitting deprecation warnings.

Node.get_marker

Removed in version 4.0.

As part of a large *Marker revamp and iteration*, `_pytest.nodes.Node.get_marker` is removed. See *the documentation* on tips on how to update your code.

`somefunction.markname`

Removed in version 4.0.

As part of a large *Marker revamp and iteration* we already deprecated using `MarkInfo` the only correct way to get markers of an element is via `node.iter_markers(name)`.

`pytest_namespace`

Removed in version 4.0.

This hook is deprecated because it greatly complicates the pytest internals regarding configuration and initialization, making some bug fixes and refactorings impossible.

Example of usage:

```
class MySymbol:
    ...

def pytest_namespace():
    return {"my_symbol": MySymbol() }
```

Plugin authors relying on this hook should instead require that users now import the plugin modules directly (with an appropriate public API).

As a stopgap measure, plugin authors may still inject their names into pytest's namespace, usually during `pytest_configure`:

```
import pytest

def pytest_configure():
    pytest.my_symbol = MySymbol()
```

5.5 Python 2.7 and 3.4 support

It is demanding on the maintainers of an open source project to support many Python versions, as there's extra cost of keeping code compatible between all versions, while holding back on features only made possible on newer Python versions.

In case of Python 2 and 3, the difference between the languages makes it even more prominent, because many new Python 3 features cannot be used in a Python 2/3 compatible code base.

Python 2.7 EOL has been reached **in 2020**, with the last release made in April, 2020.

Python 3.4 EOL has been reached **in 2019**, with the last release made in March, 2019.

For those reasons, in Jun 2019 it was decided that **pytest 4.6** series will be the last to support Python 2.7 and 3.4.

5.5.1 What this means for general users

Thanks to the `python_requires` `setuptools` option, Python 2.7 and Python 3.4 users using a modern pip version will install the last pytest 4.6.X version automatically even if 5.0 or later versions are available on PyPI.

Users should ensure they are using the latest pip and `setuptools` versions for this to work.

5.5.2 Maintenance of 4.6.X versions

Until January 2020, the pytest core team ported many bug-fixes from the main release into the `4.6.x` branch, with several 4.6.X releases being made along the year.

From now on, the core team will **no longer actively backport patches**, but the `4.6.x` branch will continue to exist so the community itself can contribute patches.

The core team will be happy to accept those patches, and make new 4.6.X releases **until mid-2020** (but consider that date as a ballpark, after that date the team might still decide to make new releases for critical bugs).

Technical aspects

(This section is a transcript from [#5275](#)).

In this section we describe the technical aspects of the Python 2.7 and 3.4 support plan.

What goes into 4.6.X releases

New 4.6.X releases will contain bug fixes only.

When will 4.6.X releases happen

New 4.6.X releases will happen after we have a few bugs in place to release, or if a few weeks have passed (say a single bug has been fixed a month after the latest 4.6.X release).

No hard rules here, just ballpark.

Who will handle applying bug fixes

We core maintainers expect that people still using Python 2.7/3.4 and being affected by bugs to step up and provide patches and/or port bug fixes from the active branches.

We will be happy to guide users interested in doing so, so please don't hesitate to ask.

Backporting changes into 4.6

Please follow these instructions:

1. `git fetch --all --prune`
2. `git checkout origin/4.6.x -b backport-XXXX # use the PR number here`
3. Locate the merge commit on the PR, in the *merged* message, for example:
`nicoddemus merged commit 0f8b462 into pytest-dev:features`
4. `git cherry-pick -m1 REVISION # use the revision you found above (0f8b462).`

5. Open a PR targeting 4.6.x:
 - Prefix the message with [4.6] so it is an obvious backport
 - Delete the PR body, it usually contains a duplicate commit message.

Providing new PRs to 4.6

Fresh pull requests to 4.6.x will be accepted provided that the equivalent code in the active branches does not contain that bug (for example, a bug is specific to Python 2 only).

Bug fixes that also happen in the mainstream version should be first fixed there, and then backported as per instructions above.

5.6 Contribution getting started

Contributions are highly welcomed and appreciated. Every little bit of help counts, so do not hesitate!

Contents

- *Contribution getting started*
 - *Feature requests and feedback*
 - *Report bugs*
 - *Fix bugs*
 - *Implement features*
 - *Write documentation*
 - *Submitting Plugins to pytest-dev*
 - *Preparing Pull Requests*
 - *Joining the Development Team*
 - *Backporting bug fixes for the next patch release*
 - *Handling stale issues/PRs*
 - *Closing Issues*

5.6.1 Feature requests and feedback

Do you like pytest? Share some love on Twitter or in your blog posts!

We'd also like to hear about your propositions and suggestions. Feel free to [submit them as issues](#) and:

- Explain in detail how they should work.
- Keep the scope as narrow as possible. This will make it easier to implement.

5.6.2 Report bugs

Report bugs for pytest in the [issue tracker](#).

If you are reporting a bug, please include:

- Your operating system name and version.
- Any details about your local setup that might be helpful in troubleshooting, specifically the Python interpreter version, installed libraries, and pytest version.
- Detailed steps to reproduce the bug.

If you can write a demonstration test that currently fails but should pass (xfail), that is a very useful commit to make as well, even if you cannot fix the bug itself.

5.6.3 Fix bugs

Look through the [GitHub issues for bugs](#).

Talk to developers to find out how you can fix specific bugs. To indicate that you are going to work on a particular issue, add a comment to that effect on the specific issue.

Don't forget to check the issue trackers of your favourite plugins, too!

5.6.4 Implement features

Look through the [GitHub issues for enhancements](#).

Talk to developers to find out how you can implement specific features.

5.6.5 Write documentation

Pytest could always use more documentation. What exactly is needed?

- More complementary documentation. Have you perhaps found something unclear?
- Documentation translations. We currently have only English.
- Docstrings. There can never be too many of them.
- Blog posts, articles and such – they're all very appreciated.

You can also edit documentation files directly in the GitHub web interface, without using a local copy. This can be convenient for small fixes.

Note: Build the documentation locally with the following command:

```
$ tox -e docs
```

The built documentation should be available in `doc/en/_build/html`, where 'en' refers to the documentation language.

Pytest has an API reference which in large part is [generated automatically](#) from the docstrings of the documented items. Pytest uses the [Sphinx docstring format](#). For example:


```

def my_function(arg: ArgType) -> Foo:
    """Do important stuff.

    More detailed info here, in separate paragraphs from the subject line.
    Use proper sentences -- start sentences with capital letters and end
    with periods.

    Can include annotated documentation:

    :param short_arg: An argument which determines stuff.
    :param long_arg:
        A long explanation which spans multiple lines, overflows
        like this.
    :returns: The result.
    :raises ValueError:
        Detailed information when this can happen.

    .. versionadded:: 6.0

    Including types into the annotations above is not necessary when
    type-hinting is being used (as in this example).
    """

```

5.6.6 Submitting Plugins to pytest-dev

Pytest development of the core, some plugins and support code happens in repositories living under the `pytest-dev` organisations:

- [pytest-dev on GitHub](#)

All pytest-dev Contributors team members have write access to all contained repositories. Pytest core and plugins are generally developed using *pull requests* to respective repositories.

The objectives of the `pytest-dev` organisation are:

- Having a central location for popular pytest plugins
- Sharing some of the maintenance responsibility (in case a maintainer no longer wishes to maintain a plugin)

You can submit your plugin by subscribing to the [pytest-dev mail list](#) and writing a mail pointing to your existing pytest plugin repository which must have the following:

- PyPI presence with packaging metadata that contains a `pytest-` prefixed name, version number, authors, short and long description.
- a [tox configuration](#) for running tests using `tox`.
- a README describing how to use the plugin and on which platforms it runs.
- a LICENSE file containing the licensing information, with matching info in its packaging metadata.
- an issue tracker for bug reports and enhancement requests.
- a [changelog](#).

If no contributor strongly objects and two agree, the repository can then be transferred to the `pytest-dev` organisation.

Here's a rundown of how a repository transfer usually proceeds (using a repository named `joedoe/pytest-xyz` as example):

- `joedoe` transfers repository ownership to `pytest-dev` administrator `calvin`.

- calvin creates `pytest-xyz-admin` and `pytest-xyz-developers` teams, inviting `joedoe` to both as **maintainer**.
- calvin transfers repository to `pytest-dev` and configures team access:
 - `pytest-xyz-admin` **admin** access;
 - `pytest-xyz-developers` **write** access;

The `pytest-dev/Contributors` team has write access to all projects, and every project administrator is in it. We recommend that each plugin has at least three people who have the right to release to PyPI.

Repository owners can rest assured that no `pytest-dev` administrator will ever make releases of your repository or take ownership in any way, except in rare cases where someone becomes unresponsive after months of contact attempts. As stated, the objective is to share maintenance and avoid “plugin-abandon”.

5.6.7 Preparing Pull Requests

Short version

1. Fork the repository.
2. Enable and install `pre-commit` to ensure style-guides and code checks are followed.
3. Follow **PEP-8** for naming and `black` for formatting.
4. Tests are run using `tox`:

```
tox -e linting,py37
```

The test environments above are usually enough to cover most cases locally.

5. Write a changelog entry: `changelog/2574.bugfix.rst`, use issue id number and one of `feature`, `improvement`, `bugfix`, `doc`, `deprecation`, `breaking`, `vendor` or `trivial` for the issue type.
6. Unless your change is a trivial or a documentation fix (e.g., a typo or reword of a small section) please add yourself to the `AUTHORS` file, in alphabetical order.

Long version

What is a “pull request”? It informs the project’s core developers about the changes you want to review and merge. Pull requests are stored on [GitHub servers](#). Once you send a pull request, we can discuss its potential modifications and even add more commits to it later on. There’s an excellent tutorial on how Pull Requests work in the [GitHub Help Center](#).

Here is a simple overview, with `pytest`-specific bits:

1. Fork the [pytest GitHub repository](#). It’s fine to use `pytest` as your fork repository name because it will live under your user.
2. Clone your fork locally using `git` and create a branch:

```
$ git clone git@github.com:YOUR_GITHUB_USERNAME/pytest.git
$ cd pytest
# now, create your own branch off "main":

$ git checkout -b your-bugfix-branch-name main
```

Given we have “major.minor.micro” version numbers, bug fixes will usually be released in micro releases whereas features will be released in minor releases and incompatible changes in major releases.

If you need some help with Git, follow this quick start guide: <https://git.wiki.kernel.org/index.php/QuickStart>

3. Install `pre-commit` and its hook on the pytest repo:

```
$ pip install --user pre-commit
$ pre-commit install
```

Afterwards `pre-commit` will run whenever you commit.

<https://pre-commit.com/> is a framework for managing and maintaining multi-language pre-commit hooks to ensure code-style and code formatting is consistent.

4. Install `tox`

`Tox` is used to run all the tests and will automatically setup `virtualenvs` to run the tests in. (will implicitly use <https://virtualenv.pypa.io/en/latest/>):

```
$ pip install tox
```

5. Run all the tests

You need to have Python 3.7 available in your system. Now running tests is as simple as issuing this command:

```
$ tox -e linting,py37
```

This command will run tests via the “`tox`” tool against Python 3.7 and also perform “`lint`” coding-style checks.

6. You can now edit your local working copy and run the tests again as necessary. Please follow PEP-8 for naming.

You can pass different options to `tox`. For example, to run tests on Python 3.7 and pass options to `pytest` (e.g. enter `pdb` on failure) to `pytest` you can do:

```
$ tox -e py37 -- --pdb
```

Or to only run tests in a particular test module on Python 3.7:

```
$ tox -e py37 -- testing/test_config.py
```

When committing, `pre-commit` will re-format the files if necessary.

7. If instead of using `tox` you prefer to run the tests directly, then we suggest to create a virtual environment and use an editable install with the `testing` extra:

```
$ python3 -m venv .venv
$ source .venv/bin/activate # Linux
$ .venv/Scripts/activate.bat # Windows
$ pip install -e ".[testing]"
```

Afterwards, you can edit the files and run `pytest` normally:

```
$ pytest testing/test_config.py
```

8. Create a new changelog entry in `changelog`. The file should be named `<issueid>.<type>.rst`, where *issueid* is the number of the issue related to the change and *type* is one of `feature`, `improvement`, `bugfix`, `doc`, `deprecation`, `breaking`, `vendor` or `trivial`. You may skip creating the changelog entry if the change doesn’t affect the documented behaviour of `pytest`.
9. Add yourself to `AUTHORS` file if not there yet, in alphabetical order.

10. Commit and push once your tests pass and you are happy with your change(s):

```
$ git commit -a -m "<commit message>"
$ git push -u
```

11. Finally, submit a pull request through the GitHub website using this data:

```
head-fork: YOUR_GITHUB_USERNAME/pytest
compare: your-branch-name

base-fork: pytest-dev/pytest
base: main
```

Writing Tests

Writing tests for plugins or for pytest itself is often done using the `pytester fixture`, as a “black-box” test.

For example, to ensure a simple test passes you can write:

```
def test_true_assertion(pytester):
    pytester.makepyfile(
        """
        def test_foo():
            assert True
        """
    )
    result = pytester.runpytest()
    result.assert_outcomes(failed=0, passed=1)
```

Alternatively, it is possible to make checks based on the actual output of the terminal using *glob-like* expressions:

```
def test_true_assertion(pytester):
    pytester.makepyfile(
        """
        def test_foo():
            assert False
        """
    )
    result = pytester.runpytest()
    result.stdout.fnmatch_lines(["*assert False*", "*1 failed*"])
```

When choosing a file where to write a new test, take a look at the existing files and see if there’s one file which looks like a good fit. For example, a regression test about a bug in the `--lf` option should go into `test_cacheprovider.py`, given that this option is implemented in `cacheprovider.py`. If in doubt, go ahead and open a PR with your best guess and we can discuss this over the code.

5.6.8 Joining the Development Team

Anyone who has successfully seen through a pull request which did not require any extra work from the development team to merge will themselves gain commit access if they so wish (if we forget to ask please send a friendly reminder). This does not mean there is any change in your contribution workflow: everyone goes through the same pull-request-and-review process and no-one merges their own pull requests unless already approved. It does however mean you can participate in the development process more fully since you can merge pull requests from other contributors yourself after having reviewed them.

5.6.9 Backporting bug fixes for the next patch release

Pytest makes feature release every few weeks or months. In between, patch releases are made to the previous feature release, containing bug fixes only. The bug fixes usually fix regressions, but may be any change that should reach users before the next feature release.

Suppose for example that the latest release was 1.2.3, and you want to include a bug fix in 1.2.4 (check <https://github.com/pytest-dev/pytest/releases> for the actual latest release). The procedure for this is:

1. First, make sure the bug is fixed the `main` branch, with a regular pull request, as described above. An exception to this is if the bug fix is not applicable to `main` anymore.
2. `git checkout origin/1.2.x -b backport-XXXX # use the main PR number here`
3. Locate the merge commit on the PR, in the *merged* message, for example:


```
nicoddemus merged commit 0f8b462 into pytest-dev:main
```
4. `git cherry-pick -x -m1 REVISION # use the revision you found above (0f8b462).`
5. Open a PR targeting `1.2.x`:
 - Prefix the message with `[1.2.x]`.
 - Delete the PR body, it usually contains a duplicate commit message.

Who does the backporting

As mentioned above, bugs should first be fixed on `main` (except in rare occasions that a bug only happens in a previous release). So, who should do the backport procedure described above?

1. If the bug was fixed by a core developer, it is the main responsibility of that core developer to do the backport.
2. However, often the merge is done by another maintainer, in which case it is nice of them to do the backport procedure if they have the time.
3. For bugs submitted by non-maintainers, it is expected that a core developer will do the backport, normally the one that merged the PR on `main`.
4. If a non-maintainer notices a bug which is fixed on `main` but has not been backported (due to maintainers forgetting to apply the *needs backport* label, or just plain missing it), they are also welcome to open a PR with the backport. The procedure is simple and really helps with the maintenance of the project.

All the above are not rules, but merely some guidelines/suggestions on what we should expect about backports.

5.6.10 Handling stale issues/PRs

Stale issues/PRs are those where pytest contributors have asked for questions/changes and the authors didn't get around to answer/implement them yet after a somewhat long time, or the discussion simply died because people seemed to lose interest.

There are many reasons why people don't answer questions or implement requested changes: they might get busy, lose interest, or just forget about it, but the fact is that this is very common in open source software.

The pytest team really appreciates every issue and pull request, but being a high-volume project with many issues and pull requests being submitted daily, we try to reduce the number of stale issues and PRs by regularly closing them. When an issue/pull request is closed in this manner, it is by no means a dismissal of the topic being tackled by the issue/pull request, but it is just a way for us to clear up the queue and make the maintainers' work more manageable. Submitters can always reopen the issue/pull request in their own time later if it makes sense.

When to close

Here are a few general rules the maintainers use deciding when to close issues/PRs because of lack of inactivity:

- Issues labeled `question` or `needs information`: closed after 14 days inactive.
- Issues labeled `proposal`: closed after six months inactive.
- Pull requests: after one month, consider pinging the author, update linked issue, or consider closing. For pull requests which are nearly finished, the team should consider finishing it up and merging it.

The above are **not hard rules**, but merely **guidelines**, and can be (and often are!) reviewed on a case-by-case basis.

Closing pull requests

When closing a Pull Request, it needs to be acknowledging the time, effort, and interest demonstrated by the person which submitted it. As mentioned previously, it is not the intent of the team to dismiss a stalled pull request entirely but to merely to clear up our queue, so a message like the one below is warranted when closing a pull request that went stale:

Hi <contributor>,

First of all, we would like to thank you for your time and effort on working on this, the pytest team deeply appreciates it.

We noticed it has been awhile since you have updated this PR, however. pytest is a high activity project, with many issues/PRs being opened daily, so it is hard for us maintainers to track which PRs are ready for merging, for review, or need more attention.

So for those reasons we, think it is best to close the PR for now, but with the only intention to clean up our queue, it is by no means a rejection of your changes. We still encourage you to re-open this PR (it is just a click of a button away) when you are ready to get back to it.

Again we appreciate your time for working on this, and hope you might get back to this at a later time!

<bye>

5.6.11 Closing Issues

When a pull request is submitted to fix an issue, add text like `closes #XYZW` to the PR description and/or commits (where `XYZW` is the issue number). See the [GitHub docs](#) for more information.

When an issue is due to user error (e.g. misunderstanding of a functionality), please politely explain to the user why the issue raised is really a non-issue and ask them to close the issue if they have no further questions. If the original requestor is unresponsive, the issue will be handled as described in the section *Handling stale issues/PRs* above.

5.7 Development Guide

The contributing guidelines are to be found [here](#). The release procedure for pytest is documented on [GitHub](#).

5.8 Sponsor

pytest is maintained by a team of volunteers from all around the world in their free time. While we work on pytest because we love the project and use it daily at our daily jobs, monetary compensation when possible is welcome to justify time away from friends, family and personal time.

Money is also used to fund local sprints, merchandising (stickers to distribute in conferences for example) and every few years a large sprint involving all members.

5.8.1 OpenCollective

[Open Collective](#) is an online funding platform for open and transparent communities. It provide tools to raise money and share your finances in full transparency.

It is the platform of choice for individuals and companies that want to make one-time or monthly donations directly to the project.

See more details in the [pytest collective](#).

5.9 pytest for enterprise

Tidelift is working with the maintainers of pytest and thousands of other open source projects to deliver commercial support and maintenance for the open source dependencies you use to build your applications. Save time, reduce risk, and improve code health, while paying the maintainers of the exact dependencies you use.

[Get more details](#)

The Tidelift Subscription is a managed open source subscription for application dependencies covering millions of open source projects across JavaScript, Python, Java, PHP, Ruby, .NET, and more.

Your subscription includes:

- **Security updates**
 - Tidelift’s security response team coordinates patches for new breaking security vulnerabilities and alerts immediately through a private channel, so your software supply chain is always secure.
- **Licensing verification and indemnification**

- Tidelift verifies license information to enable easy policy enforcement and adds intellectual property indemnification to cover creators and users in case something goes wrong. You always have a 100% up-to-date bill of materials for your dependencies to share with your legal team, customers, or partners.

- **Maintenance and code improvement**

- Tidelift ensures the software you rely on keeps working as long as you need it to work. Your managed dependencies are actively maintained and we recruit additional maintainers where required.

- **Package selection and version guidance**

- Tidelift helps you choose the best open source packages from the start—and then guide you through updates to stay on the best releases as new issues arise.

- **Roadmap input**

- Take a seat at the table with the creators behind the software you use. Tidelift’s participating maintainers earn more income as their software is used by more subscribers, so they’re interested in knowing what you need.

- **Tooling and cloud integration**

- Tidelift works with GitHub, GitLab, BitBucket, and every cloud platform (and other deployment targets, too).

The end result? All of the capabilities you expect from commercial-grade software, for the full breadth of open source you use. That means less time grappling with esoteric open source trivia, and more time building your own applications—and your business.

[Request a demo](#)

5.10 License

Distributed under the terms of the [MIT license](#), pytest is free and open source software.

```
The MIT License (MIT)
```

```
Copyright (c) 2004–2021 Holger Krekel and others
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy of
this software and associated documentation files (the "Software"), to deal in
the Software without restriction, including without limitation the rights to
use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies
of the Software, and to permit persons to whom the Software is furnished to do
so, subject to the following conditions:
```

```
The above copyright notice and this permission notice shall be included in all
copies or substantial portions of the Software.
```

```
THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR
IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY,
FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE
AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER
LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM,
OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE
SOFTWARE.
```


5.11 Contact channels

- [pytest issue tracker](#) to report bugs or suggest features (for version 2.0 and above).
- [pytest discussions at github](#) for general questions.
- [pytest discord server](#) for pytest development visibility and general assistance.
- [pytest on stackoverflow.com](#) to post precise questions with the tag `pytest`. New Questions will usually be seen by pytest users or developers and answered quickly.
- [Testing In Python](#): a mailing list for Python testing tools and discussion.
- [pytest-dev at python.org \(mailing list\)](#) pytest specific announcements and discussions.
- [contribution guide](#) for help on submitting pull requests to GitHub.
- `#pytest` on [irc.libera.chat](#) IRC channel for random questions (using an IRC client, [via webchat](#), or [via Matrix](#)).
- private mail to [Holger.Krekel at gmail com](#) if you want to communicate sensitive issues
- [merlinux.eu](#) offers pytest and tox-related professional teaching and consulting.

5.12 History

pytest has a long and interesting history. The [first commit](#) in this repository is from January 2007, and even that commit alone already tells a lot: The repository originally was from the `py` library (later split off to `pytest`), and it originally was a SVN revision, migrated to Mercurial, and finally migrated to git.

However, the commit says “create the new development trunk” and is already quite big: *435 files changed, 58640 insertions(+)*. This is because `pytest` originally was born as part of `PyPy`, to make it easier to write tests for it. Here’s how it evolved from there to its own project:

- Late 2002 / early 2003, `PyPy` was born.
- Like that blog post mentioned, from very early on, there was a big focus on testing. There were various `testsupport` files on top of `unittest.py`, and as early as June 2003, Holger Krekel ([@hpk42](#)) [refactored](#) its test framework to clean things up (`pypy.tool.test`, but still on top of `unittest.py`, with nothing `pytest`-like yet).
- In December 2003, there was [another iteration](#) at improving their testing situation, by Stefan Schwarzer, called `pypy.tool.newtest`.
- However, it didn’t seem to be around for long, as around June/July 2004, efforts started on a thing called `utest`, offering plain assertions. This seems like the start of something `pytest`-like, but unfortunately, it’s unclear where the test runner’s code was at the time. The closest thing still around is [this file](#), but that doesn’t seem like a complete test runner at all. What can be seen is that there were [various efforts](#) by Laura Creighton and Samuele Pedroni ([@pedronis](#)) at automatically converting existing tests to the new `utest` framework.
- Around the same time, for Europython 2004, [@hpk42 started a project](#) originally called “std”, intended to be a “complementary standard library” - already laying out the principles behind what later became `pytest`:
 - current “batteries included” are very useful, but
 - * some of them are written in a pretty much java-like style, especially the `unittest`-framework
 - * [...]
 - * the best API is one that doesn’t exist
 - [...]
 - a testing package should require as few boilerplate code as possible and offer much flexibility

- it should provide premium quality tracebacks and debugging aid

[...]

- first of all ... forget about limited “assertXYZ APIs” and use the real thing, e.g.:

```
assert x == y
```

- this works with plain python but you get unhelpful “assertion failed” errors with no information
 - `std.utest` (magic!) actually reinterprets the assertion expression and offers detailed information about underlying values
- In September 2004, the `py-dev` mailinglist gets born, which is now `pytest-dev`, but thankfully with all the original archives still intact.
 - Around September/October 2004, the `std` project was renamed to `py` and `std.utest` became `py.test`. This is also the first time the entire source code, seems to be available, with much of the API still being around today:
 - `py.path.local`, which is being phased out of `pytest` (in favour of `pathlib`) some 16-17 years later
 - The idea of the collection tree, including `Collector`, `FSCollector`, `Directory`, `PyCollector`, `Module`, `Class`
 - Arguments like `-x / --exitfirst`, `-l / --showlocals`, `--fulltrace`, `--pdb`, `-S / --nocapture` (`-s / --capture=off` today), `--collectonly` (`--collect-only` today)
 - In the same month, the `py` library gets split off from `PyPy`
 - It seemed to get rather quiet for a while, and little seemed to happen between October 2004 (removing `py` from `PyPy`) and January 2007 (first commit in the now-`pytest` repository). However, there were various discussions about features/ideas on the mailinglist, and a couple of releases every couple of months:
 - March 2006: `py 0.8.0-alpha2`
 - May 2007: `py 0.9.0`
 - March 2008: `py 0.9.1` (first release to be found in the `pytest changelog!`)
 - August 2008: `py 0.9.2`
 - In August 2009, `py 1.0.0` was released, introducing a lot of fundamental features:
 - `funcargs/fixtures`
 - A plugin architecture which still looks very much the same today!
 - Various default plugins, including `monkeypatch`
 - Even back there, the `FAQ` said:

Clearly, [a second standard library] was ambitious and the naming has maybe haunted the project rather than helping it. There may be a project name change and possibly a split up into different projects sometime.

and that finally happened in November 2010, when `pytest 2.0.0` was released as a package separate from `py` (but still called `py.test`).
 - In August 2016, `pytest 3.0.0` was released, which adds `pytest` (rather than `py.test`) as the recommended command-line entry point

Due to this history, it’s difficult to answer the question when `pytest` was started. It depends what point should really be seen as the start of it all. One possible interpretation is to pick Europython 2004, i.e. around June/July 2004.

5.13 Historical Notes

This page lists features or behavior from previous versions of pytest which have changed over the years. They are kept here as a historical note so users looking at old code can find documentation related to them.

5.13.1 Marker revamp and iteration

Changed in version 3.6.

pytest's marker implementation traditionally worked by simply updating the `__dict__` attribute of functions to cumulatively add markers. As a result, markers would unintentionally be passed along class hierarchies in surprising ways. Further, the API for retrieving them was inconsistent, as markers from parameterization would be stored differently than markers applied using the `@pytest.mark` decorator and markers added via `node.add_marker`.

This state of things made it technically next to impossible to use data from markers correctly without having a deep understanding of the internals, leading to subtle and hard to understand bugs in more advanced usages.

Depending on how a marker got declared/changed one would get either a `MarkerInfo` which might contain markers from sibling classes, `MarkDecorators` when marks came from parameterization or from a `node.add_marker` call, discarding prior marks. Also `MarkerInfo` acts like a single mark, when it in fact represents a merged view on multiple marks with the same name.

On top of that markers were not accessible in the same way for modules, classes, and functions/methods. In fact, markers were only accessible in functions, even if they were declared on classes/modules.

A new API to access markers has been introduced in pytest 3.6 in order to solve the problems with the initial design, providing the `_pytest.nodes.Node.iter_markers()` method to iterate over markers in a consistent manner and reworking the internals, which solved a great deal of problems with the initial design.

Updating code

The old `Node.get_marker(name)` function is considered deprecated because it returns an internal `MarkerInfo` object which contains the merged name, `*args` and `**kwargs` of all the markers which apply to that node.

In general there are two scenarios on how markers should be handled:

1. Marks overwrite each other. Order matters but you only want to think of your mark as a single item. E.g. `log_level('info')` at a module level can be overwritten by `log_level('debug')` for a specific test.

In this case, use `Node.get_closest_marker(name)`:

```
# replace this:
marker = item.get_marker("log_level")
if marker:
    level = marker.args[0]

# by this:
marker = item.get_closest_marker("log_level")
if marker:
    level = marker.args[0]
```

2. Marks compose in an additive manner. E.g. `skipif(condition)` marks mean you just want to evaluate all of them, order doesn't even matter. You probably want to think of your marks as a set here.

In this case iterate over each mark and handle their `*args` and `**kwargs` individually.

```
# replace this
skipif = item.get_marker("skipif")
if skipif:
    for condition in skipif.args:
        # eval condition
        ...

# by this:
for skipif in item.iter_markers("skipif"):
    condition = skipif.args[0]
    # eval condition
```

If you are unsure or have any questions, please consider opening an issue.

Related issues

Here is a non-exhaustive list of issues fixed by the new implementation:

- Marks don't pick up nested classes (#199).
- Markers stain on all related classes (#568).
- Combining marks - args and kwargs calculation (#2897).
- `request.node.get_marker('name')` returns `None` for markers applied in classes (#902).
- Marks applied in parametrize are stored as `markdecorator` (#2400).
- Fix marker interaction in a backward incompatible way (#1670).
- Refactor marks to get rid of the current "marks transfer" mechanism (#2363).
- Introduce `FunctionDefinition` node, use it in `generate_tests` (#2522).
- Remove named marker attributes and collect markers in items (#891).
- skipif mark from parametrize hides module level skipif mark (#1540).
- skipif + parametrize not skipping tests (#1296).
- Marker transfer incompatible with inheritance (#535).

More details can be found in the [original PR](#).

Note: in a future major release of pytest we will introduce class based markers, at which point markers will no longer be limited to instances of `Mark`.

5.13.2 cache plugin integrated into the core

The functionality of the *core cache* plugin was previously distributed as a third party plugin named `pytest-cache`. The core plugin is compatible regarding command line options and API usage except that you can only store/receive data between test runs that is json-serializable.

5.13.3 funcargs and `pytest_funcarg__`

In versions prior to 2.3 there was no `@pytest.fixture` marker and you had to use a magic `pytest_funcarg__NAME` prefix for the fixture factory. This remains and will remain supported but is not anymore advertised as the primary means of declaring fixture functions.

5.13.4 `@pytest.yield_fixture` decorator

Prior to version 2.10, in order to use a `yield` statement to execute teardown code one had to mark a fixture using the `yield_fixture` marker. From 2.10 onward, normal fixtures can use `yield` directly so the `yield_fixture` decorator is no longer needed and considered deprecated.

5.13.5 `[pytest]` header in `setup.cfg`

Prior to 3.0, the supported section name was `[pytest]`. Due to how this may collide with some distutils commands, the recommended section name for `setup.cfg` files is now `[tool:pytest]`.

Note that for `pytest.ini` and `tox.ini` files the section name is `[pytest]`.

5.13.6 Applying marks to `@pytest.mark.parametrize` parameters

Prior to version 3.1 the supported mechanism for marking values used the syntax:

```
import pytest

@pytest.mark.parametrize(
    "test_input,expected", [
        ("3+5", 8), ("2+4", 6), pytest.mark.xfail(("6*9", 42))]
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

This was an initial hack to support the feature but soon was demonstrated to be incomplete, broken for passing functions or applying multiple marks with the same name but different parameters.

The old syntax is planned to be removed in `pytest-4.0`.

5.13.7 `@pytest.mark.parametrize` argument names as a tuple

In versions prior to 2.4 one needed to specify the argument names as a tuple. This remains valid but the simpler `"name1, name2, ..."` comma-separated-string syntax is now advertised first because it's easier to write and produces less line noise.

5.13.8 setup: is now an “autouse fixture”

During development prior to the pytest-2.3 release the name `pytest.setup` was used but before the release it was renamed and moved to become part of the general fixture mechanism, namely *Autouse fixtures (fixtures you don't have to request)*

5.13.9 Conditions as strings instead of booleans

Prior to pytest-2.4 the only way to specify skipif/xfail conditions was to use strings:

```
import sys

@pytest.mark.skipif("sys.version_info >= (3,3)")
def test_function():
    ...
```

During test function setup the skipif condition is evaluated by calling `eval('sys.version_info >= (3,0)', namespace)`. The namespace contains all the module globals, and `os` and `sys` as a minimum.

Since pytest-2.4 *boolean conditions* are considered preferable because markers can then be freely imported between test modules. With strings you need to import not only the marker but all variables used by the marker, which violates encapsulation.

The reason for specifying the condition as a string was that pytest can report a summary of skip conditions based purely on the condition string. With conditions as booleans you are required to specify a *reason* string.

Note that string conditions will remain fully supported and you are free to use them if you have no need for cross-importing markers.

The evaluation of a condition string in `pytest.mark.skipif(conditionstring)` or `pytest.mark.xfail(conditionstring)` takes place in a namespace dictionary which is constructed as follows:

- the namespace is initialized by putting the `sys` and `os` modules and the `pytest config` object into it.
- updated with the module globals of the test function for which the expression is applied.

The `pytest config` object allows you to skip based on a test configuration value which you might have added:

```
@pytest.mark.skipif("not config.getvalue('db')")
def test_function():
    ...
```

The equivalent with “boolean conditions” is:

```
@pytest.mark.skipif(not pytest.config.getvalue("db"), reason="--db was not specified")
def test_function():
    pass
```

Note: You cannot use `pytest.config.getvalue()` in code imported before pytest’s argument parsing takes place. For example, `conftest.py` files are imported before command line parsing and thus `config.getvalue()` will not execute correctly.

5.13.10 `pytest.set_trace()`

Previous to version 2.4 to set a break point in code one needed to use `pytest.set_trace()`:

```
import pytest

def test_function():
    ...
    pytest.set_trace() # invoke PDB debugger and tracing
```

This is no longer needed and one can use the native `import pdb;pdb.set_trace()` call directly.

For more details see breakpoints.

5.13.11 “compat” properties

Access of `Module`, `Function`, `Class`, `Instance`, `File` and `Item` through `Node` instances have long been documented as deprecated, but started to emit warnings from `pytest 3.9` and onward.

Users should just `import pytest` and access those objects using the `pytest` module.

5.14 Talks and Tutorials

5.14.1 Books

- `pytest Quick Start Guide`, by Bruno Oliveira (2018).
- `Python Testing with pytest`, by Brian Okken (2017).

5.14.2 Talks and blog postings

- Webinar: `pytest: Test Driven Development für Python (German)`, Florian Bruhin, via `mylearning.ch`, 2020
- Webinar: `Simplify Your Tests with Fixtures`, Oliver Bestwalter, via `JetBrains`, 2020
- Training: `Introduction to pytest - simple, rapid and fun testing with Python`, Florian Bruhin, `PyConDE 2019`
- Abridged metaprogramming classics - this episode: `pytest`, Oliver Bestwalter, `PyConDE 2019` ([repository](#), [recording](#))
- `Testing PySide/PyQt code easily using the pytest framework`, Florian Bruhin, `Qt World Summit 2019` ([slides](#), [recording](#))
- `pytest: recommendations, basic packages for testing in Python and Django`, Andreu Vallbona, `PyBCN June 2019`.
- `pytest: recommendations, basic packages for testing in Python and Django`, Andreu Vallbona, `PyconES 2017` ([slides in english](#), [video in spanish](#))
- `pytest advanced`, Andrew Svetlov (Russian, `PyCon Russia`, 2016).
- `Pythonic testing`, Igor Starikov (Russian, `PyNsk`, November 2016).
- `pytest - Rapid Simple Testing`, Florian Bruhin, `Swiss Python Summit 2016`.
- `Improve your testing with Pytest and Mock`, Gabe Hollombe, `PyCon SG 2015`.
- `Introduction to pytest`, Andreas Pelme, `EuroPython 2014`.

- Advanced Uses of py.test Fixtures, Floris Bruynooghe, EuroPython 2014.
- Why i use py.test and maybe you should too, Andy Todd, Pycon AU 2013
- 3-part blog series about pytest from @pydanny alias Daniel Greenfeld (January 2014)
- pytest: helps you write better Django apps, Andreas Pelme, DjangoCon Europe 2014.
- Testing Django Applications with pytest, Andreas Pelme, EuroPython 2013.
- Testes pythonics com py.test, Vinicius Belchior Assef Neto, Plone Conf 2013, Brazil.
- Introduction to py.test fixtures, FOSDEM 2013, Floris Bruynooghe.
- pytest feature and release highlights, Holger Krekel (GERMAN, October 2013)
- pytest introduction from Brian Okken (January 2013)
- pycon australia 2012 pytest talk from Brianna Laughner (video, slides, code)
- pycon 2012 US talk video from Holger Krekel
- monkey patching done right (blog post, consult monkeypatch plugin for up-to-date API)

Test parametrization:

- generating parametrized tests with fixtures.
- test generators and cached setup
- parametrizing tests, generalized (blog post)
- putting test-hooks into local or global plugins (blog post)

Assertion introspection:

- (07/2011) Behind the scenes of pytest's new assertion rewriting

Distributed testing:

- simultaneously test your code on all platforms (blog entry)

Plugin specific examples:

- skipping slow tests by default in pytest (blog entry)
- many examples in the docs for plugins

Non-alphabetical

__contains__() (*Stash method*), 229
 __delitem__() (*Stash method*), 229
 __getitem__() (*Stash method*), 228
 __len__() (*Stash method*), 229
 __setitem__() (*Stash method*), 228
 __str__() (*LineMatcher method*), 183

A

add_cleanup() (*Config method*), 209
 add_hookcall_monitoring() (*PytestPluginManager method*), 224
 add_hookspecs() (*PytestPluginManager method*), 224
 add_marker() (*Node method*), 220
 add_report_section() (*Item method*), 214
 addfinalizer() (*FixtureRequest method*), 168
 addfinalizer() (*Node method*), 220
 addini() (*Parser method*), 222
 addinvalue_line() (*Config method*), 209
 adoption() (*OptionGroup method*), 223
 adoption() (*Parser method*), 221
 adopts

- configuration value, 233

 applymarker() (*FixtureRequest method*), 168
 approx() (*in module pytest*), 152
 args (*Config.InvocationParams attribute*), 208
 args (*Mark attribute*), 216
 args (*MarkDecorator property*), 215
 assert_outcomes() (*RunResult method*), 183
 at_level() (*LogCaptureFixture method*), 172

B

built-in function

- pytest.mark.filterwarnings(), 160
- pytest.mark.skip(), 161
- pytest.mark.skipif(), 161
- pytest.mark.usefixtures(), 161
- pytest.mark.xfail(), 162

C

cache

fixture, 164
 Cache (*class in pytest*), 164
 cache_dir

- configuration value, 233

 CallInfo (*class in pytest*), 205
 capfd

- fixture, 166

 capfd() (*in module _pytest.capture*), 167
 capfdbinary

- fixture, 167

 capfdbinary() (*in module _pytest.capture*), 167
 caplog

- fixture, 170

 caplog (*CollectReport property*), 207
 caplog (*TestReport property*), 227
 caplog() (*in module _pytest.logging*), 170
 capstderr (*CollectReport property*), 207
 capstderr (*TestReport property*), 227
 capstdout (*CollectReport property*), 207
 capstdout (*TestReport property*), 227
 capsys

- fixture, 165

 capsys() (*in module _pytest.capture*), 166
 capsysbinary

- fixture, 166

 capsysbinary() (*in module _pytest.capture*), 166
 CaptureFixture (*class in pytest*), 166
 chdir() (*MonkeyPatch method*), 174
 chdir() (*Pytester method*), 175
 chdir() (*Testdir method*), 186
 check_pending() (*PytestPluginManager method*), 224
 Class (*class in pytest*), 206
 clear() (*LogCaptureFixture method*), 171
 clear() (*WarningsRecorder method*), 189
 cls (*FixtureRequest property*), 168
 cls (*Metafunc attribute*), 217
 collect() (*Class method*), 206
 collect() (*Collector method*), 206
 collect() (*Module method*), 218
 collect() (*Pytester.Session method*), 177
 collect() (*Session method*), 226

`collect()` (*Testdir.Session method*), 185
`collect_by_name()` (*Pytester method*), 180
`collect_by_name()` (*Testdir method*), 188
`collect_ignore`
 global variable interpreted by
 pytest, 230
`collect_ignore_glob`
 global variable interpreted by
 pytest, 230
`Collector` (*class in pytest*), 206
`Collector.CollectError`, 206
`CollectReport` (*class in _pytest.reports*), 206
`combined_with()` (*Mark method*), 216
`Config` (*class in pytest*), 208
`config` (*FixtureRequest property*), 168
`config` (*Metafunc attribute*), 217
`Config.InvocationParams` (*class in pytest*), 208
configuration value
 `addopts`, 233
 `cache_dir`, 233
 `console_output_style`, 233
 `doctest_encoding`, 233
 `doctest_optionflags`, 233
 `empty_parameter_set_mark`, 233
 `faulthandler_timeout`, 234
 `filterwarnings`, 234
 `junit_duration_report`, 234
 `junit_family`, 234
 `junit_log_passing_tests`, 235
 `junit_logging`, 235
 `junit_suite_name`, 235
 `log_auto_indent`, 235
 `log_cli`, 236
 `log_cli_date_format`, 236
 `log_cli_format`, 236
 `log_cli_level`, 236
 `log_date_format`, 236
 `log_file`, 236
 `log_file_date_format`, 236
 `log_file_format`, 236
 `log_file_level`, 237
 `log_format`, 237
 `log_level`, 237
 `markers`, 237
 `minversion`, 237
 `norecursedirs`, 237
 `python_classes`, 238
 `python_files`, 238
 `python_functions`, 238
 `pythonpath`, 239
 `required_plugins`, 239
 `testpaths`, 239
 `usefixtures`, 239
 `xfail_strict`, 239

`console_output_style`
 configuration value, 233
`context()` (*MonkeyPatch class method*), 172
`copy_example()` (*Pytester method*), 177
`copy_example()` (*Testdir method*), 187
`count_towards_summary` (*CollectReport property*),
 207
`count_towards_summary` (*TestReport property*), 227

D

`definition` (*Metafunc attribute*), 217
`delattr()` (*MonkeyPatch method*), 173
`delenv()` (*MonkeyPatch method*), 174
`delitem()` (*MonkeyPatch method*), 173
`deprecated_call()` (*in module pytest*), 159
`dir` (*Config.InvocationParams attribute*), 208
`disabled()` (*CaptureFixture method*), 166
`doctest_encoding`
 configuration value, 233
`doctest_namespace`
 fixture, 167
`doctest_namespace()` (*in module _pytest.doctest*),
 167
`doctest_optionflags`
 configuration value, 233
`duration` (*CallInfo attribute*), 205
`duration` (*RunResult attribute*), 183
`duration` (*TestReport attribute*), 227

E

`empty_parameter_set_mark`
 configuration value, 233
`enable_tracing()` (*PytestPluginManager method*),
 224
environment variable
 `FORCE_COLOR`, 231
 `NO_COLOR`, 231
 `PY_COLORS`, 231
 `PYTEST_ADDOPTS`, 230
 `PYTEST_CURRENT_TEST`, 230, 255, 289
 `PYTEST_DEBUG`, 231
 `PYTEST_DISABLE_PLUGIN_AUTOLOAD`, 231
 `PYTEST_PLUGINS`, 89, 231
 `PYTEST_THEME`, 149, 231
 `PYTEST_THEME_MODE`, 149, 231
 `python:PYTHONWARNINGS`, 78
 `PYTHONPATH`, 239
`errisinstance()` (*ExceptionInfo method*), 211
`errlines` (*RunResult attribute*), 182
`ExceptionInfo` (*class in pytest*), 210
`excinfo` (*CallInfo attribute*), 205
`exconly()` (*ExceptionInfo method*), 211
`exit()` (*in module pytest*), 156
`ExitCode` (*class in pytest*), 212

extra_keyword_matches (*Node attribute*), 219

F

fail() (*in module pytest*), 154

faulthandler_timeout
configuration value, 234

File (*class in pytest*), 213

fill_unfilled() (*ExceptionInfo method*), 211

filterwarnings
configuration value, 234

finalize() (*Testdir method*), 186

fixture

- cache, 164
- capfd, 166
- capfdbinary, 167
- caplog, 170
- capsys, 165
- capsysbinary, 166
- doctest_namespace, 167
- monkeypatch, 172
- pytestconfig, 169
- pytester, 174
- record_property, 169
- record_testsuite_property, 170
- recwarn, 189
- request, 168
- testdir, 185
- tmp_path, 189
- tmp_path_factory, 190
- tmpdir, 190
- tmpdir_factory, 191

fixture() (*in module pytest*), 164

FixtureDef (*class in _pytest.fixtures*), 213

fixturename (*FixtureRequest attribute*), 168

fixturenames (*FixtureRequest property*), 168

fixturenames (*Metafunc attribute*), 217

FixtureRequest (*class in pytest*), 168

fnmatch_lines() (*LineMatcher method*), 184

fnmatch_lines_random() (*LineMatcher method*),
184

for_later() (*ExceptionInfo class method*), 211

freeze_includes() (*in module pytest*), 160

from_call() (*CallInfo class method*), 205

from_current() (*ExceptionInfo class method*), 210

from_exc_info() (*ExceptionInfo class method*), 210

from_item_and_call() (*TestReport class method*),
227

from_parent() (*Class class method*), 206

from_parent() (*FSCollector class method*), 213

from_parent() (*Function class method*), 214

from_parent() (*Node class method*), 219

fromdictargs() (*Config class method*), 209

FSCollector (*class in _pytest.nodes*), 213

fspath (*FixtureRequest property*), 168

fspath (*Node property*), 219

Function (*class in pytest*), 213

function (*FixtureRequest property*), 168

function (*Function property*), 214

function (*Metafunc attribute*), 217

FunctionDefinition (*class in _pytest.python*), 214

G

genitems() (*Pytester method*), 178

genitems() (*Testdir method*), 187

get() (*Cache method*), 165

get() (*Stash method*), 229

get_canonical_name() (*PytestPluginManager
method*), 224

get_closest_marker() (*Node method*), 220

get_hookcallers() (*PytestPluginManager method*),
224

get_lines_after() (*LineMatcher method*), 184

get_name() (*PytestPluginManager method*), 224

get_plugin() (*PytestPluginManager method*), 224

get_plugins() (*PytestPluginManager method*), 225

get_records() (*LogCaptureFixture method*), 171

getbasetemp() (*TempdirFactory method*), 191

getbasetemp() (*TempPathFactory method*), 190

getfixturevalue() (*FixtureRequest method*), 169

getgroup() (*Parser method*), 221

getini() (*Config method*), 210

getinicfg() (*Pytester method*), 176

getinicfg() (*Testdir method*), 186

getitem() (*Pytester method*), 180

getitem() (*Testdir method*), 188

genitems() (*Pytester method*), 180

genitems() (*Testdir method*), 188

getmodulecol() (*Pytester method*), 180

getmodulecol() (*Testdir method*), 188

getnode() (*Pytester method*), 178

getnode() (*Testdir method*), 187

getoption() (*Config method*), 210

getparent() (*Node method*), 220

getpathnode() (*Pytester method*), 178

getpathnode() (*Testdir method*), 187

getplugin() (*PytestPluginManager method*), 224

getrepr() (*ExceptionInfo method*), 211

getvalue() (*Config method*), 210

getvalueorskip() (*Config method*), 210

global variable interpreted by pytest

- collect_ignore, 230
- collect_ignore_glob, 230
- pytest_plugins, 230
- pytestmark, 230

H

handler (*LogCaptureFixture property*), 171

has_plugin() (*PytestPluginManager method*), 225

hasplugin() (*PytestPluginManager* method), 224
 head_line (*CollectReport* property), 207
 head_line (*TestReport* property), 227
 HookRecorder (*class in _pytest.pytester*), 185

I

ihook (*Node* property), 219
 import_plugin() (*PytestPluginManager* method), 224
 importorskip() (*in module pytest*), 155
 inifile (*Config* property), 209
 inipath (*Config* property), 209
 inline_genitems() (*Pytester* method), 179
 inline_genitems() (*Testdir* method), 187
 inline_run() (*Pytester* method), 179
 inline_run() (*Testdir* method), 187
 inline_runsource() (*Pytester* method), 178
 inline_runsource() (*Testdir* method), 187
 instance (*FixtureRequest* property), 168
 INTERNAL_ERROR (*ExitCode* attribute), 212
 INTERRUPTED (*ExitCode* attribute), 212
 invocation_dir (*Config* property), 209
 invocation_params (*Config* attribute), 208
 is_blocked() (*PytestPluginManager* method), 225
 is_registered() (*PytestPluginManager* method), 225
 issue_config_time_warning() (*Config* method), 209
 Item (*class in pytest*), 214
 iter_markers() (*Node* method), 220
 iter_markers_with_node() (*Node* method), 220

J

junit_duration_report
 configuration value, 234
 junit_family
 configuration value, 234
 junit_log_passing_tests
 configuration value, 235
 junit_logging
 configuration value, 235
 junit_suite_name
 configuration value, 235

K

keywords (*FixtureRequest* property), 168
 keywords (*Node* attribute), 219
 keywords (*TestReport* attribute), 226
 kwargs (*Mark* attribute), 216
 kwargs (*MarkDecorator* property), 215

L

LineMatcher (*class in _pytest.pytester*), 183

list (*WarningsRecorder* property), 189
 list_name_plugin() (*PytestPluginManager* method), 225
 list_plugin_distinfo() (*PytestPluginManager* method), 225
 listchain() (*Node* method), 220
 listextrakeywords() (*Node* method), 220
 load_setuptools_entrypoints() (*PytestPluginManager* method), 225
 location (*TestReport* attribute), 226
 log_auto_indent
 configuration value, 235
 log_cli
 configuration value, 236
 log_cli_date_format
 configuration value, 236
 log_cli_format
 configuration value, 236
 log_cli_level
 configuration value, 236
 log_date_format
 configuration value, 236
 log_file
 configuration value, 236
 log_file_date_format
 configuration value, 236
 log_file_format
 configuration value, 236
 log_file_level
 configuration value, 237
 log_format
 configuration value, 237
 log_level
 configuration value, 237
 LogCaptureFixture (*class in pytest*), 170
 longrepr (*CollectReport* attribute), 207
 longrepr (*TestReport* attribute), 227
 longreprtext (*CollectReport* property), 207
 longreprtext (*TestReport* property), 228

M

main() (*in module pytest*), 156
 make_hook_recorder() (*Pytester* method), 175
 make_hook_recorder() (*Testdir* method), 186
 makeconftest() (*Pytester* method), 176
 makeconftest() (*Testdir* method), 186
 mkdir() (*Cache* method), 165
 makefile() (*Pytester* method), 175
 makefile() (*Testdir* method), 186
 makeini() (*Pytester* method), 176
 makeini() (*Testdir* method), 186
 makepyfile() (*Pytester* method), 176
 makepyfile() (*Testdir* method), 187
 makepyprojecttoml() (*Pytester* method), 176

makepyprojecttoml() (*Testdir method*), 187
 maketxtfile() (*Pytester method*), 176
 maketxtfile() (*Testdir method*), 187
 Mark (*class in pytest*), 216
 MarkDecorator (*class in pytest*), 215
 markers
 configuration value, 237
 MarkGenerator (*class in pytest*), 216
 match() (*ExceptionInfo method*), 212
 matchreport() (*HookRecorder method*), 185
 messages (*LogCaptureFixture property*), 171
 Metafunc (*class in pytest*), 217
 minversion
 configuration value, 237
 mkdir() (*Cache method*), 165
 mkdir() (*Pytester method*), 177
 mkdir() (*Testdir method*), 187
 mkpydir() (*Pytester method*), 177
 mkpydir() (*Testdir method*), 187
 mktemp() (*TempdirFactory method*), 191
 mktemp() (*TempPathFactory method*), 190
 Module (*class in pytest*), 218
 module (*FixtureRequest property*), 168
 module (*Metafunc attribute*), 217
 monkeypatch
 fixture, 172
 MonkeyPatch (*class in pytest*), 172
 monkeypatch() (*in module _pytest.monkeypatch*), 172

N

name (*Mark attribute*), 216
 name (*MarkDecorator property*), 215
 name (*Node attribute*), 219
 no_fnmatch_line() (*LineMatcher method*), 184
 no_re_match_line() (*LineMatcher method*), 184
 NO_TESTS_COLLECTED (*ExitCode attribute*), 212
 Node (*class in _pytest.nodes*), 219
 node (*FixtureRequest property*), 168
 nodeid (*CollectReport attribute*), 207
 nodeid (*Node property*), 219
 nodeid (*TestReport attribute*), 226
 norecursedirs
 configuration value, 237

O

OK (*ExitCode attribute*), 212
 option (*Config attribute*), 208
 OptionGroup (*class in pytest*), 223
 originalname (*Function attribute*), 214
 outcome (*CollectReport attribute*), 207
 outcome (*TestReport attribute*), 226
 outlines (*RunResult attribute*), 182
 own_markers (*Node attribute*), 219

P

param() (*in module pytest*), 156
 parametrize() (*Metafunc method*), 217
 parent (*Node attribute*), 219
 parse_hookimpl_opts() (*PytestPluginManager method*), 223
 parse_hookspec_opts() (*PytestPluginManager method*), 223
 parse_known_and_unknown_args() (*Parser method*), 222
 parse_known_args() (*Parser method*), 222
 parse_summary_nouns() (*RunResult class method*), 183
 parseconfig() (*Pytester method*), 179
 parseconfig() (*Testdir method*), 188
 parseconfigure() (*Pytester method*), 180
 parseconfigure() (*Testdir method*), 188
 parseoutcomes() (*RunResult method*), 183
 Parser (*class in pytest*), 221
 path (*Node attribute*), 219
 path (*Pytester property*), 175
 perform_collect() (*Pytester.Session method*), 177
 perform_collect() (*Session method*), 226
 perform_collect() (*Testdir.Session method*), 186
 pluginmanager (*Config attribute*), 208
 plugins (*Config.InvocationParams attribute*), 208
 pop() (*WarningsRecorder method*), 189
 popen() (*Pytester method*), 180
 popen() (*Testdir method*), 188
 pytest_addhooks() (*in module _pytest.hookspec*), 193
 pytest_adoption() (*in module _pytest.hookspec*), 193
 pytest_assertion_pass() (*in module _pytest.hookspec*), 203
 pytest_assertrepr_compare() (*in module _pytest.hookspec*), 203
 pytest_cmdline_main() (*in module _pytest.hookspec*), 193
 pytest_cmdline_parse() (*in module _pytest.hookspec*), 192
 pytest_cmdline_preparse() (*in module _pytest.hookspec*), 192
 pytest_collect_file() (*in module _pytest.hookspec*), 195
 pytest_collection() (*in module _pytest.hookspec*), 194
 pytest_collection_finish() (*in module _pytest.hookspec*), 197
 pytest_collection_modifyitems() (*in module _pytest.hookspec*), 196
 pytest_collectreport() (*in module _pytest.hookspec*), 200

pytest_collectstart() (in module `_pytest.hookspec`), 200
 pytest_configure() (in module `_pytest.hookspec`), 193
 PYTEST_CURRENT_TEST, 255, 289
 pytest_deselected() (in module `_pytest.hookspec`), 200
 pytest_enter_pdb() (in module `_pytest.hookspec`), 204
 pytest_exception_interact() (in module `_pytest.hookspec`), 204
 pytest_fixture_post_finalizer() (in module `_pytest.hookspec`), 202
 pytest_fixture_setup() (in module `_pytest.hookspec`), 202
 pytest_generate_tests() (in module `_pytest.hookspec`), 196
 pytest_ignore_collect() (in module `_pytest.hookspec`), 195
 pytest_internalerror() (in module `_pytest.hookspec`), 204
 pytest_itemcollected() (in module `_pytest.hookspec`), 200
 pytest_keyboard_interrupt() (in module `_pytest.hookspec`), 204
 pytest_load_initial_conftests() (in module `_pytest.hookspec`), 192
 pytest_make_collect_report() (in module `_pytest.hookspec`), 200
 pytest_make_parametrize_id() (in module `_pytest.hookspec`), 196
 pytest_plugin_registered() (in module `_pytest.hookspec`), 194
 PYTEST_PLUGINS, 89
 pytest_plugins
 global variable interpreted by `pytest`, 230
 pytest_pycollect_makeitem() (in module `_pytest.hookspec`), 196
 pytest_pycollect_makemodule() (in module `_pytest.hookspec`), 196
 pytest_pyfunc_call() (in module `_pytest.hookspec`), 199
 pytest_report_collectionfinish() (in module `_pytest.hookspec`), 201
 pytest_report_header() (in module `_pytest.hookspec`), 200
 pytest_report_teststatus() (in module `_pytest.hookspec`), 201
 pytest_runttest_call() (in module `_pytest.hookspec`), 198
 pytest_runttest_logfinish() (in module `_pytest.hookspec`), 198
 pytest_runttest_logreport() (in module `_pytest.hookspec`), 203
 pytest_runttest_logstart() (in module `_pytest.hookspec`), 198
 pytest_runttest_makereport() (in module `_pytest.hookspec`), 199
 pytest_runttest_protocol() (in module `_pytest.hookspec`), 197
 pytest_runttest_setup() (in module `_pytest.hookspec`), 198
 pytest_runttest_teardown() (in module `_pytest.hookspec`), 199
 pytest_runttestloop() (in module `_pytest.hookspec`), 197
 pytest_sessionfinish() (in module `_pytest.hookspec`), 194
 pytest_sessionstart() (in module `_pytest.hookspec`), 194
 pytest_terminal_summary() (in module `_pytest.hookspec`), 201
 PYTEST_THEME, 149, 231
 PYTEST_THEME_MODE, 149
 pytest_unconfigure() (in module `_pytest.hookspec`), 194
 pytest_warning_captured() (in module `_pytest.hookspec`), 202
 pytest_warning_recorded() (in module `_pytest.hookspec`), 202
 PytestAssertRewriteWarning (class in `pytest`), 231
 PytestCacheWarning (class in `pytest`), 231
 PytestCollectionWarning (class in `pytest`), 232
 pytestconfig
 fixture, 169
 pytestconfig() (in module `_pytest.fixtures`), 169
 PytestConfigWarning (class in `pytest`), 232
 PytestDeprecationWarning (class in `pytest`), 232
 pytester
 fixture, 174
 Pytester (class in `pytest`), 175
 Pytester.Session (class in `pytest`), 177
 Pytester.Session.Failed, 177
 Pytester.Session.Interrupted, 177
 Pytester.TimeoutExpired, 175
 PytestExperimentalApiWarning (class in `pytest`), 232
 pytestmark
 global variable interpreted by `pytest`, 230
 pytest.mark.filterwarnings()
 built-in function, 160
 pytest.mark.skip()
 built-in function, 161
 pytest.mark.skipif()
 built-in function, 161

pytest.mark.usefixtures()
 built-in function, 161
 pytest.mark.xfail()
 built-in function, 162
 PytestPluginManager (class in *pytest*), 223
 PytestUnhandledCoroutineWarning (class in *pytest*), 232
 PytestUnhandledThreadExceptionWarning (class in *pytest*), 232
 PytestUnknownMarkWarning (class in *pytest*), 232
 PytestUnraisableExceptionWarning (class in *pytest*), 232
 PytestWarning (class in *pytest*), 231
 Python Enhancement Proposals
 PEP 302, 91
 PEP 0373#maintenance-releases, 337
 PEP 0429#release-schedule, 337
 PEP 506, 78
 PEP 565, 77
 python:PYTHONWARNINGS, 78
 python_classes
 configuration value, 238
 python_files
 configuration value, 238
 python_functions
 configuration value, 238
 PYTHONPATH, 239
 pythonpath
 configuration value, 239

R

raiseerror() (*FixtureRequest* method), 169
 raises() (in module *pytest*), 157
 re_match_lines() (*LineMatcher* method), 184
 re_match_lines_random() (*LineMatcher* method), 184
 readouterr() (*CaptureFixture* method), 166
 record_property
 fixture, 169
 record_property() (in module *_pytest.junitxml*), 169
 record_testsuite_property
 fixture, 170
 record_testsuite_property() (in module *_pytest.junitxml*), 170
 record_tuples (*LogCaptureFixture* property), 171
 records (*LogCaptureFixture* property), 171
 recwarn
 fixture, 189
 recwarn() (in module *_pytest.recwarn*), 189
 register() (*PytestPluginManager* method), 223
 register_assert_rewrite() (in module *pytest*), 159
 repr_failure() (*Collector* method), 206
 repr_failure() (*Function* method), 214
 repr_failure() (*Node* method), 220
 request
 fixture, 168
 required_plugins
 configuration value, 239
 result (*CallInfo* property), 205
 result (*CollectReport* attribute), 207
 ret (*RunResult* attribute), 182
 rootdir (*Config* property), 209
 rootpath (*Config* property), 209
 run() (*Pytester* method), 181
 run() (*Testdir* method), 188
 runitem() (*Pytester* method), 178
 runitem() (*Testdir* method), 187
 runpytest() (*Pytester* method), 179
 runpytest() (*Testdir* method), 188
 runpytest_inprocess() (*Pytester* method), 179
 runpytest_inprocess() (*Testdir* method), 188
 runpytest_subprocess() (*Pytester* method), 181
 runpytest_subprocess() (*Testdir* method), 188
 runpython() (*Pytester* method), 181
 runpython() (*Testdir* method), 188
 runpython_c() (*Pytester* method), 181
 runpython_c() (*Testdir* method), 188
 RunResult (class in *_pytest.pytester*), 182
 runtest() (*Function* method), 214
 runtest() (*FunctionDefinition* method), 214

S

scope (*FixtureDef* property), 213
 scope (*FixtureRequest* property), 168
 sections (*CollectReport* attribute), 207
 sections (*TestReport* attribute), 227
 Session (class in *pytest*), 225
 session (*FixtureRequest* property), 168
 Session.Failed, 225
 Session.Interrupted, 225
 set() (*Cache* method), 165
 set_blocked() (*PytestPluginManager* method), 225
 set_level() (*LogCaptureFixture* method), 171
 setattr() (*MonkeyPatch* method), 173
 setdefault() (*Stash* method), 229
 setenv() (*MonkeyPatch* method), 173
 setitem() (*MonkeyPatch* method), 173
 setup() (*FunctionDefinition* method), 214
 skip() (in module *pytest*), 155
 spawn() (*Pytester* method), 182
 spawn() (*Testdir* method), 189
 spawn_pytest() (*Pytester* method), 182
 spawn_pytest() (*Testdir* method), 188
 stardir (*Pytester.Session* property), 178
 stardir (*Session* property), 225
 stardir (*Testdir.Session* property), 186
 start (*CallInfo* attribute), 205

startpath (*Pytester.Session* property), 178
startpath (*Session* property), 225
startpath (*Testdir.Session* property), 186
Stash (class in *pytest*), 228
stash (*Config* attribute), 208
stash (*Node* attribute), 219
StashKey (class in *pytest*), 229
stderr (*RunResult* attribute), 182
stdout (*RunResult* attribute), 182
stop (*CallInfo* attribute), 205
str () (*LineMatcher* method), 185
subset_hook_caller () (*PytestPluginManager* method), 225
syspath_prepend () (*MonkeyPatch* method), 174
syspathinsert () (*Pytester* method), 177
syspathinsert () (*Testdir* method), 187

T

tb (*ExceptionInfo* property), 211
TempdirFactory (class in *pytest*), 191
TempPathFactory (class in *pytest*), 190
testdir
 fixture, 185
Testdir (class in *pytest*), 185
Testdir.Session (class in *pytest*), 185
Testdir.Session.Failed, 185
Testdir.Session.Interrupted, 185
Testdir.TimeoutExpired, 185
testpaths
 configuration value, 239
TestReport (class in *_pytest.reports*), 226
TESTS_FAILED (*ExitCode* attribute), 212
text (*LogCaptureFixture* property), 171
tmp_path
 fixture, 189
tmp_path () (in module *_pytest.tmpdir*), 190
tmp_path_factory
 fixture, 190
tmpdir
 fixture, 190
tmpdir (*Testdir* property), 186
tmpdir () (in module *_pytest.tmpdir*), 191
tmpdir_factory
 fixture, 191
traceback (*ExceptionInfo* property), 211
type (*ExceptionInfo* property), 211
typename (*ExceptionInfo* property), 211

U

undo () (*MonkeyPatch* method), 174
unregister () (*PytestPluginManager* method), 225
USAGE_ERROR (*ExitCode* attribute), 212
UsageError (class in *pytest*), 231
usefixtures

 configuration value, 239
user_properties (*Item* attribute), 214
user_properties (*TestReport* attribute), 227

V

value (*ExceptionInfo* property), 211

W

warn () (*Node* method), 219
WarningsRecorder (class in *pytest*), 189
warns () (in module *pytest*), 159
when (*CallInfo* attribute), 205
when (*TestReport* attribute), 227
with_args () (*MarkDecorator* method), 216

X

xfail () (in module *pytest*), 155
xfail_strict
 configuration value, 239