pytest Documentation

Release 9.1

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

Apr 21, 2026

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 . . . 21
- 2.4 How to mark test functions with attributes . . . 52
- 2.5 How to parametrize fixtures and test functions . . . 54
- 2.6 How to use temporary directories and files in tests . . . 58
- 2.7 How to monkeypatch/mock modules and environments . . . 61
- 2.8 How to run doctests . . . 68
- 2.9 How to re-run failed tests and maintain state between test runs . . . 72
- 2.10 How to manage logging . . . 78
- 2.11 How to capture stdout/stderr output . . . 82
- 2.12 How to capture warnings . . . 84
- 2.13 How to use skip and xfail to deal with tests that cannot succeed . . . 92
- 2.14 How to install and use plugins . . . 99
- 2.15 Writing plugins . . . 101
- 2.16 Writing hook functions . . . 108
- 2.17 How to use pytest with an existing test suite . . . 113
- 2.18 How to use `unittest`-based tests with pytest . . . 113
- 2.19 How to implement xunit-style set-up . . . 117
- 2.20 How to set up bash completion . . . 119

3 Reference guides **121**
- 3.1 Fixtures reference . . . 121
- 3.2 Pytest Plugin List . . . 140
- 3.3 Configuration . . . 286
- 3.4 API Reference . . . 290

4 Explanation **423**
- 4.1 Anatomy of a test . . . 423
- 4.2 About fixtures . . . 423
- 4.3 Good Integration Practices . . . 426
- 4.4 Flaky tests . . . 431
- 4.5 pytest import mechanisms and `sys.path`/`PYTHONPATH` . . . 434

5 Further topics **439**
- 5.1 Examples and customization tricks . . . 439

5.2 Backwards Compatibility Policy . . . 507
5.3 History . . . 508
5.4 Python version support . . . 509
5.5 Deprecations and Removals . . . 509
5.6 Contributing . . . 530
5.7 Development Guide . . . 539
5.8 Sponsor . . . 539
5.9 pytest for enterprise . . . 540
5.10 License . . . 541
5.11 Contact channels . . . 541
5.12 History . . . 542
5.13 Historical Notes . . . 544
5.14 Talks and Tutorials . . . 548

Download latest version as PDF

CHAPTER

ONE

START HERE

1.1 Get Started

1.1.1 Install `pytest`

1. Run the following command in your command line:

```
pip install -U pytest
```

2. Check that you installed the correct version:

```
$ pytest --version
pytest 9.0.3
```

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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_sample.py F                                                  [100%]

================================ FAILURES ================================
_____________________________ test_answer _____________________________

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

(continues on next page)

(continued from previous page)

```
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
.                                                                    [100%]
1 passed in 0.12s
```

Note

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

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

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):
```

(continues on next page)

(continued from previous page)

```
        self.value = 1
        assert self.value == 1

    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 Compare floating-point values with pytest.approx

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

```
# content of test_approx.py
import pytest


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

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

1.1.7 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.

1.1.8 Continue reading

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

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

CHAPTER

TWO

HOW-TO GUIDES

2.1 How to invoke pytest

See also

Complete pytest command-line flags 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 or from a file (see below for *reading arguments from file*).

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`. Use `""` instead of `''` in expression when running this on Windows

Run tests by collection arguments

Pass the module filename relative to the working directory, followed by specifiers like the class name and function name separated by `::` characters, and parameters from parameterization enclosed in `[]`.

To run a specific test within a module:

```
pytest tests/test_mod.py::test_func
```

To run all tests in a class:

```
pytest tests/test_mod.py::TestClass
```

Specifying a specific test method:

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

Specifying a specific parametrization of a test:

```
pytest tests/test_mod.py::test_func[x1,y2]
```

Run tests by marker expressions

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

```
pytest -m slow
```

To run all tests which are decorated with the annotated `@pytest.mark.slow(phase=1)` decorator, with the `phase` keyword argument set to `1`:

```
pytest -m "slow(phase=1)"
```

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.

Read arguments from file

Added in version 8.2.

All of the above can be read from a file using the `@` prefix:

```
pytest @tests_to_run.txt
```

where `tests_to_run.txt` contains an entry per line, e.g.:

```
tests/test_file.py
tests/test_mod.py::test_func[x1,y2]
tests/test_mod.py::TestClass
-m slow
```

This file can also be generated using `pytest --collect-only -q` and modified as needed.

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 `importlib` 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. If you don't pass it any arguments, `main` reads the arguments from the command line arguments of the process (`sys.argv`), which may be undesirable. You can pass in options and arguments explicitly:

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

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

```
# content of myinvoke.py
import sys

import pytest


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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
```

(continues on next page)

(continued from previous page)

```
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.

If a message is specified with the assertion like this:

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

it is printed alongside the assertion introspection in the traceback.

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

2.2.2 Assertions about approximate equality

When comparing floating point values (or arrays of floats), small rounding errors are common. Instead of using `assert abs(a - b) < tol` or `numpy.isclose`, you can use *`pytest.approx()`*:

```
import pytest
import numpy as np


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


def test_arrays():
    a = np.array([1.0, 2.0, 3.0])
    b = np.array([0.9999, 2.0001, 3.0])
    assert a == pytest.approx(b)
```

`pytest.approx` works with scalars, lists, dictionaries, and NumPy arrays. It also supports comparisons involving NaNs.

See *`pytest.approx()`* for details.

2.2.3 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`.

Note that `pytest.raises` will match the exception type or any subclasses (like the standard `except` statement). If you want to check if a block of code is raising an exact exception type, you need to check that explicitly:

```
def test_foo_not_implemented():
    def foo():
        raise NotImplementedError

    with pytest.raises(RuntimeError) as excinfo:
        foo()
    assert excinfo.type is RuntimeError
```

The *pytest.raises()* call will succeed, even though the function raises `NotImplementedError`, because `NotImplementedError` is a subclass of `RuntimeError`; however the following `assert` statement will catch the problem.

Matching exception messages

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()
```

Notes:

- The `match` parameter is matched with the `re.search()` function, so in the above example `match='123'` would have worked as well.
- The `match` parameter also matches against PEP-678 `__notes__`.

Assertions about expected exception groups

When expecting a `BaseExceptionGroup` or `ExceptionGroup` you can use *pytest.RaisesGroup*:

```
def test_exception_in_group():
    with pytest.RaisesGroup(ValueError):
        raise ExceptionGroup("group msg", [ValueError("value msg")])
    with pytest.RaisesGroup(ValueError, TypeError):
        raise ExceptionGroup("msg", [ValueError("foo"), TypeError("bar")])
```

It accepts a `match` parameter, that checks against the group message, and a `check` parameter that takes an arbitrary callable which it passes the group to, and only succeeds if the callable returns `True`.

```
def test_raisesgroup_match_and_check():
    with pytest.RaisesGroup(BaseException, match="my group msg"):
        raise BaseExceptionGroup("my group msg", [KeyboardInterrupt()])
    with pytest.RaisesGroup(
        Exception, check=lambda eg: isinstance(eg.__cause__, ValueError)
    ):
        raise ExceptionGroup("", [TypeError()]) from ValueError()
```

It is strict about structure and unwrapped exceptions, unlike except*, so you might want to set the `flatten_subgroups` and/or `allow_unwrapped` parameters.

```
def test_structure():
    with pytest.RaisesGroup(pytest.RaisesGroup(ValueError)):
        raise ExceptionGroup("", (ExceptionGroup("", (ValueError(),)),))
    with pytest.RaisesGroup(ValueError, flatten_subgroups=True):
        raise ExceptionGroup("1st group", [ExceptionGroup("2nd group",␣
↪[ValueError()])])
    with pytest.RaisesGroup(ValueError, allow_unwrapped=True):
        raise ValueError
```

To specify more details about the contained exception you can use *pytest.RaisesExc*

```
def test_raises_exc():
    with pytest.RaisesGroup(pytest.RaisesExc(ValueError, match="foo")):
        raise ExceptionGroup("", (ValueError("foo")))
```

They both supply a method *pytest.RaisesGroup.matches()* *pytest.RaisesExc.matches()* if you want to do matching outside of using it as a context manager. This can be helpful when checking `.__context__` or `.__cause__`.

```
def test_matches():
    exc = ValueError()
    exc_group = ExceptionGroup("", [exc])
    if RaisesGroup(ValueError).matches(exc_group):
        ...
    # helpful error is available in `.fail_reason` if it fails to match
    r = RaisesExc(ValueError)
    assert r.matches(e), r.fail_reason
```

Check the documentation on *pytest.RaisesGroup* and *pytest.RaisesExc* for more details and examples.

ExceptionInfo.group_contains()

Warning

This helper makes it easy to check for the presence of specific exceptions, but it is very bad for checking that the group does *not* contain *any other exceptions*. So this will pass:

```
class EXTREMELYBADERROR(BaseException):
    """This is a very bad error to miss"""


def test_for_value_error():
    with pytest.raises(ExceptionGroup) as excinfo:
        excs = [ValueError()]
        if very_unlucky():
            excs.append(EXTREMELYBADERROR())
        raise ExceptionGroup("", excs)
    # This passes regardless of if there's other exceptions.
    assert excinfo.group_contains(ValueError)
    # You can't simply list all exceptions you *don't* want to get here.
```

There is no good way of using *excinfo.group_contains()* to ensure you're not getting *any* other exceptions than the one you expected. You should instead use *pytest.RaisesGroup*, see *Assertions about expected exception groups*.

You can also use the *excinfo.group_contains()* method to test for exceptions returned as part of an ExceptionGroup:

```
def test_exception_in_group():
    with pytest.raises(ExceptionGroup) as excinfo:
        raise ExceptionGroup(
            "Group message",
            [
                RuntimeError("Exception 123 raised"),
            ],
        )
    assert excinfo.group_contains(RuntimeError, match=r".* 123 .*")
    assert not excinfo.group_contains(TypeError)
```

The optional `match` keyword parameter works the same way as for *pytest.raises()*.

By default `group_contains()` will recursively search for a matching exception at any level of nested `ExceptionGroup` instances. You can specify a `depth` keyword parameter if you only want to match an exception at a specific level; exceptions contained directly in the top `ExceptionGroup` would match `depth=1`.

```
def test_exception_in_group_at_given_depth():
    with pytest.raises(ExceptionGroup) as excinfo:
        raise ExceptionGroup(
            "Group message",
            [
                RuntimeError(),
                ExceptionGroup(
```

(continues on next page)

(continued from previous page)

```
                    "Nested group",
                    [
                        TypeError(),
                    ],
                ),
            ],
        )
    assert excinfo.group_contains(RuntimeError, depth=1)
    assert excinfo.group_contains(TypeError, depth=2)
    assert not excinfo.group_contains(RuntimeError, depth=2)
    assert not excinfo.group_contains(TypeError, depth=1)
```

Alternate `pytest.raises` form (legacy)

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

```
def func(x):
    if x <= 0:
        raise ValueError("x needs to be larger than zero")


pytest.raises(ValueError, func, x=-1)
```

This form was the original *`pytest.raises()`* API, developed before the `with` statement was added to the Python language. Nowadays, this form is rarely used, with the context-manager form (using `with`) being considered more readable.

xfail mark and pytest.raises

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:

```
def f():
    raise IndexError()


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

This will only "xfail" if the test fails by raising `IndexError` or subclasses.

- Using *pytest.mark.xfail* with the `raises` parameter is probably better for something like documenting unfixed bugs (where the test describes what "should" happen) or bugs in dependencies.
- Using *`pytest.raises()`* is likely to be better for cases where you are testing exceptions your own code is deliberately raising, which is the majority of cases.

You can also use *`pytest.RaisesGroup`*:

```
def f():
    raise ExceptionGroup("", [IndexError()])
```

(continues on next page)

(continued from previous page)

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

2.2.4 Assertions about expected warnings

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

2.2.5 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-9.x.y, pluggy-1.x.y
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
E         Extra items in the left set:
E         '1'
E         Extra items in the right set:
E         '5'
E         Use -v to get more 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

In string context diffs, lines prefixed with - come from the left-hand side of `assert left == right`, while lines prefixed with + come from the right-hand side.

See the *reporting demo* for many more examples.

2.2.6 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** – The pytest config object.
- **op** – The operator, e.g. `"=="`, `"!="`, `"not in"`.
- **left** – The left operand.
- **right** – The right operand.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

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:",
            f"   vals: {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
```

(continues on next page)

(continued from previous page)

```
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():
        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
```

2.2.7 Returning non-None value in test functions

A *pytest.PytestReturnNotNoneWarning* is emitted when a test function returns a value other than `None`.

This helps prevent a common mistake made by beginners who assume that returning a `bool` (e.g., `True` or `False`) will determine whether a test passes or fails.

Example:

```
@pytest.mark.parametrize(
    ["a", "b", "result"],
    [
        [1, 2, 5],
        [2, 3, 8],
        [5, 3, 18],
    ],
)
def test_foo(a, b, result):
    return foo(a, b) == result  # Incorrect usage, do not do this.
```

Since pytest ignores return values, it might be surprising that the test will never fail based on the returned value.

The correct fix is to replace the `return` statement with an `assert`:

```
@pytest.mark.parametrize(
    ["a", "b", "result"],
    [
        [1, 2, 5],
        [2, 3, 8],
```

(continues on next page)

(continued from previous page)

```
        [5, 3, 18],
    ],
)
def test_foo(a, b, result):
    assert foo(a, b) == result
```

2.2.8 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, e.g. 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"
```

(continues on next page)

(continued from previous page)

```
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
```

(continues on next page)

(continued from previous page)

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


# 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
```

(continues on next page)

(continued from previous page)

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


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 smtplib

import pytest


@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-9.x.y, pluggy-1.x.y
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):
```

(continues on next page)

(continued from previous page)

```
        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_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 **exact 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 where the fixture is defined, including sub-packages and sub-directories within it.
- `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

Added 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):
```

(continues on next page)

(continued from previous page)

```
        # do some cleanup
        pass


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
from emaillib import Email, MailAdminClient

import pytest


@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
    user.clear_mailbox()
    mail_admin.delete_user(user)
```

(continues on next page)

(continued from previous page)

```
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
.                                                                    [100%]
1 passed in 0.12s
```

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
from emaillib import Email, MailAdminClient

import pytest


@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)
```

(continues on next page)

(continued from previous page)

```
@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


@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
```

Note on finalizer order

Finalizers are executed in a first-in-last-out order. For yield fixtures, the first teardown code to run is from the right-most fixture, i.e. the last test parameter.

```
# content of test_finalizers.py
import pytest


def test_bar(fix_w_yield1, fix_w_yield2):
    print("test_bar")


@pytest.fixture
def fix_w_yield1():
    yield
    print("after_yield_1")


@pytest.fixture
```

(continues on next page)

(continued from previous page)

```
def fix_w_yield2():
    yield
    print("after_yield_2")
```

```
$ pytest -s test_finalizers.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_finalizers.py test_bar
.after_yield_2
after_yield_1


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

For finalizers, the first fixture to run is last call to `request.addfinalizer`.

```
# content of test_finalizers.py
from functools import partial
import pytest


@pytest.fixture
def fix_w_finalizers(request):
    request.addfinalizer(partial(print, "finalizer_2"))
    request.addfinalizer(partial(print, "finalizer_1"))


def test_bar(fix_w_finalizers):
    print("test_bar")
```

```
$ pytest -s test_finalizers.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_finalizers.py test_bar
.finalizer_1
finalizer_2


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

This is so because yield fixtures use `addfinalizer` behind the scenes: when the fixture executes, `addfinalizer` registers a function that resumes the generator, which in turn calls the teardown code.

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
from emaillib import Email, MailAdminClient

import pytest


@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)


@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)
```

(continues on next page)

(continued from previous page)

```
    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)
```

(continues on next page)

(continued from previous page)

```
@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:
```

(continues on next page)

(continued from previous page)

```
    @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 smtplib

import pytest


@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(f"finalizing {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


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 <smtplib.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": []}
```

(continues on next page)

(continued from previous page)

```
    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 smtplib

import pytest


@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(f"finalizing {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
FFFF                                                                  [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 51200000\nETRN\nSTARTTLS\nAUTH DIGEST-MD5 NTLM CRAM-MD5\nENHANCEDSTATUSCODES\
↪n8BITMIME\nDSN\nSMTPUTF8\nCHUNKING'

test_module.py:6: AssertionError
-------------------------- Captured stdout setup ---------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef0004>
_____________________________ test_noop[mail.python.org] _____________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0005>

    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
```

(continues on next page)

(continued from previous page)

```
        assert response == 250
>       assert 0  # for demo purposes
        ^^^^^^^^
E       assert 0

test_module.py:13: AssertionError
------------------------- Captured stdout teardown -------------------------
finalizing <smtplib.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:

```python
# 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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 12 items

<Dir fixtures.rst-234>
  <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_finalizers.py>
    <Function test_bar>
  <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]>

======================== 12 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):
```

(continues on next page)

(continued from previous page)

```
    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-9.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 reuse 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-9.x.y, pluggy-1.x.y -- $PYTHON_PREFIX/bin/
↪python
cachedir: .pytest_cache
rootdir: /home/sweet/project
```

(continues on next page)

(continued from previous page)

```
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 respective 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:

```python
# 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(f"  RUN test2 with otherarg {otherarg} and modarg {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-9.x.y, pluggy-1.x.y -- $PYTHON_PREFIX/bin/
↪python
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 8 items

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 led 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 torn 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
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", encoding="utf-8") 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 a configuration file:

```
# content of pytest.toml
[pytest]
usefixtures = ["cleandir"]
```

Warning

`@pytest.mark.usefixtures` cannot be used on **fixture functions**. For example, this is an error:

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

2.3.15 Overriding fixtures on various levels

In a relatively large test suite, you may want to *override* a fixture, to augment or change its behavior inside of certain test modules or directories.

Override a fixture on a directory (conftest) level

Given the tests file structure is:

```
tests/
    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'

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

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

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

As you can see, a fixture with the same name can be overridden for a certain test directory 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/
    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 a certain test module.

Override a fixture with direct test parametrization

Given the tests file structure is:

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

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

        @pytest.fixture
```

(continues on next page)

(continued from previous page)

```
        def other_username(username):
            return 'other-' + username

    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/
    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']
```

(continues on next page)

(continued from previous page)

```
test_something_else.py
    # content of tests/test_something_else.py
    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 directory 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 configuration file like this:

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

```
[pytest]
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.addinivalue_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 configuration file or using a custom `pytest_configure` hook.

When the `strict_markers` configuration option is set, 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 setting `strict_markers` in your configuration:

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

```
[pytest]
strict_markers = true
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.

Note

See subtests for an alternative to parametrization.

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-9.x.y, pluggy-1.x.y
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",␣
↪42)])
    def test_eval(test_input, expected):
```

(continues on next page)

(continued from previous page)

```
>       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 configuration file:

```
[pytest]
disable_test_id_escaping_and_forfeit_all_rights_to_community_support = true
```

```
[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
```

(continues on next page)

(continued from previous page)

```
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-9.x.y, pluggy-1.x.y
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=[],
        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"))
```

Note

The *pytest_generate_tests* hook can also be implemented directly in a test module or inside a test class; unlike other hooks, pytest will discover it there as well. Other hooks must live in a *conftest.py* or a plugin. See *Writing hook functions*.

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[!] _____________________________
```

(continues on next page)

(continued from previous page)

```
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 for (stringinput)
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 each test function.

`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, encoding="utf-8")
    assert p.read_text(encoding="utf-8") == 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-9.x.y, pluggy-1.x.y
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, encoding="utf-8")
        assert p.read_text(encoding="utf-8") == 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 ==========================
```

By default, `pytest` retains the temporary directory for the last 3 `pytest` invocations. Concurrent invocations of the same test function are supported by configuring the base temporary directory to be unique for each concurrent run. See *temporary directory location and retention* for details.

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
```

(continues on next page)

(continued from previous page)

```
# 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.

Note

These days, it is preferred to use `tmp_path` and `tmp_path_factory`.

In order to help modernize old code bases, one can run pytest with the legacypath plugin disabled:

```
pytest -p no:legacypath
```

This will trigger errors on tests using the legacy paths. It can also be permanently set as part of the *addopts* parameter in the config file.

See *tmpdir tmpdir_factory* API for details.

2.6.4 Temporary directory location and retention

The temporary directories, as returned by the *tmp_path* and (now deprecated) *tmpdir* fixtures, are automatically created under a base temporary directory, in a structure that depends on the *--basetemp* option:

- By default (when the *--basetemp* option is not set), the temporary directories will follow this template:

  ```
  {temproot}/pytest-of-{user}/pytest-{num}/{testname}/
  ```

  where:

  - `{temproot}` is the system temporary directory as determined by `tempfile.gettempdir()`. It can be overridden by the *PYTEST_DEBUG_TEMPROOT* environment variable.
  - `{user}` is the user name running the tests,
  - `{num}` is a number that is incremented with each test suite run
  - `{testname}` is a sanitized version of *the name of the current test*.

  The auto-incrementing `{num}` placeholder provides a basic retention feature and avoids that existing results of previous test runs are blindly removed. By default, the last 3 temporary directories are kept, but this behavior can be configured with *tmp_path_retention_count* and *tmp_path_retention_policy*.

- When the *--basetemp* option is used (e.g. `pytest --basetemp=mydir`), it will be used directly as base temporary directory:

  ```
  {basetemp}/{testname}/
  ```

  Note that there is no retention feature in this case: only the results of the most recent run will be kept.

> **Warning**
>
> The directory given to *--basetemp* will be cleared blindly before each test run, 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 temporary 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.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)*
- *monkeypatch.context()*

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()`.

6. Use *monkeypatch.context* to apply patches only in a specific scope, which can help control teardown of complex fixtures or patches to the stdlib.

See the monkeypatch blog post for some introduction material and a discussion of its motivation.

2.7.1 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
```

(continues on next page)

(continued from previous page)

```
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
    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"}
```

(continues on next page)

(continued from previous page)

```
# 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 #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")

    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):
```

(continues on next page)

(continued from previous page)

```
    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
```

(continues on next page)

(continued from previous page)

```
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-9.x.y, pluggy-1.x.y
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-9.x.y, pluggy-1.x.y
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 configuration file like this:

```
# content of pytest.toml
[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* configuration option:

```
[pytest]
doctest_encoding = "latin1"
```

```
[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"]
```

```
[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. The numbers are compared using *`pytest.approx()`* with relative tolerance equal to the precision. For example, the following output would only need to match to 2 decimal places when comparing `3.14` to `pytest.approx(math.pi, rel=10**-2)`:

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

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

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

```
Expected:
    0.233
Got:
    0.23300000000000001
```

  `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.

  Added 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 the standard doctest module's format options (see `doctest.REPORT_UDIFF`, `doctest.REPORT_CDIFF`, `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')
```

(continues on next page)

(continued from previous page)

```
>>> ...
>>>
```

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 pytest
import numpy


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

Note that like the normal `conftest.py`, the fixtures are discovered in the directory tree conftest is in. Meaning that if you put your doctest with your source code, the relevant conftest.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 invocations 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):
>           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-9.x.y, pluggy-1.x.y
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
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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 50 items
run-last-failure: rerun previous 2 failures first

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

================================= FAILURES =================================
_____________________________ test_num[17] _____________________________
```

(continues on next page)

(continued from previous page)

```
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* option: 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

The *--lfnf, --last-failed-no-failures* option governs the behavior of *--last-failed*. Determines whether to execute tests when there are no previously (known) failures or when no cached `lastfailed` data was found.

There are two options:

- `all`: when there are no known test failures, runs all tests (the full test suite). This is the default.
- `none`: when there are no known test failures, just emits a message stating this and exit successfully.

Example:

```
pytest --last-failed --last-failed-no-failures all    # runs the full test suite␣
↪(default behavior)
pytest --last-failed --last-failed-no-failures none   # runs no tests and exits␣
↪successfully
```

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 reuses previously created state across pytest invocations:

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

(continues on next page)

(continued from previous page)

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


@pytest.fixture
def mydata(pytestconfig):
    val = pytestconfig.cache.get("example/value", None)
    if val is None:
        expensive_computation()
        val = 42
        pytestconfig.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:19: 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
```

(continues on next page)

(continued from previous page)

```
test_caching.py:19: 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-9.x.y, pluggy-1.x.y
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']
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-9.x.y, pluggy-1.x.y
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 stdlog 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 stdlog 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 a configuration file:

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

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

Specific loggers can be disabled via `--log-disable={logger_name}`. This argument can be passed multiple times:

```
pytest --log-disable=main --log-disable=testing
```

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)
```

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")
```

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(f"warning messages encountered during testing: {messages}")
```

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

Warning

The `caplog` fixture adds a handler to the root logger to capture logs. If the root logger is modified during a test, for example with `logging.config.dictConfig`, this handler may be removed and cause no logs to be captured. To avoid this, ensure that any root logger configuration only adds to the existing handlers.

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 or numeric values as seen in logging's documentation.

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 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 by default, which means that it will be overwritten at each test session. If you'd like the file opened in append mode instead, then you can pass *--log-file-mode=a*. Note that relative paths for the

log-file location, whether passed on the CLI or declared in a config file, are always resolved relative to the current working directory.

You can also specify the logging level for the log file by passing *--log-file-level*. This setting accepts the logging level names or numeric values as seen in logging's documentation.

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 file. The option names are:

- *log_file*
- *log_file_mode*
- *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**. Note that `set_log_path()` respects the *log_file_mode* option.

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(trylast=True)
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 configuration file:

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

```
[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 these options to your configuration file:

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

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

More details about the discussion that led to these changes can be read in #3013.

2.11 How to capture stdout/stderr output

Pytest intercepts stdout and stderr as configured by the *`--capture=`* command-line argument or by using fixtures. The `--capture=` flag configures reporting, whereas the fixtures offer more granular control and allow inspection of output during testing. The reports can be customized with the *`-r`* flag.

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 capturing 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)


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-9.x.y, pluggy-1.x.y
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*, *capteesys*, *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.

The return value of `readouterr()` is a `namedtuple` with two attributes, `out` and `err`.

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

If you want to capture at the file descriptor 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). Similarly to *capsysbinary*, *capfdbinary* can be used to capture `bytes` at the file descriptor level.

To temporarily disable capture within a test, the capture fixtures 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")
```

Note

When a capture fixture such as *capsys* or *capfd* is used, it takes precedence over the global capturing configuration set via command-line options such as `-s` or `--capture=no`.

This means that output produced within a test using a capture fixture will still be captured and available via `readouterr()`, even if global capturing is disabled.

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
```

(continues on next page)

(continued from previous page)

```
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-9.x.y, pluggy-1.x.y
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 =========================
```

2.12.1 Controlling warnings

Similar to Python's warning filter and `-W option` flag, pytest provides its own `-W` flag to control which warnings are ignored, displayed, or turned into errors. See the warning filter documentation for more advanced use-cases.

This code sample shows how to treat any `UserWarning` category class of warning as an error:

```
$ 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
```

(continues on next page)

(continued from previous page)

```
======================== 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 configuration file using the *filterwarnings* configuration 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]
filterwarnings = [
    'error',
    'ignore::UserWarning',
    # Note the use of single quote below to denote "raw" strings in TOML.
    'ignore:function ham\(\) is deprecated:DeprecationWarning',
]
```

```
[pytest]
filterwarnings =
    error
    ignore::UserWarning
    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.

Note

The `-W` flag and the *filterwarnings* configuration option use warning filters that are similar in structure, but each configuration option interprets its filter differently. For example, *message* in `filterwarnings` is a string containing a regular expression that the start of the warning message must match, case-insensitively, while *message* in `-W` is a literal string that the start of the warning message must contain (case-insensitively), ignoring any whitespace at the start or end of message. Consult the warning filter documentation for more details.

2.12.2 `@pytest.mark.filterwarnings`

You can use the *@pytest.mark.filterwarnings* mark 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
```

You can specify multiple filters with separate decorators:

```
# Ignore "api v1" warnings, but fail on all other warnings
@pytest.mark.filterwarnings("ignore:api v1")
@pytest.mark.filterwarnings("error")
def test_one():
    assert api_v1() == 1
```

You can also pass multiple filters to a single mark by providing multiple arguments:

```
# Later arguments take precedence, matching warnings.filterwarnings behavior.
@pytest.mark.filterwarnings("error", "ignore:api v1")
def test_one():
    assert api_v1() == 1
```

Important

Regarding decorator order and filter precedence: it's important to remember that decorators are evaluated in reverse order, so you have to list the warning filters in the reverse order compared to traditional `warnings.filterwarnings()` and `-W option` usage. This means in practice that filters from earlier *@pytest.mark.filterwarnings* decorators take precedence over filters from later decorators, as illustrated in the example above.

Filters applied using a mark take precedence over filters passed on the command line or configured by the *filterwarnings* configuration 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")
```

Note

If you want to apply multiple filters (by assigning a list of *filterwarnings* mark to *pytestmark*), you must use the traditional `warnings.filterwarnings()` ordering approach (later filters take precedence), which is the reverse of the decorator approach mentioned above.

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

2.12.3 Setting a maximum number of warnings

Added in version 9.1.

You can use the *--max-warnings* command-line option to fail the test run if the total number of warnings exceeds a given threshold:

```
pytest --max-warnings=10
```

If all tests pass but the number of warnings exceeds the threshold, pytest will exit with code `6` (*ExitCode* `MAX_WARNINGS_ERROR`). This is useful for gradually ratcheting down warnings in a codebase.

Note that *filtered warnings* do not count toward this maximum total.

The threshold can also be set in the configuration file using *max_warnings*:

```
[pytest]
max_warnings = 10
```

```
[pytest]
max_warnings = 10
```

Note

If tests fail, the exit code will be `1` (*`ExitCode`* `TESTS_FAILED`) regardless of the warning count. `MAX_WARNINGS_ERROR` is only reported when all tests pass but the warning threshold is exceeded.

2.12.4 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.

2.12.5 Disabling warning capture entirely

This plugin is enabled by default but can be disabled entirely in your configuration file with:

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

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

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

2.12.6 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.

However, in the specific case where users capture any type of warnings in their test, either with *`pytest.warns()`*, *`pytest.deprecated_call()`* or using the *`recwarn`* fixture, no warning will be displayed at all.

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 (configuration or marks) to ignore those warnings.

For example:

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

```
[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"`.

See *@pytest.mark.filterwarnings* and *Controlling warnings* for more examples.

> **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 565** suggestion of resetting all warning filters because it might break test suites that configure warning filters themselves by calling `warnings.simplefilter()` (see #2430 for an example of that).

2.12.7 Ensuring code triggers a deprecation warning

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

```
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.

2.12.8 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* (except that *raises* does not capture all exceptions, only the `expected_exception`):

```
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. Use the keyword argument `match` to assert that the warning matches a text or regex. To match a literal string that may contain regular expression metacharacters like `(` or `.`, the pattern can first be escaped with `re.escape`.

Some examples:

```
>>> 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)
...
```

(continues on next page)

(continued from previous page)

```
>>> with warns(UserWarning, match=r"must be \d+$"):
...     warnings.warn("this is not here", UserWarning)
...
Traceback (most recent call last):
  ...
Failed: Regex pattern did not match any of the 1 warnings emitted.
 Regex: ...
 Emitted warnings: ...UserWarning...

>>> with warns(UserWarning, match=re.escape("issue with foo() func")):
...     warnings.warn("issue with foo() func")
...
```

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.

2.12.9 Recording warnings

You can record raised warnings either using the *pytest.warns()* context manager 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 issubclass(w.category, UserWarning)
```

(continues on next page)

(continued from previous page)

```
    assert str(w.message) == "hello"
    assert w.filename
    assert w.lineno
```

Both the *recwarn* fixture and the *pytest.warns()* context manager 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.

2.12.10 Additional use cases of warnings in tests

Here are some use cases involving warnings that often come up in tests, and suggestions on how to deal with them:

- To ensure that **at least one** of the indicated warnings is issued, use:

```
def test_warning():
    with pytest.warns((RuntimeWarning, UserWarning)):
        ...
```

- To ensure that **only** certain warnings are issued, use:

```
def test_warning(recwarn):
    ...
    assert len(recwarn) == 1
    user_warning = recwarn.pop(UserWarning)
    assert issubclass(user_warning.category, UserWarning)
```

- To ensure that **no** warnings are emitted, use:

```
def test_warning():
    with warnings.catch_warnings():
        warnings.simplefilter("error")
        ...
```

- To suppress warnings, use:

```
with warnings.catch_warnings():
    warnings.simplefilter("ignore")
    ...
```

2.12.11 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.

2.12.12 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*.

2.12.13 Resource Warnings

Additional information of the source of a `ResourceWarning` can be obtained when captured by pytest if `tracemalloc` module is enabled.

One convenient way to enable `tracemalloc` when running tests is to set the `PYTHONTRACEMALLOC` to a large enough number of frames (say `20`, but that number is application dependent).

For more information, consult the Python Development Mode section in the Python 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 Python3.13:

```
import sys


@pytest.mark.skipif(sys.version_info < (3, 13), reason="requires python3.13 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 particularly 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 `strict_xfail` ini option:

```
[pytest]
xfail_strict = true
```

```
[pytest]
strict_xfail = 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:

```
from __future__ import annotations

import pytest


xfail = pytest.mark.xfail


@xfail
def test_hello():
    assert 0


@xfail(run=False)
def test_hello2():
    assert 0


@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 xxxxxxx                                                  [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.
- allure-pytest: report test results via allure-framework.

To see a complete list of all plugins with their latest testing status against different pytest and Python versions, please visit *Pytest 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 configuration file:

```
[pytest]
addopts = ["-p", "no:NAME"]
```

```
[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.

2.14.4 Disabling plugins from autoloading

If you want to disable plugins from loading automatically, instead of requiring you to manually specify each plugin with *-p* or *PYTEST_PLUGINS*, you can use *--disable-plugin-autoload* or *PYTEST_DISABLE_PLUGIN_AUTOLOAD*.

```
export PYTEST_DISABLE_PLUGIN_AUTOLOAD=1
export PYTEST_PLUGINS=NAME,NAME2
pytest
```

```
pytest --disable-plugin-autoload -p NAME -p NAME2
```

```
[pytest]
addopts = ["--disable-plugin-autoload", "-p", "NAME", "-p", "NAME2"]
```

```
[pytest]
addopts =
    --disable-plugin-autoload
    -p NAME
    -p NAME2
```

Added in version 8.4: The *--disable-plugin-autoload* command-line flag.

Note

-p and *PYTEST_PLUGINS* are both ways to explicitly control which plugins are loaded, but they serve slightly different use-cases.

- *-p* loads (or disables with `-p no:<name>`) a plugin by name or entry point for a specific pytest invocation, and is processed early during startup.
- *PYTEST_PLUGINS* is a comma-separated list of Python modules that are imported and registered as plugins during startup. This mechanism is commonly used by test suites, for example when testing a plugin.

When explicitly controlling plugin loading (especially with *PYTEST_DISABLE_PLUGIN_AUTOLOAD* or *--disable-plugin-autoload*), avoid specifying the same plugin via multiple mechanisms. Registering the same plugin more than once can lead to errors during plugin registration.

Examples:

```
# Disable auto-loading and load only specific plugins for this invocation
PYTEST_DISABLE_PLUGIN_AUTOLOAD=1 pytest -p xdist
```

```
# Disable auto-loading and load plugin modules during startup
PYTEST_DISABLE_PLUGIN_AUTOLOAD=1 PYTEST_PLUGINS=mymodule.plugin,xdist pytest
```

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*: installed third-party modules discovered through *entry points* in their packaging metadata
- *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 installed third-party package *entry points*, unless the *PYTEST_DISABLE_PLUGIN_AUTOLOAD* environment variable is set.

5. by loading all plugins specified through the *PYTEST_PLUGINS* environment variable.
6. by loading all "initial" `conftest.py` files:
   - determine the test paths: specified on the command line, otherwise in *testpaths* if defined and running from the rootdir, otherwise the current dir
   - for each test path, load `conftest.py` and `test*/conftest.py` relative to the directory part of the test path, if exist. Before a `conftest.py` file is loaded, load `conftest.py` files in all of its parent directories. After a `conftest.py` file is loaded, recursively load all plugins specified in its *pytest_plugins* variable if present.

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)

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 cannot be implemented in conftest.py files which are not *initial* 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 packaging tools.

pytest looks up the `pytest11` entrypoint to discover its plugins, thus you can make your plugin available by defining it in your `pyproject.toml` file.

```
# sample ./pyproject.toml file
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "myproject"
classifiers = [
    "Framework :: Pytest",
]

[project.entry-points.pytest11]
myproject = "myproject.pluginmodule"
```

If a package is installed this way, `pytest` will load `myproject.pluginmodule` as a plugin which can define *hooks*. Confirm registration with `pytest --trace-config`

> **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.

Parameters

`names` – The module names to register.

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 entry point packaging metadata 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.addinivalue_line("markers", "cool_marker: this one is for cool tests.")
    config.addinivalue_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):
    name = request.config.getoption("name")

    def _hello(name=None):
        if not name:
            name = request.config.getoption("name")
        return f"Hello {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):
```

(continues on next page)

(continued from previous page)

```
            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 configuration file to tell pytest where to look for example files.

```
# content of pytest.toml
[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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
configfile: pytest.toml
collected 2 items

test_example.py ..                                                   [100%]
```

(continues on next page)

(continued from previous page)

```
============================ 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_runtest_*` 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 hook wrappers: 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, or will propagate an exception if they raised.

Here is an example definition of a hook wrapper:

```
import pytest


@pytest.hookimpl(wrapper=True)
def pytest_pyfunc_call(pyfuncitem):
```

(continues on next page)

(continued from previous page)

```
    do_something_before_next_hook_executes()

    # If the outcome is an exception, will raise the exception.
    res = yield

    new_res = post_process_result(res)

    # Override the return value to the plugin system.
    return new_res
```

The hook wrapper needs to return a result for the hook, or raise an exception.

In many cases, the wrapper only needs to perform tracing or other side effects around the actual hook implementations, in which case it can return the result value of the `yield`. The simplest (though useless) hook wrapper is `return (yield)`.

In other cases, the wrapper wants to adjust or adapt the result, in which case it can return a new value. If the result of the underlying hook is a mutable object, the wrapper may modify that result, but it's probably better to avoid it.

If the hook implementation failed with an exception, the wrapper can handle that exception using a `try-catch-finally` around the `yield`, by propagating it, suppressing it, or raising a different exception entirely.

For more information, consult the pluggy documentation about hook wrappers.

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
    ...


# Plugin 2
@pytest.hookimpl(trylast=True)
def pytest_collection_modifyitems(items):
    # will execute as late as possible
    ...


# Plugin 3
@pytest.hookimpl(wrapper=True)
def pytest_collection_modifyitems(items):
    # will execute even before the tryfirst one above!
    try:
        return (yield)
    finally:
        # will execute after all non-wrappers 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 the result from calling the non-wrappers, or raises an exception if the non-wrappers raised.

It's possible to use `tryfirst` and `trylast` also on hook wrappers in which case it will influence the ordering of hook wrappers 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 – The pytest plugin manager.

> **Note**
>
> This hook is incompatible with hook wrappers.

Use in conftest plugins

If a conftest plugin implements this hook, it will be called immediately when the conftest is registered.

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 `pytestconfig` 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 `conftest.py`.

Example:

```
def pytest_my_hook(config):
    """
    Print all active hooks to the screen.
    """
    print(config.hook)
```

Note

Unlike other hooks, the *pytest_generate_tests* hook is also discovered when defined inside a test module or test class. Other hooks must live in *conftest.py plugins* or external plugins. See *How to parametrize fixtures and test functions* and the *Hooks*.

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."""


# contents of myplugin.py
```

(continues on next page)

(continued from previous page)

```
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 *Item*s 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, *item.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 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()`/`unittest.skipIf()` style decorators
- `unittest.TestCase.setUp()`/`unittest.TestCase.tearDown()`
- `unittest.TestCase.setUpClass()`/`unittest.TestCase.tearDownClass()`
- `unittest.setUpModule()`/`unittest.tearDownModule()`
- `unittest.TestCase.subTest()` (since version `9.0`)

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

- load_tests protocol;

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
```

(continues on next page)

(continued from previous page)

```
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 reuse 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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 2 items

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")
```

(continues on next page)

(continued from previous page)

```
>       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:11: 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:14: 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 unittest

import pytest


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", encoding="utf-8")

    def test_method(self):
        with open("samplefile.ini", encoding="utf-8") 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
.                                                                    [100%]
1 passed in 0.12s
```

… 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 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.

2.19.1 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):
```

(continues on next page)

(continued from previous page)

```
    """teardown any state that was previously setup with a setup_module
    method.
    """
```

As of pytest-3.0, the `module` parameter is optional.

2.19.2 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.
    """
```

2.19.3 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.

2.20 How to set up bash completion

When using bash as your shell, `pytest` can use argcomplete (https://kislyuk.github.io/argcomplete/) 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)"
```

CHAPTER

THREE

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`.

capteesys
: Capture in the same manner as *capsys*, but also pass text through according to *--capture*.

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 doctests namespace.

monkeypatch
: Temporarily modify classes, functions, dictionaries, `os.environ`, and other objects.

pytestconfig
: Access to configuration values, pluginmanager and plugin hooks.

subtests
Enable declaring subtests inside test functions.

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:

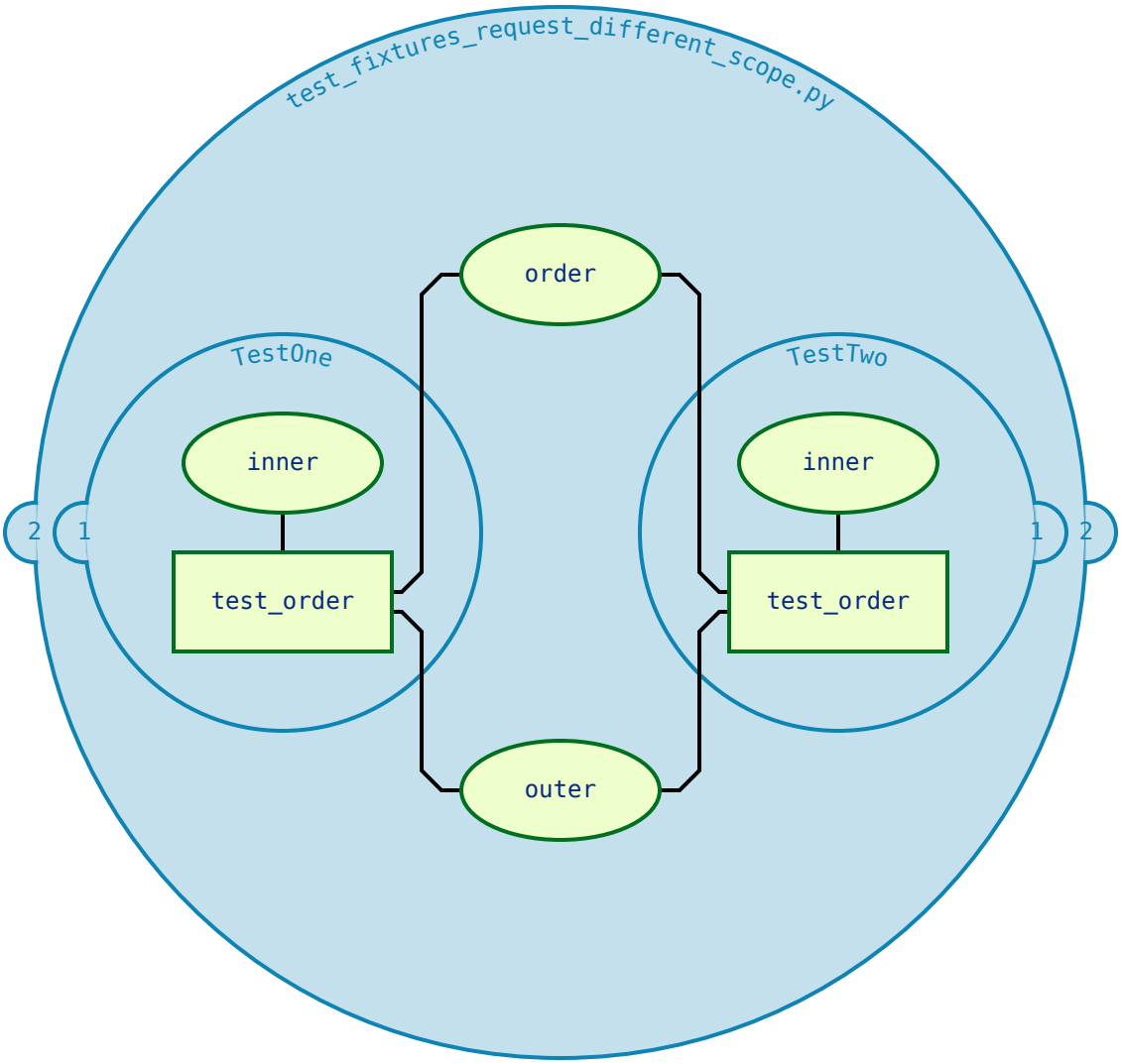
```
from __future__ import annotations

import pytest


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


@pytest.fixture
def outer(order, inner):
```

(continues on next page)

(continued from previous page)

```
    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:

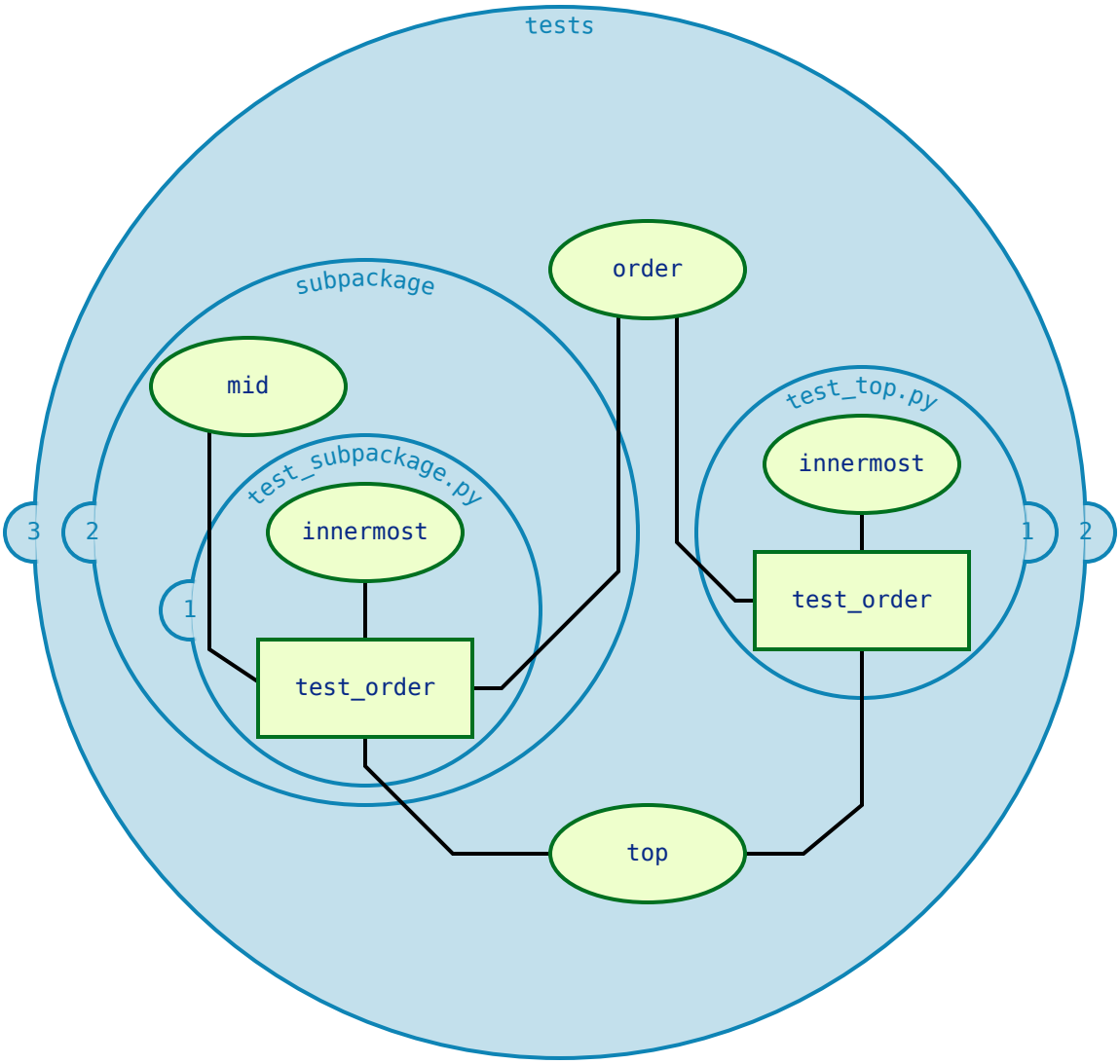
```
tests/
    __init__.py
```

(continues on next page)

(continued from previous page)

```
conftest.py
    # content of tests/conftest.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

    conftest.py
        # content of tests/subpackage/conftest.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.

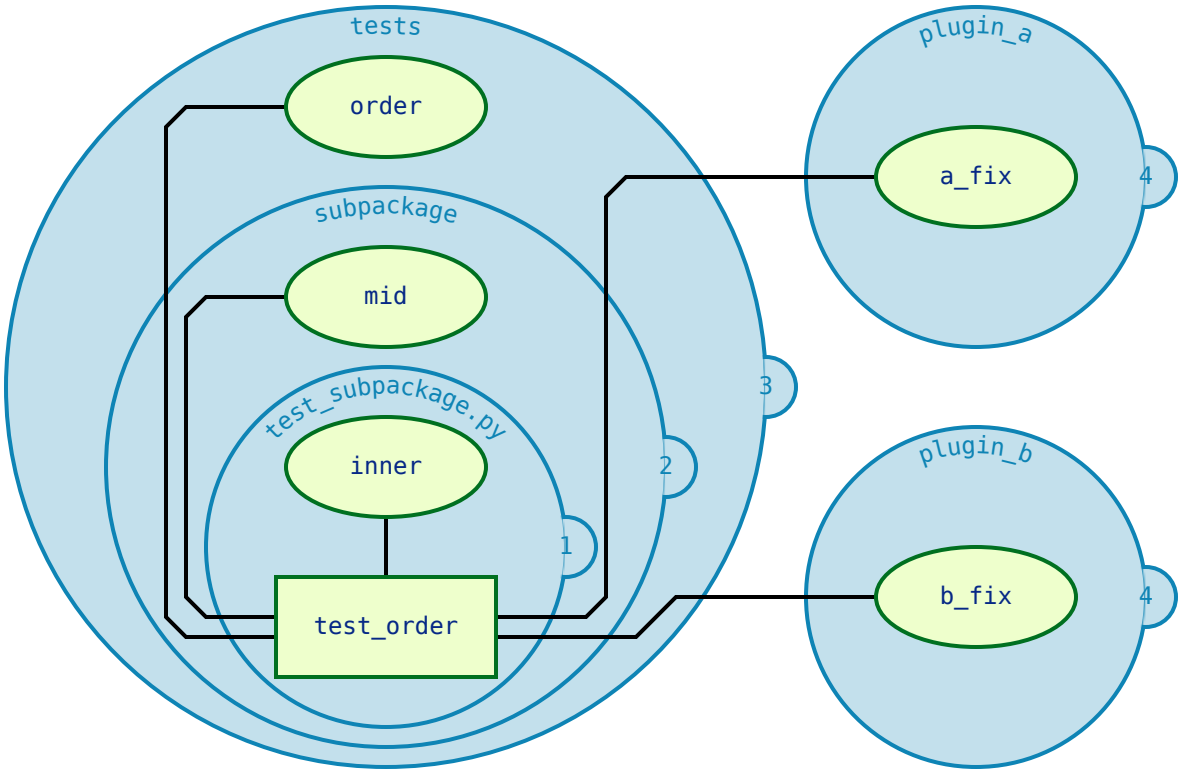
Because they're provided from outside the structure of your test suite, third-party plugins don't really provide a scope like `conftest.py` files and the directories in your test suite do. As a result, pytest will search for fixtures stepping out

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/`.

Note

pytest can tell you what fixtures are available for a given test if you call `pytest` along with the test's name (or the scope it's in), and provide the *`--fixtures`* flag, e.g. `pytest --fixtures test_something.py` (fixtures with names that start with _ will only be shown if you also provide the *`-v`* flag).

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

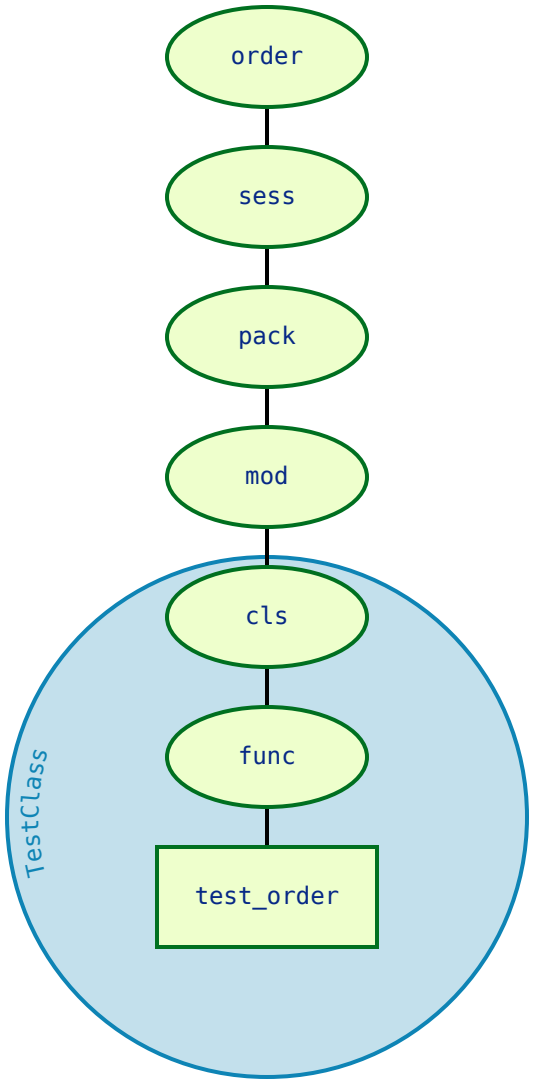
Within a function request for fixtures, those of higher-scopes (such as `session`) are executed before lower-scoped fixtures (such as `function` or `class`).

Here's an example:

```
from __future__ import annotations

import pytest


@pytest.fixture(scope="session")
def order():
    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:

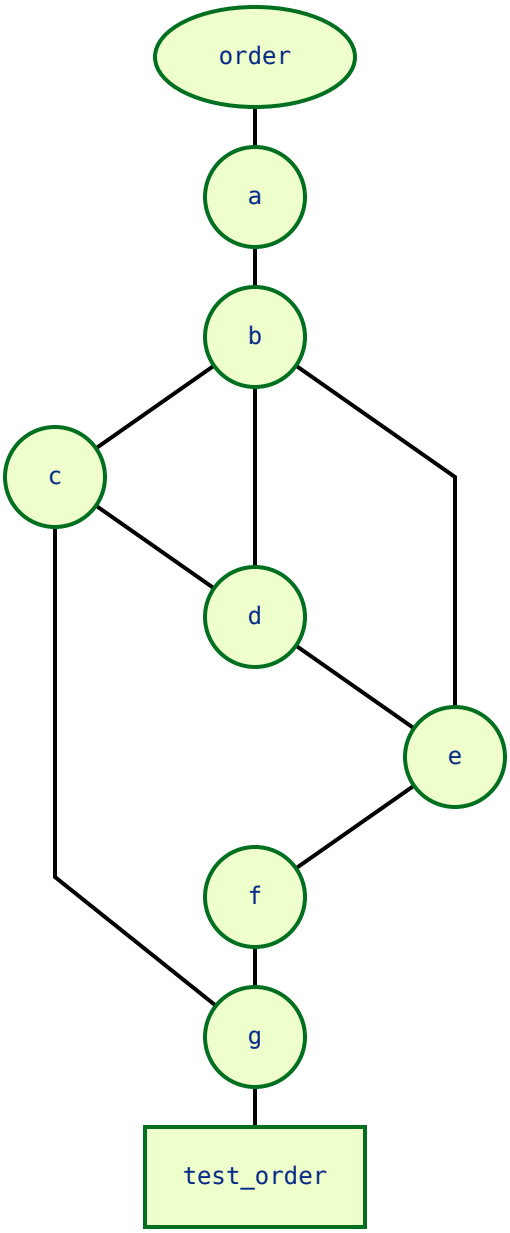
```
from __future__ import annotations

import pytest


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


@pytest.fixture
```

(continues on next page)

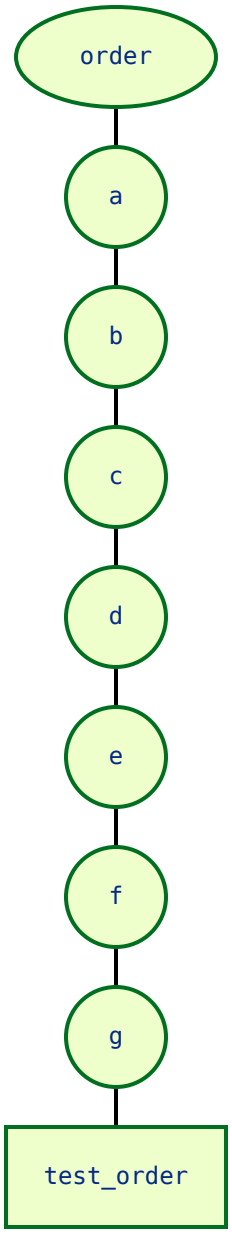
(continued from previous page)

```
def a(order):
    order.append("a")


@pytest.fixture
def b(a, order):
    order.append("b")


@pytest.fixture
def c(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 looks like this:

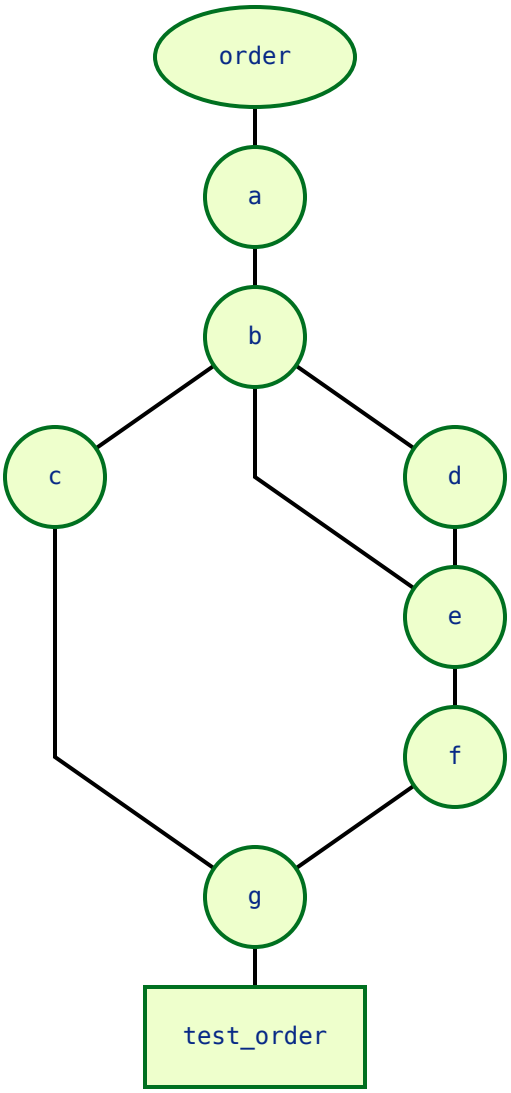


The rules provided by each fixture (as to what fixture(s) each one has to come after) are comprehensive enough that it can be flattened to this:



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.

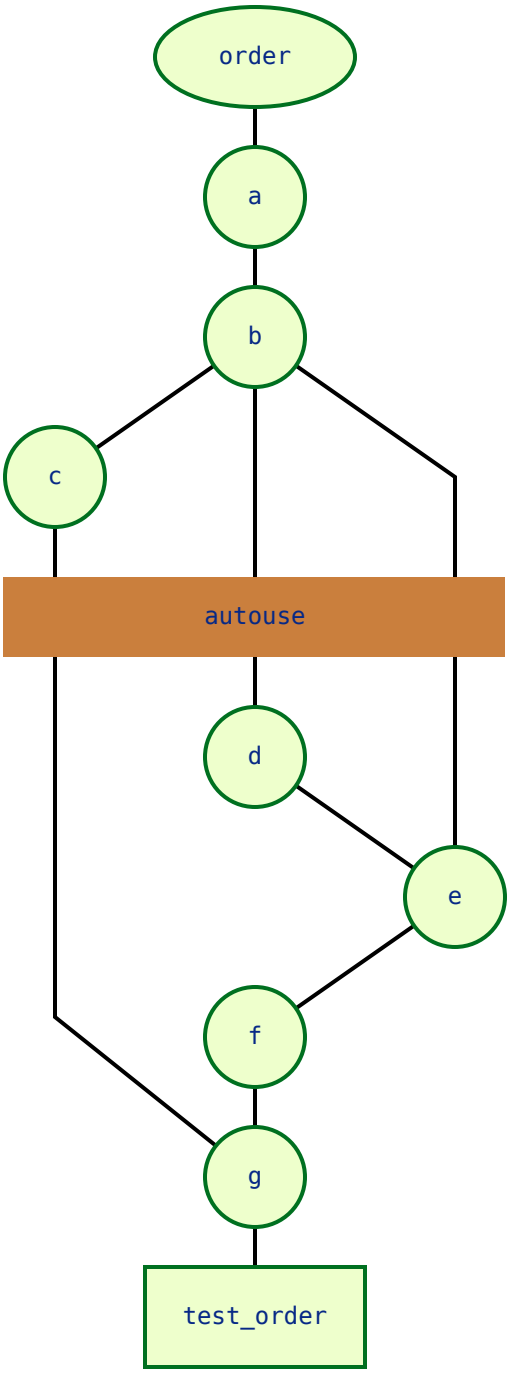
So if fixture `a` is autouse and fixture `b` is not, but fixture `a` requests fixture `b`, then fixture `b` will effectively be an autouse fixture as well, but only for the tests that `a` applies to.

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.

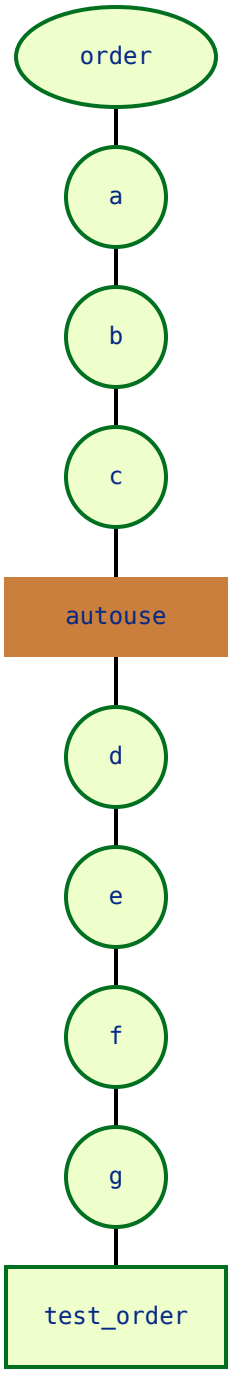
So if the test file looked like this:

```
from __future__ import annotations

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:

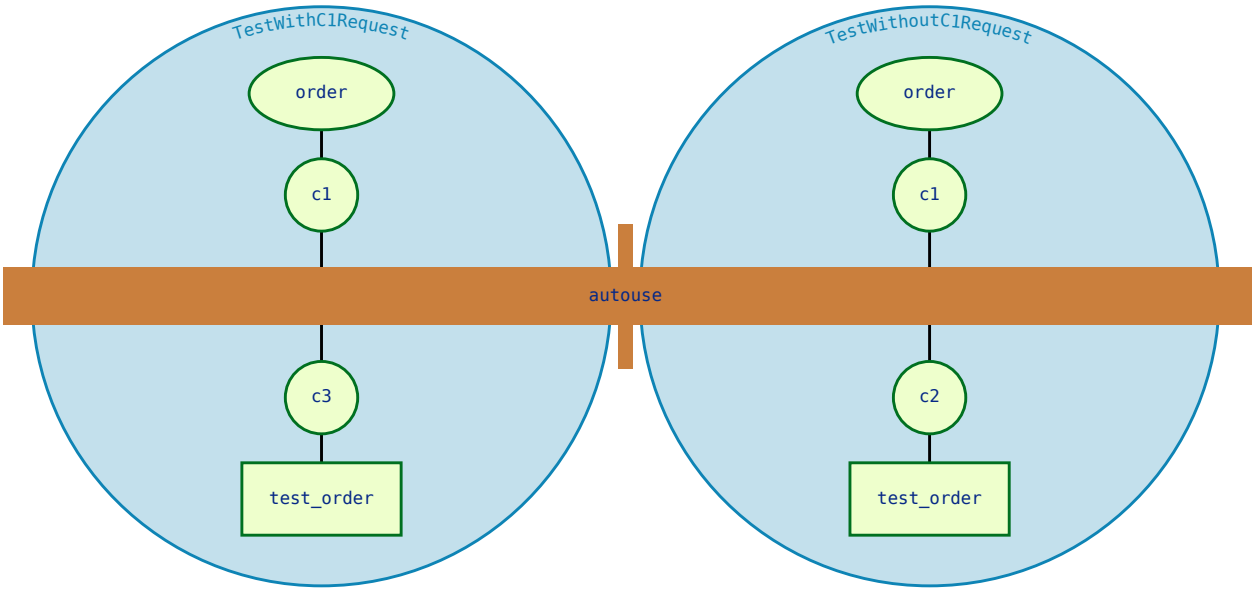
```
from __future__ import annotations

import pytest
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture(scope="class")
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:

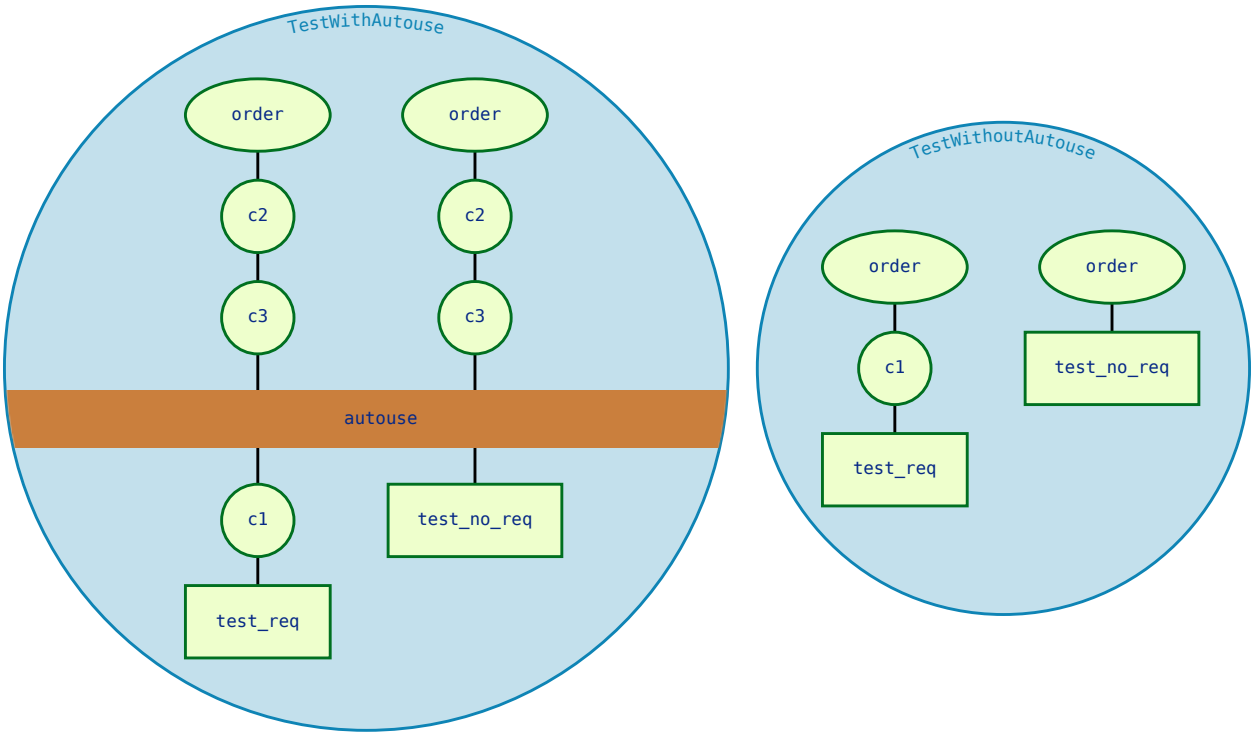


But just because one autouse fixture requested a non-autouse fixture, that doesn't mean the non-autouse fixture becomes an autouse fixture for all contexts that it can apply to. It only effectively becomes an autouse fixture for the contexts the real autouse fixture (the one that requested the non-autouse fixture) can apply to.

For example, take a look at this test file:

```
from __future__ import annotations

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`.

Note

pytest can tell you what order the fixtures will execute in for a given test if you call `pytest` along with the test's name (or the scope it's in), and provide the *`--setup-plan`* flag, e.g. `pytest --setup-plan test_something.py` (fixtures with names that start with _ will only be shown if you also provide the *`-v`* flag).

3.2 Pytest Plugin List

Below is an automated compilation of `pytest` plugins available on PyPI. It includes PyPI projects whose names begin with `pytest-` or `pytest_` and a handful of manually selected projects. Packages classified as inactive are excluded.

For detailed insights into how this list is generated, please refer to the update script.

Warning

Please be aware that this list is not a curated collection of projects and does not undergo a systematic review process. It serves purely as an informational resource to aid in the discovery of `pytest` plugins.

Do not presume any endorsement from the `pytest` project or its developers, and always conduct your own quality assessment before incorporating any of these plugins into your own projects.

This list contains 1947 plugins.

databricks-labs-pytester
last release: Oct 17, 2025, *status*: 4 - Beta, *requires*: pytest>=8.3

Python Testing for Databricks

logassert
last release: Aug 14, 2025, *status*: 5 - Production/Stable, *requires*: pytest; extra == "dev"

Simple but powerful assertion and verification of logged lines

logot
last release: Mar 31, 2026, *status*: 5 - Production/Stable, *requires*: pytest<10,>=7; extra == "pytest"

Test whether your code is logging correctly

nuts
last release: Nov 17, 2025, *status*: N/A, *requires*: pytest<8,>=7

Network Unit Testing System

pytest-abort
last release: Feb 11, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Pytest plugin + helpers for attributing hard crashes (SIGSEGV/SIGABRT) to the last running test and keeping pytest-html reports mergeable.

pytest-abq
last release: Apr 07, 2023, *status*: N/A, *requires*: N/A

Pytest integration for the ABQ universal test runner.

pytest-abstracts
last release: May 25, 2022, *status*: N/A, *requires*: N/A

A contextmanager pytest fixture for handling multiple mock abstracts

pytest-accept
last release: Mar 01, 2026, *status*: N/A, *requires*: pytest>=7

pytest-adaptavist
last release: Oct 13, 2022, *status*: N/A, *requires*: pytest (>=5.4.0)

pytest plugin for generating test execution results within Jira Test Management (tm4j)

pytest-adaptavist-fixed
last release: Jan 17, 2025, *status*: N/A, *requires*: pytest>=5.4.0

pytest plugin for generating test execution results within Jira Test Management (tm4j)

pytest-adbc-replay
last release: Mar 13, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.0

pytest plugin to record and replay ADBC database queries

pytest-addons-test
last release: Aug 02, 2021, *status*: N/A, *requires*: pytest (>=6.2.4,<7.0.0)

用于测试pytest的插件

pytest-adf
last release: May 10, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Pytest plugin for writing Azure Data Factory integration tests

pytest-adf-azure-identity
last release: Mar 06, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Pytest plugin for writing Azure Data Factory integration tests

pytest-ads-testplan
last release: Sep 15, 2022, *status*: N/A, *requires*: N/A

Azure DevOps Test Case reporting for pytest tests

pytest-adversarial
last release: Jan 22, 2026, *status*: N/A, *requires*: pytest>=7.0.0

Generate adversarial pytest tests using LLM

pytest-affected
last release: Nov 06, 2023, *status*: N/A, *requires*: N/A

pytest-agent
last release: Nov 25, 2021, *status*: N/A, *requires*: N/A

Service that exposes a REST API that can be used to interract remotely with Pytest. It is shipped with a dashboard that enables running tests in a more convenient way.

pytest-agentcontract
last release: Feb 18, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Deterministic CI tests for LLM agent trajectories — record once, replay offline, assert contracts

pytest-agent-digest
last release: Mar 21, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

A Pytest plugin to generate a Markdown report for AI Agents

pytest-agent-evals
last release: Mar 13, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

Pytest plugin for evaluating AI Agents

pytest-agent-health
last release: Apr 03, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Catch silent agent failures in CI. Behavior lint for LLM agents, powered by agent-failure-debugger.

pytest-agents
last release: Feb 20, 2026, *status*: 3 - Alpha, *requires*: pytest>=8.0.0

Pytest plugin framework with AI agent capabilities for multi-agent testing

pytest-aggreport
last release: Mar 07, 2021, *status*: 4 - Beta, *requires*: pytest (>=6.2.2)

pytest plugin for pytest-repeat that generate aggregate report of the same test cases with additional statistics details.

pytest-ai
last release: Jan 22, 2025, *status*: N/A, *requires*: N/A

A Python package to generate regular, edge-case, and security HTTP tests.

pytest-ai1899
last release: Mar 13, 2024, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for connecting to ai1899 smart system stack

pytest-aio
last release: Feb 12, 2026, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin for testing async python code

pytest-aioboto3
last release: Jan 17, 2025, *status*: N/A, *requires*: N/A

Aioboto3 Pytest with Moto

pytest-aiofiles
last release: May 14, 2017, *status*: 5 - Production/Stable, *requires*: N/A

pytest fixtures for writing aiofiles tests with pyfakefs

pytest-aiogram
last release: May 06, 2023, *status*: N/A, *requires*: N/A

pytest-aiohttp
last release: Jan 23, 2025, *status*: 4 - Beta, *requires*: pytest>=6.1.0

Pytest plugin for aiohttp support

pytest-aiohttp-client
last release: Jan 10, 2023, *status*: N/A, *requires*: pytest (>=7.2.0,<8.0.0)

Pytest `client` fixture for the Aiohttp

pytest-aiohttp-mock
last release: Sep 13, 2025, *status*: 3 - Alpha, *requires*: pytest>=8

Send responses to aiohttp.

pytest-aiohutils
last release: Feb 10, 2026, *status*: N/A, *requires*: pytest

Pytest plugin providing fixtures and configuration for aiohutils projects (offline, record, cleanup modes).

pytest-aiomoto
last release: Jun 24, 2023, *status*: N/A, *requires*: pytest (>=7.0,<8.0)

pytest-aiomoto

pytest-aioresponses
last release: Jan 02, 2025, *status*: 4 - Beta, *requires*: pytest>=3.5.0

py.test integration for aioresponses

pytest-aioworkers
last release: Dec 26, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=8.3.4

A plugin to test aioworkers project with pytest

pytest-airflow
last release: Apr 03, 2019, *status*: 3 - Alpha, *requires*: pytest (>=4.4.0)

pytest support for airflow.

pytest-airflow-utils
last release: Nov 15, 2021, *status*: N/A, *requires*: N/A

pytest-alembic
last release: May 27, 2025, *status*: N/A, *requires*: pytest>=7.0

A pytest plugin for verifying alembic migrations.

pytest-alerts
last release: Feb 21, 2025, *status*: 4 - Beta, *requires*: pytest>=7.4.0

A pytest plugin for sending test results to Slack and Telegram

pytest-allclose
last release: Jul 30, 2019, *status*: 5 - Production/Stable, *requires*: pytest

Pytest fixture extending Numpy's allclose function

pytest-allure-adaptor
last release: Jan 10, 2018, *status*: N/A, *requires*: pytest (>=2.7.3)

Plugin for py.test to generate allure xml reports

pytest-allure-adaptor2
last release: Oct 14, 2020, *status*: N/A, *requires*: pytest (>=2.7.3)

Plugin for py.test to generate allure xml reports

pytest-allure-collection
last release: Apr 13, 2023, *status*: N/A, *requires*: pytest

pytest plugin to collect allure markers without running any tests

pytest-allure-dsl
last release: Oct 25, 2020, *status*: 4 - Beta, *requires*: pytest

pytest plugin to test case doc string dls instructions

pytest-allure-host
last release: Nov 03, 2025, *status*: 3 - Alpha, *requires*: N/A

Publish Allure static reports to private S3 behind CloudFront with history preservation

pytest-allure-id2history
last release: May 14, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Overwrite allure history id with testcase full name and testcase id if testcase has id, exclude parameters.

pytest-allure-intersection
last release: Oct 27, 2022, *status*: N/A, *requires*: pytest (<5)

pytest-allure-spec-coverage
last release: Oct 26, 2021, *status*: N/A, *requires*: pytest

The pytest plugin aimed to display test coverage of the specs(requirements) in Allure

pytest-allure-step
last release: Jul 13, 2025, *status*: 3 - Alpha, *requires*: pytest>=6.0.0

Enhanced logging integration with Allure reports for pytest

pytest-alphamoon
last release: Dec 30, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=3.5.0)

Static code checks used at Alphamoon

pytest-amaranth-sim
last release: Feb 18, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Fixture to automate running Amaranth simulations

pytest-ampel-core
last release: Dec 17, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin to provide AmpelContext fixtures in pytest

pytest-analyzer
last release: Feb 21, 2024, *status*: N/A, *requires*: pytest <8.0.0,>=7.3.1

this plugin allows to analyze tests in pytest project, collect test metadata and sync it with testomat.io TCM system

pytest-android
last release: Feb 21, 2019, *status*: 3 - Alpha, *requires*: pytest

This fixture provides a configured "driver" for Android Automated Testing, using uiautomator2.

pytest-anki
last release: Jul 31, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A pytest plugin for testing Anki add-ons

pytest-annotate
last release: Jun 07, 2022, *status*: 3 - Alpha, *requires*: pytest (<8.0.0,>=3.2.0)

pytest-annotate: Generate PyAnnotate annotations from your pytest tests.

pytest-annotated
last release: Sep 30, 2024, *status*: N/A, *requires*: pytest>=8.3.3

Pytest plugin to allow use of Annotated in tests to resolve fixtures

pytest-ansible
last release: Apr 01, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6

Plugin for pytest to simplify calling ansible modules from tests or fixtures

pytest-ansible-playbook
last release: Mar 08, 2019, *status*: 4 - Beta, *requires*: N/A

Pytest fixture which runs given ansible playbook file.

pytest-ansible-playbook-runner
last release: Dec 02, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.1.0)

Pytest fixture which runs given ansible playbook file.

pytest-ansible-units
last release: Apr 14, 2022, *status*: N/A, *requires*: N/A

A pytest plugin for running unit tests within an ansible collection

pytest-antilru
last release: Jul 28, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7; python_version >= "3.10"

Bust functools.lru_cache when running pytest to avoid test pollution

pytest-anyio
last release: Jun 29, 2021, *status*: N/A, *requires*: pytest

The pytest anyio plugin is built into anyio. You don't need this package.

pytest-anything
last release: Jan 18, 2024, *status*: N/A, *requires*: pytest

Pytest fixtures to assert anything and something

pytest-aoc
last release: Dec 02, 2023, *status*: 5 - Production/Stable, *requires*: pytest ; extra == 'test'

Downloads puzzle inputs for Advent of Code and synthesizes PyTest fixtures

pytest-aoreporter
last release: Jun 27, 2022, *status*: N/A, *requires*: N/A

pytest report

pytest-api
last release: May 12, 2022, *status*: N/A, *requires*: pytest (>=7.1.1,<8.0.0)

An ASGI middleware to populate OpenAPI Specification examples from pytest functions

pytest-apibean
last release: Dec 30, 2025, *status*: N/A, *requires*: pytest

Pytest plugin providing apibean-based API testing fixtures integrated with apibean-client, designed for testing apibean REST services and datacore backends.

pytest-api-cov
last release: Apr 14, 2026, *status*: N/A, *requires*: pytest>=6.0.0

Pytest Plugin to provide API Coverage statistics for Python Web Frameworks

pytest-api-coverage
last release: Mar 24, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

Pytest plugin for API test coverage analysis

pytest-api-framework
last release: Jun 22, 2025, *status*: N/A, *requires*: pytest==7.2.2

pytest framework

pytest-api-framework-alpha
last release: Apr 14, 2026, *status*: N/A, *requires*: pytest==7.2.2

pytest-api-soup
last release: Aug 27, 2022, *status*: N/A, *requires*: N/A

Validate multiple endpoints with unit testing using a single source of truth.

pytest-apistellar
last release: Jun 18, 2019, *status*: N/A, *requires*: N/A

apistellar plugin for pytest.

pytest-apiver
last release: Jun 21, 2024, *status*: N/A, *requires*: pytest

pytest-appengine
last release: Feb 27, 2017, *status*: N/A, *requires*: N/A

AppEngine integration that works well with pytest-django

pytest-appium
last release: Dec 05, 2019, *status*: N/A, *requires*: N/A

Pytest plugin for appium

pytest-appium-scheduler
last release: Apr 13, 2026, *status*: N/A, *requires*: pytest>=7.0

Pytest plugin for Appium device scheduling and driver lifecycle management.

pytest-approval
last release: Mar 24, 2026, *status*: N/A, *requires*: pytest>=8.3.5

A simple approval test library utilizing external diff programs such as PyCharm and Visual Studio Code to compare approved and received output.

pytest-approvaltests
last release: May 08, 2022, *status*: 4 - Beta, *requires*: pytest (>=7.0.1)

A plugin to use approvaltests with pytest

pytest-approvaltests-geo
last release: Jul 14, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Extension for ApprovalTests.Python specific to geo data verification

pytest-archon
last release: Sep 19, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.2

Rule your architecture like a real developer

pytest-argus
last release: Jun 24, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=6.2.4)

pyest results colection plugin

pytest-argus-reporter
last release: Mar 03, 2026, *status*: 4 - Beta, *requires*: pytest~=9.0.0; extra == "dev"

A simple plugin to report results of test into argus

pytest-argus-server
last release: Mar 05, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin that provides a running Argus API server for tests

pytest-arrakis
last release: Apr 15, 2026, *status*: 3 - Alpha, *requires*: pytest

Pytest plugin providing Arrakis fixtures for testing

pytest-arraydiff
last release: Nov 27, 2023, *status*: 4 - Beta, *requires*: pytest >=4.6

pytest plugin to help with comparing array output from tests

pytest-artifact
last release: Feb 09, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Pytest plugin for managing test artifacts

pytest-artifacts
last release: Apr 17, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Pytest plugin for managing test artifacts

pytest-asdf-plugin
last release: Aug 18, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7

Pytest plugin for testing ASDF schemas

pytest-asgi-server
last release: Dec 12, 2020, *status*: N/A, *requires*: pytest (>=5.4.1)

Convenient ASGI client/server fixtures for Pytest

pytest-aspec
last release: Dec 20, 2023, *status*: 4 - Beta, *requires*: N/A

A rspec format reporter for pytest

pytest-asptest
last release: Apr 28, 2018, *status*: 4 - Beta, *requires*: N/A

test Answer Set Programming programs

pytest-assay
last release: Mar 30, 2026, *status*: 4 - Beta, *requires*: N/A

Evaluation framework for Pydantic AI agents

pytest-assertcount
last release: Oct 23, 2022, *status*: N/A, *requires*: pytest (>=5.0.0)

Plugin to count actual number of asserts in pytest

pytest-assertions
last release: Apr 27, 2022, *status*: N/A, *requires*: N/A

Pytest Assertions

pytest-assert-type
last release: Oct 26, 2025, *status*: 3 - Alpha, *requires*: pytest>=6.2.0

Use typing.assert_type() to test runtime behavior

pytest-assertutil
last release: May 10, 2019, *status*: N/A, *requires*: N/A

pytest-assertutil

pytest-assert-utils
last release: Apr 14, 2022, *status*: 3 - Alpha, *requires*: N/A

Useful assertion utilities for use with pytest

pytest-assist
last release: Oct 29, 2025, *status*: 4 - Beta, *requires*: pytest

pytest plugin library

pytest-assume
last release: Jun 24, 2021, *status*: N/A, *requires*: pytest (>=2.7)

A pytest plugin that allows multiple failures per test

pytest-assurka
last release: Aug 04, 2022, *status*: N/A, *requires*: N/A

A pytest plugin for Assurka Studio

pytest-ast-back-to-python
last release: Sep 29, 2019, *status*: 4 - Beta, *requires*: N/A

A plugin for pytest devs to view how assertion rewriting recodes the AST

pytest-asteroid
last release: Aug 15, 2022, *status*: N/A, *requires*: pytest (>=6.2.5,<8.0.0)

PyTest plugin for docker-based testing on database images

pytest-astropy
last release: Sep 26, 2023, *status*: 5 - Production/Stable, *requires*: pytest >=4.6

Meta-package containing dependencies for testing

pytest-astropy-header
last release: Sep 06, 2022, *status*: 3 - Alpha, *requires*: pytest (>=4.6)

pytest plugin to add diagnostic information to the header of the test output

pytest-ast-transformer
last release: May 04, 2019, *status*: 3 - Alpha, *requires*: pytest

pytest_async
last release: Feb 26, 2020, *status*: N/A, *requires*: N/A

pytest-async - Run your coroutine in event loop without decorator

pytest-async-benchmark
last release: May 28, 2025, *status*: N/A, *requires*: pytest>=8.3.5

pytest-async-benchmark: Modern pytest benchmarking for async code.

pytest-async-generators
last release: Jul 05, 2023, *status*: N/A, *requires*: N/A

Pytest fixtures for async generators

pytest-asyncio
last release: Apr 15, 2026, *status*: 5 - Production/Stable, *requires*: pytest<10,>=8.2

Pytest support for asyncio

pytest-asyncio-concurrent
last release: Apr 09, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Pytest plugin to execute python async tests concurrently.

pytest-asyncio-cooperative
last release: Jun 24, 2025, *status*: N/A, *requires*: N/A

Run all your asynchronous tests cooperatively.

pytest-asyncio-network-simulator
last release: Jul 31, 2018, *status*: 3 - Alpha, *requires*: pytest (<3.7.0,>=3.3.2)

pytest-asyncio-network-simulator: Plugin for pytest for simulator the network in tests

pytest-async-mongodb
last release: Oct 18, 2017, *status*: 5 - Production/Stable, *requires*: pytest (>=2.5.2)

pytest plugin for async MongoDB

pytest-async-sqlalchemy
last release: Oct 07, 2021, *status*: 4 - Beta, *requires*: pytest (>=6.0.0)

Database testing fixtures using the SQLAlchemy asyncio API

pytest-atf-allure
last release: Nov 29, 2023, *status*: N/A, *requires*: pytest (>=7.4.2,<8.0.0)

基于allure-pytest进行自定义

pytest-atomic
last release: Nov 24, 2018, *status*: 4 - Beta, *requires*: N/A

Skip rest of tests if previous test failed.

pytest-atstack
last release: Jan 02, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A simple plugin to use with pytest

pytest-attempt-summary
last release: Jan 04, 2026, *status*: N/A, *requires*: pytest>=7.0

Enhanced Allure Attempt Summary for Playwright + Pytest

pytest-attrib
last release: May 24, 2016, *status*: 4 - Beta, *requires*: N/A

pytest plugin to select tests based on attributes similar to the nose-attrib plugin

pytest-attributes
last release: Jun 24, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin that allows users to add attributes to their tests. These attributes can then be referenced by fixtures or the test itself.

pytest-audioeval
last release: Mar 18, 2026, *status*: 4 - Beta, *requires*: pytest>=8.0

Pytest plugin for STT/TTS integration testing with httpx, metrics, and embedded audio samples.

pytest-austin
last release: Oct 11, 2020, *status*: 4 - Beta, *requires*: N/A

Austin plugin for pytest

pytest-auto-api2-cli
last release: Mar 26, 2026, *status*: N/A, *requires*: pytest==8.4.1

CLI for generating and running pytest-auto-api2 test cases.

pytest-autocap
last release: May 15, 2022, *status*: N/A, *requires*: pytest (<7.2,>=7.1.2)

automatically capture test & fixture stdout/stderr to files

pytest-autochecklog
last release: Apr 25, 2015, *status*: 4 - Beta, *requires*: N/A

automatically check condition and log all the checks

pytest-autofixture
last release: Aug 01, 2024, *status*: N/A, *requires*: pytest>=8

simplify pytest fixtures

pytest-autofocus
last release: Dec 02, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Auto-focus plugin: run only @pytest.mark.focus tests when –auto-focus is set

pytest-automation
last release: Apr 24, 2024, *status*: N/A, *requires*: pytest>=7.0.0

pytest plugin for building a test suite, using YAML files to extend pytest parameterize functionality.

pytest-automock
last release: May 16, 2023, *status*: N/A, *requires*: pytest ; extra == 'dev'

Pytest plugin for automatical mocks creation

pytest-auto-parametrize
last release: Oct 02, 2016, *status*: 3 - Alpha, *requires*: N/A

pytest plugin: avoid repeating arguments in parametrize

pytest-autoprofile
last release: Mar 13, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

`line_profiler.autoprofile`-ing your `pytest` test suite

pytest-autotest
last release: Aug 25, 2021, *status*: N/A, *requires*: pytest

This fixture provides a configured "driver" for Android Automated Testing, using uiautomator2.

pytest-aviator
last release: Nov 04, 2022, *status*: 4 - Beta, *requires*: pytest

Aviator's Flakybot pytest plugin that automatically reruns flaky tests.

pytest-avoidance
last release: May 23, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Makes pytest skip tests that don not need rerunning

pytest-awaiting-fix
last release: Aug 09, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A simple plugin to use with pytest for traceability across Jira and disabled automated tests

pytest-aws
last release: Oct 04, 2017, *status*: 4 - Beta, *requires*: N/A

pytest plugin for testing AWS resource configurations

pytest-aws-apigateway
last release: May 24, 2024, *status*: 4 - Beta, *requires*: pytest

pytest plugin for AWS ApiGateway

pytest-aws-config
last release: May 28, 2021, *status*: N/A, *requires*: N/A

Protect your AWS credentials in unit tests

pytest-aws-fixtures
last release: Nov 11, 2025, *status*: N/A, *requires*: pytest<10.0.0,>=8.0.0

A series of fixtures to use in integration tests involving actual AWS services.

pytest-aws-fixtures-293984
last release: Dec 04, 2025, *status*: 3 - Alpha, *requires*: N/A

AWS configuration utilities for Python applications

pytest-axe
last release: Nov 12, 2018, *status*: N/A, *requires*: pytest (>=3.0.0)

pytest plugin for axe-selenium-python

pytest-axe-playwright-snapshot
last release: Jul 25, 2023, *status*: N/A, *requires*: pytest

A pytest plugin that runs Axe-core on Playwright pages and takes snapshots of the results.

pytest-azure
last release: Jan 18, 2023, *status*: 3 - Alpha, *requires*: pytest

Pytest utilities and mocks for Azure

pytest-azure-devops
last release: Jul 16, 2025, *status*: 4 - Beta, *requires*: pytest>=3.5.0

Simplifies using azure devops parallel strategy (https://docs.microsoft.com/en-us/azure/devops/pipelines/test/parallel-testing-any-test-runner) with pytest.

pytest-azurepipelines
last release: Oct 06, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=5.0.0)

Formatting PyTest output for Azure Pipelines UI

pytest-balance
last release: Apr 09, 2026, *status*: 3 - Alpha, *requires*: pytest>=8

Intelligent test distribution for pytest based on actual execution times, not file count

pytest-bandit
last release: Feb 23, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A bandit plugin for pytest

pytest-bandit-xayon
last release: Oct 17, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A bandit plugin for pytest

pytest-base-url
last release: Jan 31, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

pytest plugin for URL based testing

pytest-bashdoctest
last release: Oct 03, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin for testing bash command examples in markdown documentation

pytest-batch-regression
last release: May 08, 2024, *status*: N/A, *requires*: pytest>=6.0.0

A pytest plugin to repeat the entire test suite in batches.

pytest-bazel
last release: Oct 31, 2025, *status*: 4 - Beta, *requires*: pytest

A pytest runner with bazel support

pytest-bdd
last release: Dec 05, 2024, *status*: 6 - Mature, *requires*: pytest>=7.0.0

BDD for pytest

pytest-bdd-html
last release: Nov 22, 2022, *status*: 3 - Alpha, *requires*: pytest (!=6.0.0,>=5.0)

pytest plugin to display BDD info in HTML test report

pytest-bdd-md-report
last release: Feb 07, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Markdown test report formatter for pytest-bdd with pytest-playwright screenshot support

pytest-bdd-ng
last release: Nov 26, 2024, *status*: 4 - Beta, *requires*: pytest>=5.2

BDD for pytest

pytest-bdd-property
last release: Mar 12, 2026, *status*: 3 - Alpha, *requires*: pytest>=8.0

Property-based testing plugin for pytest-bdd — express universal invariants in standard Gherkin, executed by Hypothesis

pytest-bdd-report
last release: Dec 29, 2025, *status*: N/A, *requires*: pytest>=7.1.3

A pytest-bdd plugin for generating useful and informative BDD test reports

pytest-bdd-reporter
last release: Oct 14, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=6.0.0

Enterprise-grade BDD test reporting with interactive dashboards, suite management, and comprehensive email integration

pytest-bdd-splinter
last release: Aug 12, 2019, *status*: 5 - Production/Stable, *requires*: pytest (>=4.0.0)

Common steps for pytest bdd and splinter integration

pytest-bdd-web
last release: Jan 02, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A simple plugin to use with pytest

pytest-bdd-wrappers
last release: Feb 11, 2020, *status*: 2 - Pre-Alpha, *requires*: N/A

pytest-beacon
last release: Apr 17, 2026, *status*: 3 - Alpha, *requires*: pytest>=9.0.0

Highly customizable pytest reporting plugin combining AI-optimized CTRF reports and rich test metrics

pytest-beakerlib
last release: Mar 17, 2017, *status*: 5 - Production/Stable, *requires*: pytest

A pytest plugin that reports test results to the BeakerLib framework

pytest-beartype
last release: Oct 31, 2024, *status*: N/A, *requires*: pytest

Pytest plugin to run your tests with beartype checking enabled.

pytest-bec-e2e
last release: Apr 16, 2026, *status*: 3 - Alpha, *requires*: pytest

BEC pytest plugin for end-to-end tests

pytest-beds
last release: Jun 07, 2016, *status*: 4 - Beta, *requires*: N/A

Fixtures for testing Google Appengine (GAE) apps

pytest-beeprint
last release: Jul 04, 2023, *status*: 4 - Beta, *requires*: N/A

use icdiff for better error messages in pytest assertions

pytest-bench
last release: Jul 21, 2014, *status*: 3 - Alpha, *requires*: N/A

Benchmark utility that plugs into pytest.

pytest-benchmark
last release: Nov 09, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=8.1

A ``pytest`` fixture for benchmarking code. It will group the tests into rounds that are calibrated to the chosen timer.

pytest-better-datadir
last release: Mar 13, 2023, *status*: N/A, *requires*: N/A

A small example package

pytest-better-parametrize
last release: Mar 05, 2024, *status*: 4 - Beta, *requires*: pytest >=6.2.0

Better description of parametrized test cases

pytest-bg-process
last release: Jan 24, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Pytest plugin to initialize background process

pytest-bigchaindb
last release: Jan 24, 2022, *status*: 4 - Beta, *requires*: N/A

A BigchainDB plugin for pytest.

pytest-bigquery-mock
last release: Dec 28, 2022, *status*: N/A, *requires*: pytest (>=5.0)

Provides a mock fixture for python bigquery client

pytest-bisect-tests
last release: Jun 09, 2024, *status*: N/A, *requires*: N/A

Find tests leaking state and affecting other

pytest-black
last release: Dec 15, 2024, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin to enable format checking with black

pytest-black-multipy
last release: Jan 14, 2021, *status*: 5 - Production/Stable, *requires*: pytest (!=3.7.3,>=3.5) ; extra == 'testing'

Allow '–black' on older Pythons

pytest-black-ng
last release: Oct 20, 2022, *status*: 4 - Beta, *requires*: pytest (>=7.0.0)

A pytest plugin to enable format checking with black

pytest-blame
last release: May 04, 2019, *status*: N/A, *requires*: pytest (>=4.4.0)

A pytest plugin helps developers to debug by providing useful commits history.

pytest-blender
last release: Apr 18, 2026, *status*: N/A, *requires*: pytest

Blender Pytest plugin.

pytest-blink1
last release: Jan 07, 2018, *status*: 4 - Beta, *requires*: N/A

Pytest plugin to emit notifications via the Blink(1) RGB LED

pytest-blockage
last release: Dec 21, 2021, *status*: N/A, *requires*: pytest

Disable network requests during a test run.

pytest-blocker
last release: Sep 07, 2015, *status*: 4 - Beta, *requires*: N/A

pytest plugin to mark a test as blocker and skip all other tests

pytest-b-logger
last release: Dec 16, 2025, *status*: N/A, *requires*: pytest

BLogger is a Pytest plugin for enhanced test logging and generating convenient and lightweight reports.

pytest-blue
last release: Sep 05, 2022, *status*: N/A, *requires*: N/A

A pytest plugin that adds a `blue` fixture for printing stuff in blue.

pytest-board
last release: Jan 20, 2019, *status*: N/A, *requires*: N/A

Local continuous test runner with pytest and watchdog.

pytest-boardfarm3
last release: Sep 15, 2025, *status*: N/A, *requires*: pytest

Integrate boardfarm as a pytest plugin.

pytest-boilerplate
last release: Sep 12, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=4.0.0

The pytest plugin for your Django Boilerplate.

pytest-bonsai
last release: Apr 08, 2025, *status*: N/A, *requires*: pytest>=6

pytest-boost-xml
last release: Nov 30, 2022, *status*: 4 - Beta, *requires*: N/A

Plugin for pytest to generate boost xml reports

pytest-bootstrap
last release: Mar 04, 2022, *status*: N/A, *requires*: N/A

pytest-boto-mock
last release: Jan 20, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.2.0

Thin-wrapper around the mock package for easier use with pytest

pytest-bpdb
last release: Jan 19, 2015, *status*: 2 - Pre-Alpha, *requires*: N/A

A py.test plug-in to enable drop to bpdb debugger on test failure.

pytest-bq
last release: May 08, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=6.2

BigQuery fixtures and fixture factories for Pytest.

pytest-bravado
last release: Feb 15, 2022, *status*: N/A, *requires*: N/A

Pytest-bravado automatically generates from OpenAPI specification client fixtures.

pytest-breadcrumb
last release: Mar 09, 2026, *status*: 2 - Pre-Alpha, *requires*: pytest>=8.0; extra == "dev"

Self-healing test framework for Playwright. Your tests survive app changes.

pytest-breakword
last release: Aug 04, 2021, *status*: N/A, *requires*: pytest (>=6.2.4,<7.0.0)

Use breakword with pytest

pytest-breed-adapter
last release: Nov 07, 2018, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A simple plugin to connect with breed-server

pytest-briefcase
last release: Jun 14, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A pytest plugin for running tests on a Briefcase project.

pytest-brightest
last release: Jul 15, 2025, *status*: 3 - Alpha, *requires*: pytest>=8.4.1

Bright ideas for improving your pytest experience

pytest-broadcaster
last release: Mar 02, 2025, *status*: 3 - Alpha, *requires*: pytest

Pytest plugin to broadcast pytest output to various destinations

pytest-browser
last release: Dec 10, 2016, *status*: 3 - Alpha, *requires*: N/A

A pytest plugin for console based browser test selection just after the collection phase

pytest-browsermob-proxy
last release: Jun 11, 2013, *status*: 4 - Beta, *requires*: N/A

BrowserMob proxy plugin for py.test.

pytest_browserstack
last release: Jan 27, 2016, *status*: 4 - Beta, *requires*: N/A

Py.test plugin for BrowserStack

pytest-browserstack-local
last release: Feb 09, 2018, *status*: N/A, *requires*: N/A

``py.test`` plugin to run ``BrowserStackLocal`` in background.

pytest-budosystems
last release: May 07, 2023, *status*: 3 - Alpha, *requires*: pytest

Budo Systems is a martial arts school management system. This module is the Budo Systems Pytest Plugin.

pytest-bug
last release: Dec 30, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.0

Pytest plugin for marking tests as a bug

pytest-bugtong-tag
last release: Jan 16, 2022, *status*: N/A, *requires*: N/A

pytest-bugtong-tag is a plugin for pytest

pytest-bugzilla
last release: May 05, 2010, *status*: 4 - Beta, *requires*: N/A

py.test bugzilla integration plugin

pytest-bugzilla-notifier
last release: Jun 15, 2018, *status*: 4 - Beta, *requires*: pytest (>=2.9.2)

A plugin that allows you to execute create, update, and read information from BugZilla bugs

pytest-buildkite
last release: Jul 13, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Plugin for pytest that automatically publishes coverage and pytest report annotations to Buildkite.

pytest-builtin-types
last release: Nov 17, 2021, *status*: N/A, *requires*: pytest

pytest-bwrap
last release: Feb 25, 2024, *status*: 3 - Alpha, *requires*: N/A

Run your tests in Bubblewrap sandboxes

pytest-cache
last release: Jun 04, 2013, *status*: 3 - Alpha, *requires*: N/A

pytest plugin with mechanisms for caching across test runs

pytest-cache-assert
last release: Aug 14, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=6.0.0)

Cache assertion data to simplify regression testing of complex serializable data

pytest-cagoule
last release: Jan 01, 2020, *status*: 3 - Alpha, *requires*: N/A

Pytest plugin to only run tests affected by changes

pytest-cairo
last release: Apr 17, 2022, *status*: N/A, *requires*: pytest

Pytest support for cairo-lang and starknet

pytest-call-checker
last release: Oct 16, 2022, *status*: 4 - Beta, *requires*: pytest (>=7.1.3,<8.0.0)

Small pytest utility to easily create test doubles

pytest-camel-collect
last release: Aug 02, 2020, *status*: N/A, *requires*: pytest (>=2.9)

Enable CamelCase-aware pytest class collection

pytest-canonical-data
last release: May 08, 2020, *status*: 2 - Pre-Alpha, *requires*: pytest (>=3.5.0)

A plugin which allows to compare results with canonical results, based on previous runs

pytest-canvas
last release: Jul 22, 2025, *status*: N/A, *requires*: pytest<9,>=8.4

A minimal pytest plugin that streamlines testing for projects using the Canvas SDK.

pytest-capquery
last release: Apr 04, 2026, *status*: N/A, *requires*: pytest

A pytest fixture for high-precision SQL testing in SQLAlchemy.

pytest-caprng
last release: May 02, 2018, *status*: 4 - Beta, *requires*: N/A

A plugin that replays pRNG state on failure.

pytest-capsqlalchemy
last release: Mar 19, 2025, *status*: 4 - Beta, *requires*: N/A

Pytest plugin to allow capturing SQLAlchemy queries.

pytest-capture-deprecatedwarnings
last release: Apr 30, 2019, *status*: N/A, *requires*: N/A

pytest plugin to capture all deprecatedwarnings and put them in one file

pytest-capture-warnings
last release: May 03, 2022, *status*: N/A, *requires*: pytest

pytest plugin to capture all warnings and put them in one file of your choice

pytest-case
last release: Nov 25, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.3.3

A clean, modern, wrapper for pytest.mark.parametrize

pytest-case-provider
last release: Dec 15, 2025, *status*: 3 - Alpha, *requires*: pytest>=8

Advanced pytest parametrization plugin that generates test case instances from sync or async factories.

pytest-cases
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: pytest

Separate test code from test cases in pytest.

pytest-case-start-from
last release: Oct 28, 2025, *status*: 4 - Beta, *requires*: pytest>=6.0.0

A pytest plugin to start test execution from a specific test case

pytest-casewise-package-install
last release: Oct 31, 2025, *status*: 3 - Alpha, *requires*: pytest>=6.0.0

A pytest plugin for test case-level dynamic dependency management

pytest-cassandra
last release: Nov 04, 2017, *status*: 1 - Planning, *requires*: N/A

Cassandra CCM Test Fixtures for pytest

pytest-catchlog
last release: Jan 24, 2016, *status*: 4 - Beta, *requires*: pytest (>=2.6)

py.test plugin to catch log messages. This is a fork of pytest-capturelog.

pytest-catch-server
last release: Dec 12, 2019, *status*: 5 - Production/Stable, *requires*: N/A

Pytest plugin with server for catching HTTP requests.

pytest-cdist
last release: Jan 08, 2026, *status*: N/A, *requires*: pytest>=8

A pytest plugin to split your test suite into multiple parts

pytest-celery
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Pytest plugin for Celery

pytest-celery-py37
last release: May 23, 2025, *status*: 5 - Production/Stable, *requires*: N/A

Pytest plugin for Celery (compatible with python 3.7)

pytest-celery-utils
last release: Jan 28, 2026, *status*: N/A, *requires*: pytest>=9.0.1

Pytest plugin for inspecting Celery task queues in Redis during tests

pytest-cfg-fetcher
last release: Feb 26, 2024, *status*: N/A, *requires*: N/A

Pass config options to your unit tests.

pytest-chainmaker
last release: Oct 15, 2021, *status*: N/A, *requires*: N/A

pytest plugin for chainmaker

pytest-chalice
last release: Jul 01, 2020, *status*: 4 - Beta, *requires*: N/A

A set of py.test fixtures for AWS Chalice

pytest-change-assert
last release: Oct 19, 2022, *status*: N/A, *requires*: N/A

修改报错中文为英文

pytest-change-demo
last release: Mar 02, 2022, *status*: N/A, *requires*: pytest

turn . into √, turn F into x

pytest-change-report
last release: Sep 14, 2020, *status*: N/A, *requires*: pytest

turn . into √, turn F into x

pytest-change-xds
last release: Apr 16, 2022, *status*: N/A, *requires*: pytest

turn . into √, turn F into x

pytest-chdir
last release: Jan 28, 2020, *status*: N/A, *requires*: pytest (>=5.0.0,<6.0.0)

A pytest fixture for changing current working directory

pytest-check
last release: Mar 22, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

A pytest plugin that allows multiple failures per test.

pytest-checkdocs
last release: Dec 26, 2025, *status*: 5 - Production/Stable, *requires*: pytest!=8.1.*,>=6; extra == "test"

check the README when running tests

pytest-checkers
last release: Dec 27, 2025, *status*: N/A, *requires*: pytest>=9.0.2

Pytest Plugin for dry-run checks LSPs, Type Checkers, Linters, and Formatters during testing

pytest-checkipdb
last release: Dec 04, 2023, *status*: 5 - Production/Stable, *requires*: pytest >=2.9.2

plugin to check if there are ipdb debugs left

pytest-check-library
last release: Jul 17, 2022, *status*: N/A, *requires*: N/A

check your missing library

pytest-check-libs
last release: Jul 17, 2022, *status*: N/A, *requires*: N/A

check your missing library

pytest-check-links
last release: Jul 29, 2020, *status*: N/A, *requires*: pytest<9,>=7.0

Check links in files

pytest-checklist
last release: May 23, 2025, *status*: N/A, *requires*: N/A

Pytest plugin to track and report unit/function coverage.

pytest-check-mk
last release: Nov 19, 2015, *status*: 4 - Beta, *requires*: pytest

pytest plugin to test Check_MK checks

pytest-checkpoint
last release: Oct 04, 2025, *status*: N/A, *requires*: pytest>=8.0.0

Restore a checkpoint in pytest

pytest-ch-framework
last release: Apr 17, 2024, *status*: N/A, *requires*: pytest==8.0.1

My pytest framework

pytest-chic-report
last release: Nov 01, 2024, *status*: N/A, *requires*: pytest>=6.0

Simple pytest plugin for generating and sending report to messengers.

pytest-chinesereport
last release: Apr 16, 2025, *status*: 4 - Beta, *requires*: pytest>=3.5.0

pytest-choose
last release: Feb 04, 2024, *status*: N/A, *requires*: pytest >=7.0.0

Provide the pytest with the ability to collect use cases based on rules in text files

pytest-chronicle
last release: Dec 15, 2025, *status*: N/A, *requires*: pytest>=8.0; extra == "dev"

Reusable pytest results ingestion tooling with database export and CLI helpers.

pytest-chunks
last release: Jul 05, 2022, *status*: N/A, *requires*: pytest (>=6.0.0)

Run only a chunk of your test suite

pytest_cid
last release: Sep 01, 2023, *status*: 4 - Beta, *requires*: pytest >= 5.0, < 7.0

Compare data structures containing matching CIDs of different versions and encoding

pytest-circleci
last release: May 03, 2019, *status*: N/A, *requires*: N/A

py.test plugin for CircleCI

pytest-circleci-parallelized
last release: Oct 20, 2022, *status*: N/A, *requires*: N/A

Parallelize pytest across CircleCI workers.

pytest-circleci-parallelized-rjp
last release: Jun 21, 2022, *status*: N/A, *requires*: pytest

Parallelize pytest across CircleCI workers.

pytest-ckan
last release: Apr 28, 2020, *status*: 4 - Beta, *requires*: pytest

Backport of CKAN 2.9 pytest plugin and fixtures to CAKN 2.8

pytest-clab
last release: Mar 02, 2026, *status*: N/A, *requires*: pytest>=9.0.2

A pytest plugin for managing containerlab topologies in tests.

pytest-clang-tidy
last release: Mar 27, 2026, *status*: N/A, *requires*: pytest>=7.0

A pytest plugin that runs clang-tidy static analysis on C/C++ source files

pytest-clarity
last release: Jun 11, 2021, *status*: N/A, *requires*: N/A

A plugin providing an alternative, colourful diff output for failing assertions.

pytest-class-fixtures
last release: Nov 15, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.3.3

Class as PyTest fixtures (and BDD steps)

pytest-claude-agent-sdk
last release: Jan 19, 2026, *status*: 3 - Alpha, *requires*: pytest>=6.0

Use Claude Code in your pytests, or pytest your own Claude Code agents — or both

pytest-cldf
last release: Nov 07, 2022, *status*: N/A, *requires*: pytest (>=3.6)

Easy quality control for CLDF datasets using pytest

pytest-clean-database
last release: Mar 14, 2025, *status*: 3 - Alpha, *requires*: pytest<9,>=7.0

A pytest plugin that cleans your database up after every test.

pytest-cleanslate
last release: Apr 10, 2025, *status*: N/A, *requires*: pytest

Collects and executes pytest tests separately

pytest_cleanup
last release: Jan 28, 2020, *status*: N/A, *requires*: N/A

Automated, comprehensive and well-organised pytest test cases.

pytest-cleanuptotal
last release: Jul 22, 2025, *status*: 5 - Production/Stable, *requires*: N/A

A cleanup plugin for pytest

pytest-clerk
last release: Feb 04, 2026, *status*: N/A, *requires*: pytest<10.0.0,>=8.0.0

A set of pytest fixtures to help with integration testing with Clerk.

pytest-clerk-mock
last release: Apr 01, 2026, *status*: N/A, *requires*: N/A

A pytest plugin for mocking Clerk authentication

pytest-cli2-ansible
last release: Mar 05, 2025, *status*: N/A, *requires*: N/A

pytest-click
last release: Feb 11, 2022, *status*: 5 - Production/Stable, *requires*: pytest (>=5.0)

Pytest plugin for Click

pytest-cli-fixtures
last release: Jul 28, 2022, *status*: N/A, *requires*: pytest (~=7.0)

Automatically register fixtures for custom CLI arguments

pytest-clld
last release: Oct 23, 2024, *status*: N/A, *requires*: pytest>=3.9

pytest-cloud
last release: Oct 05, 2020, *status*: 6 - Mature, *requires*: N/A

Distributed tests planner plugin for pytest testing framework.

pytest-cloudflare-worker
last release: Mar 30, 2021, *status*: 4 - Beta, *requires*: pytest (>=6.0.0)

pytest plugin for testing cloudflare workers

pytest-cloudist
last release: Sep 02, 2022, *status*: 4 - Beta, *requires*: pytest (>=7.1.2,<8.0.0)

Distribute tests to cloud machines without fuss

pytest-cloudreport
last release: Apr 18, 2026, *status*: N/A, *requires*: pytest>=7.0

pytest plugin that uploads test results to pytest-cloudreport for analytics and flaky test detection

pytest-cmake
last release: Jan 03, 2026, *status*: N/A, *requires*: pytest<10,>=4

Provide CMake module for Pytest

pytest-cmake-presets
last release: Dec 26, 2022, *status*: N/A, *requires*: pytest (>=7.2.0,<8.0.0)

Execute CMake Presets via pytest

pytest-cmdline-add-args
last release: Sep 01, 2024, *status*: N/A, *requires*: N/A

Pytest plugin for custom argument handling and Allure reporting. This plugin allows you to add arguments before running a test.

pytest-cobra
last release: Jun 29, 2019, *status*: 3 - Alpha, *requires*: pytest (<4.0.0,>=3.7.1)

PyTest plugin for testing Smart Contracts for Ethereum blockchain.

pytest-cocotb
last release: Nov 09, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin that enables using pytest as the regression manager for running cocotb tests.

pytest-cocotb-cov
last release: Nov 09, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin for measuring HDL coverage.

pytest-cocotb-fusesoc
last release: Jan 07, 2026, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin to integrate FuseSoC with Cocotb.

pytest-cocotb-pyuvm
last release: Nov 09, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin that enables using pytest as the regression manager for running pyuvm tests.

pytest-codeblock
last release: Apr 07, 2026, *status*: 4 - Beta, *requires*: pytest

Pytest plugin to collect and test code blocks in reStructuredText and Markdown files.

pytest_codeblocks
last release: Sep 17, 2023, *status*: 5 - Production/Stable, *requires*: pytest >= 7.0.0

Test code blocks in your READMEs

pytest-codecarbon
last release: Jun 15, 2022, *status*: N/A, *requires*: pytest

Pytest plugin for measuring carbon emissions

pytest-codecheckers
last release: Feb 13, 2010, *status*: N/A, *requires*: N/A

pytest plugin to add source code sanity checks (pep8 and friends)

pytest-codecov
last release: Mar 25, 2025, *status*: 4 - Beta, *requires*: pytest>=4.6.0

Pytest plugin for uploading pytest-cov results to codecov.io

pytest-codegen
last release: Aug 23, 2020, *status*: 2 - Pre-Alpha, *requires*: N/A

Automatically create pytest test signatures

pytest-codeowners
last release: Mar 30, 2022, *status*: 4 - Beta, *requires*: pytest (>=6.0.0)

Pytest plugin for selecting tests by GitHub CODEOWNERS.

pytest-codestyle
last release: Mar 23, 2020, *status*: 3 - Alpha, *requires*: N/A

pytest plugin to run pycodestyle

pytest-codspeed
last release: Apr 14, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=3.8

Pytest plugin to create CodSpeed benchmarks

pytest-collect-appoint-info
last release: Aug 03, 2023, *status*: N/A, *requires*: pytest

set your encoding

pytest-collect-formatter
last release: Mar 29, 2021, *status*: 5 - Production/Stable, *requires*: N/A

Formatter for pytest collect output

pytest-collect-formatter2
last release: May 31, 2021, *status*: 5 - Production/Stable, *requires*: N/A

Formatter for pytest collect output

pytest-collect-interface-info-plugin
last release: Sep 25, 2023, *status*: 4 - Beta, *requires*: N/A

Get executed interface information in pytest interface automation framework

pytest-collect-markers
last release: Jan 24, 2026, *status*: N/A, *requires*: pytest>=7.0.0

A pytest plugin to collect and output test markers to JSON

pytest-collector
last release: Aug 02, 2022, *status*: N/A, *requires*: pytest (>=7.0,<8.0)

Python package for collecting pytest.

pytest-collect-pytest-interinfo
last release: Sep 26, 2023, *status*: 4 - Beta, *requires*: N/A

A simple plugin to use with pytest

pytest-collect-requirements
last release: Feb 07, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.1

A pytest plugin to collect test requirements from requirements marker.

pytest-colordots
last release: Oct 06, 2017, *status*: 5 - Production/Stable, *requires*: N/A

Colorizes the progress indicators

pytest-comfyui
last release: Jan 09, 2026, *status*: N/A, *requires*: N/A

Integration testing framework for ComfyUI nodes and workflows.

pytest-commander
last release: Aug 17, 2021, *status*: N/A, *requires*: pytest (<7.0.0,>=6.2.4)

An interactive GUI test runner for PyTest

pytest-common-subject
last release: Oct 22, 2025, *status*: N/A, *requires*: pytest<9,>=3.6

pytest framework for testing different aspects of a common method

pytest-compare
last release: Jun 22, 2023, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for comparing call arguments.

pytest-concurrency
last release: Apr 02, 2026, *status*: N/A, *requires*: pytest>=7.0

A pytest plugin for parallel test execution with configurable concurrency

pytest-concurrent
last release: Jan 12, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.1.1)

Concurrently execute test cases with multithread, multiprocess and gevent

pytest-conductor
last release: Jul 30, 2025, *status*: N/A, *requires*: pytest<8.4; python_version == "3.8"

Pytest plugin for coordinating the order in which marked tests run.

pytest-config
last release: Nov 07, 2014, *status*: 5 - Production/Stable, *requires*: N/A

Base configurations and utilities for developing your Python project test suite with pytest.

pytest-confluence-report
last release: Apr 17, 2022, *status*: N/A, *requires*: N/A

Package stands for pytest plugin to upload results into Confluence page.

pytest-console-scripts
last release: May 31, 2023, *status*: 4 - Beta, *requires*: pytest (>=4.0.0)

Pytest plugin for testing console scripts

pytest-consul
last release: Nov 24, 2018, *status*: 3 - Alpha, *requires*: pytest

pytest plugin with fixtures for testing consul aware apps

pytest-container
last release: Jun 30, 2025, *status*: 4 - Beta, *requires*: pytest>=3.10

Pytest fixtures for writing container based tests

pytest-contextfixture
last release: Mar 12, 2013, *status*: 4 - Beta, *requires*: N/A

Define pytest fixtures as context managers.

pytest-contexts
last release: May 19, 2021, *status*: 4 - Beta, *requires*: N/A

A plugin to run tests written with the Contexts framework using pytest

pytest-continuous
last release: Apr 23, 2024, *status*: N/A, *requires*: N/A

A pytest plugin to run tests continuously until failure or interruption.

pytest-cookies
last release: Mar 22, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=3.9.0)

The pytest plugin for your Cookiecutter templates. 🍪

pytest-copie
last release: Sep 29, 2025, *status*: 3 - Alpha, *requires*: pytest

The pytest plugin for your copier templates 📒

pytest-copier
last release: Dec 11, 2023, *status*: 4 - Beta, *requires*: pytest>=7.3.2

A pytest plugin to help testing Copier templates

pytest-couchdbkit
last release: Apr 17, 2012, *status*: N/A, *requires*: N/A

py.test extension for per-test couchdb databases using couchdbkit

pytest-count
last release: Jan 12, 2018, *status*: 4 - Beta, *requires*: N/A

count erros and send email

pytest-cov
last release: Mar 21, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7

Pytest plugin for measuring coverage.

pytest-cov-container
last release: Mar 23, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Pytest plugin to collect code coverage from applications running inside Docker containers

pytest-cover
last release: Aug 01, 2015, *status*: 5 - Production/Stable, *requires*: N/A

Pytest plugin for measuring coverage. Forked from `pytest-cov`.

pytest-coverage
last release: Jun 17, 2015, *status*: N/A, *requires*: N/A

pytest-coverage-context
last release: Jun 28, 2023, *status*: 4 - Beta, *requires*: N/A

Coverage dynamic context support for PyTest, including sub-processes

pytest-coverage-gate
last release: Apr 14, 2026, *status*: 5 - Production/Stable, *requires*: N/A

A pre-commit hook that enforces a coverage quality gate using coverage.xml and a baseline file

pytest-coverage-impact
last release: Jan 16, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

Sensoria: High-fidelity coverage impact analysis for Python.

pytest-coveragemarkers
last release: May 15, 2025, *status*: N/A, *requires*: pytest<8.0.0,>=7.1.2

Using pytest markers to track functional coverage and filtering of tests

pytest-cov-exclude
last release: Apr 29, 2016, *status*: 4 - Beta, *requires*: pytest (>=2.8.0,<2.9.0); extra == 'dev'

Pytest plugin for excluding tests based on coverage data

pytest_covid
last release: Jun 24, 2020, *status*: N/A, *requires*: N/A

Too many faillure, less tests.

pytest-cpp
last release: Sep 18, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Use pytest's runner to discover and execute C++ tests

pytest-cppcheck
last release: Mar 26, 2026, *status*: N/A, *requires*: pytest>=7.0

A pytest plugin that runs cppcheck static analysis on C/C++ source files

pytest-cpplint
last release: Mar 26, 2026, *status*: N/A, *requires*: pytest>=7.0

A pytest plugin that runs cpplint style checking on C/C++ source files

pytest-cqase
last release: Aug 22, 2022, *status*: N/A, *requires*: pytest (>=7.1.2,<8.0.0)

Custom qase pytest plugin

pytest-cram
last release: Aug 08, 2020, *status*: N/A, *requires*: N/A

Run cram tests with pytest.

pytest-crap
last release: Dec 02, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0

pytest plugin that calculates CRAP scores to guide test writing

pytest-crate
last release: May 28, 2019, *status*: 3 - Alpha, *requires*: pytest (>=4.0)

Manages CrateDB instances during your integration tests

pytest-cratedb
last release: Jan 05, 2026, *status*: 4 - Beta, *requires*: pytest<10

Manage CrateDB instances for integration tests

pytest-cratedb-reporter
last release: Mar 11, 2025, *status*: N/A, *requires*: pytest>=6.0.0

A pytest plugin for reporting test results to CrateDB

pytest-crayons
last release: Oct 14, 2025, *status*: 5 - Production/Stable, *requires*: pytest

A pytest plugin for colorful print statements

pytest-cream
last release: Oct 26, 2025, *status*: N/A, *requires*: pytest

The cream of test execution - smooth pytest workflows with intelligent orchestration

pytest-create
last release: Feb 15, 2023, *status*: 1 - Planning, *requires*: N/A

pytest-create

pytest-cricri
last release: Jan 27, 2018, *status*: N/A, *requires*: pytest

A Cricri plugin for pytest.

pytest-crontab
last release: Dec 09, 2019, *status*: N/A, *requires*: N/A

add crontab task in crontab

pytest-csv
last release: Apr 22, 2021, *status*: N/A, *requires*: pytest (>=6.0)

CSV output for pytest.

pytest-csv-params
last release: May 29, 2025, *status*: 5 - Production/Stable, *requires*: pytest<9,>=8.3

Pytest plugin for Test Case Parametrization with CSV files

pytest-culprit
last release: May 15, 2025, *status*: N/A, *requires*: N/A

Find the last Git commit where a pytest test started failing

pytest-curio
last release: Oct 06, 2024, *status*: N/A, *requires*: pytest

Pytest support for curio.

pytest-curl-report
last release: Dec 11, 2016, *status*: 4 - Beta, *requires*: N/A

pytest plugin to generate curl command line report

pytest-custom-concurrency
last release: Feb 08, 2021, *status*: N/A, *requires*: N/A

Custom grouping concurrence for pytest

pytest-custom-exit-code
last release: Aug 07, 2019, *status*: 4 - Beta, *requires*: pytest (>=4.0.2)

Exit pytest test session with custom exit code in different scenarios

pytest-custom-nodeid
last release: Mar 07, 2021, *status*: N/A, *requires*: N/A

Custom grouping for pytest-xdist, rename test cases name and test cases nodeid, support allure report

pytest-custom-outputs
last release: Jul 10, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin that allows users to create and use custom outputs instead of the standard Pass and Fail. Also allows users to retrieve test results in fixtures.

pytest-custom-report
last release: Jan 30, 2019, *status*: N/A, *requires*: pytest

Configure the symbols displayed for test outcomes

pytest-custom-scheduling
last release: Mar 01, 2021, *status*: N/A, *requires*: N/A

Custom grouping for pytest-xdist, rename test cases name and test cases nodeid, support allure report

pytest-custom-timeout
last release: Jan 08, 2025, *status*: 4 - Beta, *requires*: pytest>=8.0.0

Use custom logic when a test times out. Based on pytest-timeout.

pytest-cython
last release: Mar 11, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8

A plugin for testing Cython extension modules.

pytest-cython-collect
last release: Jun 17, 2022, *status*: N/A, *requires*: pytest

pytest-dag
last release: Apr 06, 2026, *status*: N/A, *requires*: pytest>=7.0

A pytest plugin that enforces test execution order via a dependency DAG

pytest-darker
last release: Feb 25, 2024, *status*: N/A, *requires*: pytest <7,>=6.0.1

A pytest plugin for checking of modified code using Darker

pytest-dash
last release: Mar 18, 2019, *status*: N/A, *requires*: N/A

pytest fixtures to run dash applications.

pytest-dashboard
last release: Jun 02, 2025, *status*: N/A, *requires*: pytest<8.0.0,>=7.4.3

pytest-data
last release: Nov 01, 2016, *status*: 5 - Production/Stable, *requires*: N/A

Useful functions for managing data for pytest fixtures

pytest-databases
last release: Mar 10, 2026, *status*: 4 - Beta, *requires*: pytest

Reusable database fixtures for any and all databases.

pytest-databricks
last release: Jul 29, 2020, *status*: N/A, *requires*: pytest

Pytest plugin for remote Databricks notebooks testing

pytest-datadir
last release: Jul 30, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.0

pytest plugin for test data directories and files

pytest-datadir-mgr
last release: Apr 06, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=7.1)

Manager for test data: downloads, artifact caching, and a tmpdir context.

pytest-datadir-ng
last release: Dec 25, 2019, *status*: 5 - Production/Stable, *requires*: pytest

Fixtures for pytest allowing test functions/methods to easily retrieve test resources from the local filesystem.

pytest-datadir-nng
last release: Nov 09, 2022, *status*: 5 - Production/Stable, *requires*: pytest (>=7.0.0,<8.0.0)

Fixtures for pytest allowing test functions/methods to easily retrieve test resources from the local filesystem.

pytest-data-extractor
last release: Jul 19, 2022, *status*: N/A, *requires*: pytest (>=7.0.1)

A pytest plugin to extract relevant metadata about tests into an external file (currently only json support)

pytest-data-file
last release: Dec 04, 2019, *status*: N/A, *requires*: N/A

Fixture “data” and “case_data” for test from yaml file

pytest-datafiles
last release: Jan 04, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6.2.0

py.test plugin to create a 'tmp_path' containing predefined files/directories.

pytest-datafixtures
last release: May 15, 2025, *status*: 5 - Production/Stable, *requires*: N/A

Data fixtures for pytest made simple.

pytest-data-from-files
last release: Oct 13, 2021, *status*: 4 - Beta, *requires*: pytest

pytest plugin to provide data from files loaded automatically

pytest-dataguard
last release: Oct 08, 2025, *status*: N/A, *requires*: pytest>=8.4.2

Data validation and integrity testing for your datasets using pytest.

pytest-data-loader
last release: Apr 14, 2026, *status*: 4 - Beta, *requires*: pytest<10,>=7.0.0

Pytest plugin for loading test data for data-driven testing (DDT)

pytest-dataplugin
last release: Sep 16, 2017, *status*: 1 - Planning, *requires*: N/A

A pytest plugin for managing an archive of test data.

pytest-datarecorder
last release: Jul 31, 2024, *status*: 5 - Production/Stable, *requires*: pytest

A py.test plugin recording and comparing test output.

pytest-dataset
last release: Sep 01, 2023, *status*: 5 - Production/Stable, *requires*: N/A

Plugin for loading different datasets for pytest by prefix from json or yaml files

pytest-data-suites
last release: Apr 06, 2024, *status*: N/A, *requires*: pytest<9.0,>=6.0

Class-based pytest parametrization

pytest-datatest
last release: Oct 15, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.3)

A pytest plugin for test driven data-wrangling (this is the development version of datatest's pytest integration).

pytest-db
last release: Nov 11, 2025, *status*: N/A, *requires*: pytest

Session scope fixture "db" for mysql query or change

pytest-dbfixtures
last release: Dec 07, 2016, *status*: 4 - Beta, *requires*: N/A

Databases fixtures plugin for py.test.

pytest-db-plugin
last release: Nov 27, 2021, *status*: N/A, *requires*: pytest (>=5.0)

pytest-dbt
last release: Jun 08, 2023, *status*: 2 - Pre-Alpha, *requires*: pytest (>=7.0.0,<8.0.0)

Unit test dbt models with standard python tooling

pytest-dbt-adapter
last release: Nov 24, 2021, *status*: N/A, *requires*: pytest (<7,>=6)

A pytest plugin for testing dbt adapter plugins

pytest-dbt-conventions
last release: Mar 02, 2022, *status*: N/A, *requires*: pytest (>=6.2.5,<7.0.0)

A pytest plugin for linting a dbt project's conventions

pytest-dbt-core
last release: Jun 04, 2024, *status*: N/A, *requires*: pytest>=6.2.5; extra == "test"

Pytest extension for dbt.

pytest-dbt-duckdb
last release: Mar 06, 2026, *status*: 4 - Beta, *requires*: pytest>=8.3.4

Fearless testing for dbt models, powered by DuckDB.

pytest-dbt-postgres
last release: Sep 03, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.3.2

Pytest tooling to unittest DBT & Postgres models

pytest-dbus-notification
last release: Mar 05, 2014, *status*: 5 - Production/Stable, *requires*: N/A

D-BUS notifications for pytest results.

pytest-dbx
last release: Nov 29, 2022, *status*: N/A, *requires*: pytest (>=7.1.3,<8.0.0)

Pytest plugin to run unit tests for dbx (Databricks CLI extensions) related code

pytest-dc
last release: Aug 16, 2023, *status*: 5 - Production/Stable, *requires*: pytest >=3.3

Manages Docker containers during your integration tests

pytest-deadfixtures
last release: Jan 15, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

A simple plugin to list unused fixtures in pytest

pytest-deduplicate
last release: Aug 12, 2023, *status*: 4 - Beta, *requires*: pytest

Identifies duplicate unit tests

pytest-deepassert
last release: Nov 04, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

A pytest plugin for enhanced assertion reporting with detailed diffs

pytest-deepcov
last release: Mar 30, 2021, *status*: N/A, *requires*: N/A

deepcov

pytest_defer
last release: Nov 13, 2024, *status*: N/A, *requires*: pytest>=8.3

A 'defer' fixture for pytest

pytest-delta
last release: Feb 12, 2026, *status*: 4 - Beta, *requires*: pytest<10.0.0,>=9.0.2

Run only tests impacted by your code changes (delta-based selection) for pytest.

pytest-demo-plugin
last release: May 15, 2021, *status*: N/A, *requires*: N/A

pytest示例插件

pytest-dependency
last release: Feb 15, 2026, *status*: 4 - Beta, *requires*: N/A

Manage dependencies of tests

pytest-depends
last release: Apr 05, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=3)

Tests that depend on other tests

pytest-depends-on
last release: Mar 28, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.1

A Python package for managing test dependencies in pytest.

pytest-depper
last release: Oct 23, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Smart test selection based on AST-level code dependency analysis

pytest-deprecate
last release: Jul 01, 2019, *status*: N/A, *requires*: N/A

Mark tests as testing a deprecated feature with a warning note.

pytest-deprecator
last release: Dec 02, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A simple plugin to use with pytest

pytest-describe
last release: Dec 12, 2025, *status*: 5 - Production/Stable, *requires*: pytest<10,>=6

Describe-style plugin for pytest

pytest-describe-beautifully
last release: Jan 28, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

Beautiful terminal and HTML output for pytest-describe.

pytest-describe-it
last release: Jul 19, 2019, *status*: 4 - Beta, *requires*: pytest

plugin for rich text descriptions

pytest-deselect-if
last release: Dec 26, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin to deselect pytests tests rather than using skipif

pytest-devpi-server
last release: Oct 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

DevPI server fixture for py.test

pytest-devtools
last release: Feb 10, 2026, *status*: N/A, *requires*: pytest>=9.0.2

Pytest plugin providing debug fixtures, ANSI-stripped capsys, whitespace-visible assertions, and terminal column management.

pytest-dfm
last release: Nov 23, 2025, *status*: N/A, *requires*: pytest

pytest-dfm provides a pytest integration for DV Flow Manager, a build system for silicon design

pytest-dhos
last release: Sep 07, 2022, *status*: N/A, *requires*: N/A

Common fixtures for pytest in DHOS services and libraries

pytest-diamond
last release: Aug 31, 2015, *status*: 4 - Beta, *requires*: N/A

pytest plugin for diamond

pytest-dicom
last release: Dec 19, 2018, *status*: 3 - Alpha, *requires*: pytest

pytest plugin to provide DICOM fixtures

pytest-dictsdiff
last release: Jul 26, 2019, *status*: N/A, *requires*: N/A

pytest-diff
last release: Mar 30, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A simple plugin to use with pytest

pytest-diff-selector
last release: Feb 24, 2022, *status*: 4 - Beta, *requires*: pytest (>=6.2.2) ; extra == 'all'

Get tests affected by code changes (using git)

pytest-difftest
last release: Feb 23, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Blazingly fast test selection for pytest - only run tests affected by your changes (Rust-powered)

pytest-difido
last release: Oct 23, 2022, *status*: 4 - Beta, *requires*: pytest (>=4.0.0)

PyTest plugin for generating Difido reports

pytest-directives
last release: Aug 11, 2025, *status*: 3 - Alpha, *requires*: pytest

Control your tests flow

pytest-dir-equal
last release: Dec 11, 2023, *status*: 4 - Beta, *requires*: pytest>=7.3.2

pytest-dir-equals is a pytest plugin providing helpers to assert directories equality allowing golden testing

pytest-dirty
last release: Jun 08, 2025, *status*: 3 - Alpha, *requires*: pytest>=8.2; extra == "dev"

Static import analysis for thrifty testing.

pytest-disable
last release: Sep 10, 2015, *status*: 4 - Beta, *requires*: N/A

pytest plugin to disable a test and skip it from testrun

pytest-disable-plugin
last release: Feb 28, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Disable plugins per test

pytest-discord
last release: May 11, 2024, *status*: 4 - Beta, *requires*: pytest!=6.0.0,<9,>=3.3.2

A pytest plugin to notify test results to a Discord channel.

pytest-discover
last release: Mar 26, 2024, *status*: N/A, *requires*: pytest

Pytest plugin to record discovered tests in a file

pytest-ditto
last release: Mar 22, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=3.5.0

Snapshot testing pytest plugin with minimal ceremony and flexible recorders.

pytest-ditto-pandas
last release: Mar 22, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=3.5.0

pytest-ditto plugin for pandas DataFrame snapshots.

pytest-ditto-pyarrow
last release: Mar 22, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=3.5.0

pytest-ditto plugin for pyarrow table snapshots.

pytest-django
last release: Feb 14, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

A Django plugin for pytest.

pytest-django-ahead
last release: Oct 27, 2016, *status*: 5 - Production/Stable, *requires*: pytest (>=2.9)

A Django plugin for pytest.

pytest-djangoapp
last release: Dec 13, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Nice pytest plugin to help you with Django pluggable application testing.

pytest-django-asyncio
last release: Mar 26, 2026, *status*: 4 - Beta, *requires*: pytest>=8.0

Temporary pytest plugin backport for async Django DB fixture handling.

pytest-django-cache-xdist
last release: May 12, 2020, *status*: 4 - Beta, *requires*: N/A

A djangocachexdist plugin for pytest

pytest-django-casperjs
last release: Mar 15, 2015, *status*: 2 - Pre-Alpha, *requires*: N/A

Integrate CasperJS with your django tests as a pytest fixture.

pytest-django-class
last release: Aug 08, 2023, *status*: 4 - Beta, *requires*: N/A

A pytest plugin for running django in class-scoped fixtures

pytest-django-docker-pg
last release: Jun 13, 2024, *status*: 5 - Production/Stable, *requires*: pytest<9.0.0,>=7.0.0

pytest-django-dotenv
last release: Nov 26, 2019, *status*: 4 - Beta, *requires*: pytest (>=2.6.0)

Pytest plugin used to setup environment variables with django-dotenv

pytest-django-factories
last release: Nov 12, 2020, *status*: 4 - Beta, *requires*: N/A

Factories for your Django models that can be used as Pytest fixtures.

pytest-django-filefield
last release: May 09, 2022, *status*: 5 - Production/Stable, *requires*: pytest >= 5.2

Replaces FileField.storage with something you can patch globally.

pytest-django-gcir
last release: Mar 06, 2018, *status*: 5 - Production/Stable, *requires*: N/A

A Django plugin for pytest.

pytest-django-haystack
last release: Sep 03, 2017, *status*: 5 - Production/Stable, *requires*: pytest (>=2.3.4)

Cleanup your Haystack indexes between tests

pytest-django-ifactory
last release: Apr 30, 2025, *status*: 5 - Production/Stable, *requires*: N/A

A model instance factory for pytest-django

pytest-django-lite
last release: Jan 30, 2014, *status*: N/A, *requires*: N/A

The bare minimum to integrate py.test with Django.

pytest-django-liveserver-ssl
last release: Jan 09, 2025, *status*: 3 - Alpha, *requires*: N/A

pytest-django-model
last release: Feb 14, 2019, *status*: 4 - Beta, *requires*: N/A

A Simple Way to Test your Django Models

pytest-django-ordering
last release: Jul 25, 2019, *status*: 5 - Production/Stable, *requires*: pytest (>=2.3.0)

A pytest plugin for preserving the order in which Django runs tests.

pytest-django-queries
last release: Mar 01, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.2.0

Generate performance reports from your django database performance tests.

pytest-djangorestframework
last release: Aug 11, 2019, *status*: 4 - Beta, *requires*: N/A

A djangorestframework plugin for pytest

pytest-django-rq
last release: Apr 13, 2020, *status*: 4 - Beta, *requires*: N/A

A pytest plugin to help writing unit test for django-rq

pytest-django-sqlcounts
last release: Jun 16, 2015, *status*: 4 - Beta, *requires*: N/A

py.test plugin for reporting the number of SQLs executed per django testcase.

pytest-django-testing-postgresql
last release: Jan 31, 2022, *status*: 4 - Beta, *requires*: N/A

Use a temporary PostgreSQL database with pytest-django

pytest-doc
last release: Jun 28, 2015, *status*: 5 - Production/Stable, *requires*: N/A

A documentation plugin for py.test.

pytest-docfiles
last release: Dec 22, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.7.0)

pytest plugin to test codeblocks in your documentation.

pytest-docgen
last release: Apr 17, 2020, *status*: N/A, *requires*: N/A

An RST Documentation Generator for pytest-based test suites

pytest-docker
last release: Nov 12, 2025, *status*: N/A, *requires*: pytest<10.0,>=4.0

Simple pytest fixtures for Docker and Docker Compose based tests

pytest-docker-apache-fixtures
last release: Aug 12, 2024, *status*: 4 - Beta, *requires*: pytest

Pytest fixtures for testing with apache2 (httpd).

pytest-docker-butla
last release: Jun 16, 2019, *status*: 3 - Alpha, *requires*: N/A

pytest-dockerc
last release: Oct 09, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=3.0)

Run, manage and stop Docker Compose project from Docker API

pytest-docker-compose
last release: Jan 26, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=3.3)

Manages Docker containers during your integration tests

pytest-docker-compose-v2
last release: Dec 17, 2025, *status*: 4 - Beta, *requires*: pytest<10,>=7.2.2

Manages Docker containers during your integration tests

pytest-docker-db
last release: Mar 20, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=3.1.1)

A plugin to use docker databases for pytests

pytest-docker-fixtures
last release: Dec 01, 2025, *status*: 3 - Alpha, *requires*: pytest

pytest docker fixtures

pytest-docker-git-fixtures
last release: Aug 12, 2024, *status*: 4 - Beta, *requires*: pytest

Pytest fixtures for testing with git scm.

pytest-docker-haproxy-fixtures
last release: Aug 12, 2024, *status*: 4 - Beta, *requires*: pytest

Pytest fixtures for testing with haproxy.

pytest-docker-pexpect
last release: Jan 14, 2019, *status*: N/A, *requires*: pytest

pytest plugin for writing functional tests with pexpect and docker

pytest-docker-postgresql
last release: Sep 24, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A simple plugin to use with pytest

pytest-docker-py
last release: Nov 27, 2018, *status*: N/A, *requires*: pytest (==4.0.0)

Easy to use, simple to extend, pytest plugin that minimally leverages docker-py.

pytest-docker-registry-fixtures
last release: Aug 12, 2024, *status*: 4 - Beta, *requires*: pytest

Pytest fixtures for testing with docker registries.

pytest-docker-service
last release: Jan 03, 2024, *status*: 3 - Alpha, *requires*: pytest (>=7.1.3)

pytest plugin to start docker container

pytest-docker-squid-fixtures
last release: Aug 12, 2024, *status*: 4 - Beta, *requires*: pytest

Pytest fixtures for testing with squid.

pytest-docker-tools
last release: Mar 16, 2025, *status*: 4 - Beta, *requires*: pytest>=6.0.1

Docker integration tests for pytest

pytest-docs
last release: Nov 11, 2018, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Documentation tool for pytest

pytest-docstyle
last release: Mar 23, 2020, *status*: 3 - Alpha, *requires*: N/A

pytest plugin to run pydocstyle

pytest-doctest-custom
last release: Jul 25, 2016, *status*: 4 - Beta, *requires*: N/A

A py.test plugin for customizing string representations of doctest results.

pytest-doctest-ellipsis-markers
last release: Jan 12, 2018, *status*: 4 - Beta, *requires*: N/A

Setup additional values for ELLIPSIS_MARKER for doctests

pytest-doctest-import
last release: Nov 13, 2018, *status*: 4 - Beta, *requires*: pytest (>=3.3.0)

A simple pytest plugin to import names and add them to the doctest namespace.

pytest-doctest-mkdocstrings
last release: Mar 02, 2024, *status*: N/A, *requires*: pytest

Run pytest –doctest-modules with markdown docstrings in code blocks (```)

pytest-doctest-only
last release: Jul 30, 2025, *status*: 4 - Beta, *requires*: pytest>=8.3.0

A plugin to run only doctest

pytest-doctestplus
last release: Jan 26, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0

Pytest plugin with advanced doctest features.

pytest-doctor
last release: Mar 22, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

A CLI tool that diagnoses weak or broken pytest suites and provides a 0-100 health score with actionable recommendations

pytest-documentary
last release: Jul 11, 2024, *status*: N/A, *requires*: pytest

A simple pytest plugin to generate test documentation

pytest-dogu-report
last release: Jul 07, 2023, *status*: N/A, *requires*: N/A

pytest plugin for dogu report

pytest-dogu-sdk
last release: Dec 14, 2023, *status*: N/A, *requires*: N/A

pytest plugin for the Dogu

pytest-dolphin
last release: Nov 30, 2016, *status*: 4 - Beta, *requires*: pytest (==3.0.4)

Some extra stuff that we use ininternally

pytest-donde
last release: Oct 01, 2023, *status*: 4 - Beta, *requires*: pytest >=7.3.1

record pytest session characteristics per test item (coverage and duration) into a persistent file and use them in your own plugin or script.

pytest-doorstop
last release: Jun 09, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A pytest plugin for adding test results into doorstop items.

pytest-dotenv
last release: Jun 16, 2020, *status*: 4 - Beta, *requires*: pytest (>=5.0.0)

A py.test plugin that parses environment files before running tests

pytest-dotenv-modern
last release: Sep 27, 2025, *status*: 4 - Beta, *requires*: pytest>=6.0.0

A modern pytest plugin that loads environment variables from dotenv files

pytest-dot-only-pkcopley
last release: Oct 27, 2023, *status*: N/A, *requires*: N/A

A Pytest marker for only running a single test

pytest-dparam
last release: Aug 27, 2024, *status*: 6 - Mature, *requires*: pytest

A more readable alternative to @pytest.mark.parametrize.

pytest-dpg
last release: Aug 13, 2024, *status*: N/A, *requires*: N/A

pytest-dpg is a pytest plugin for testing Dear PyGui (DPG) applications

pytest-draw
last release: Mar 21, 2023, *status*: 3 - Alpha, *requires*: pytest

Pytest plugin for randomly selecting a specific number of tests

pytest-drf
last release: Jul 12, 2022, *status*: 5 - Production/Stable, *requires*: pytest (>=3.7)

A Django REST framework plugin for pytest.

pytest-drift
last release: Apr 18, 2026, *status*: N/A, *requires*: pytest>=7.0

Pytest plugin for regression testing via branch comparison

pytest-drill-sergeant
last release: Feb 20, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin that enforces test quality standards through automatic marker detection and AAA structure validation

pytest-drivings
last release: Jan 13, 2021, *status*: N/A, *requires*: N/A

Tool to allow webdriver automation to be ran locally or remotely

pytest-drop-dup-tests
last release: Mar 04, 2024, *status*: 5 - Production/Stable, *requires*: pytest >=7

A Pytest plugin to drop duplicated tests during collection

pytest-dryci
last release: Sep 27, 2024, *status*: 4 - Beta, *requires*: N/A

Test caching plugin for pytest

pytest-dryrun
last release: Jan 19, 2025, *status*: 5 - Production/Stable, *requires*: pytest<9,>=7.40

A Pytest plugin to ignore tests during collection without reporting them in the test summary.

pytest-dsl
last release: Apr 01, 2026, *status*: N/A, *requires*: pytest>=7.0.0

A DSL testing framework based on pytest

pytest-dsl-ssh
last release: Jul 25, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

SSH/SFTP关键字插件，为pytest-dsl提供SSH和SFTP操作能力

pytest-dsl-ui
last release: Apr 13, 2026, *status*: N/A, *requires*: pytest>=7.0.0; extra == "dev"

Playwright-based UI automation keywords for pytest-dsl framework

pytest-dummynet
last release: Dec 15, 2021, *status*: 5 - Production/Stable, *requires*: pytest

A py.test plugin providing access to a dummynet.

pytest-dump2json
last release: Jun 29, 2015, *status*: N/A, *requires*: N/A

A pytest plugin for dumping test results to json.

pytest-duration-insights
last release: Jul 15, 2024, *status*: N/A, *requires*: N/A

pytest-durations
last release: Mar 13, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=4.6

Pytest plugin reporting fixtures and test functions execution time.

pytest-dynamic-parameterize
last release: Feb 07, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.1

A pytest plugin to dynamically parameterize tests based on external data sources.

pytest-dynamicrerun
last release: Aug 15, 2020, *status*: 4 - Beta, *requires*: N/A

A pytest plugin to rerun tests dynamically based off of test outcome and output.

pytest-dynamodb
last release: Mar 13, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.4.0

DynamoDB fixtures for pytest

pytest-easy-addoption
last release: Jan 22, 2020, *status*: N/A, *requires*: N/A

pytest-easy-addoption: Easy way to work with pytest addoption

pytest-easyMPI
last release: Oct 21, 2020, *status*: N/A, *requires*: N/A

Package that supports mpi tests in pytest

pytest-easyread
last release: Nov 17, 2017, *status*: N/A, *requires*: N/A

pytest plugin that makes terminal printouts of the reports easier to read

pytest-easy-server
last release: May 01, 2021, *status*: 4 - Beta, *requires*: pytest (<5.0.0,>=4.3.1) ; python_version < "3.5"

Pytest plugin for easy testing against servers

pytest-ebics-sandbox
last release: Aug 15, 2022, *status*: N/A, *requires*: N/A

A pytest plugin for testing against an EBICS sandbox server. Requires docker.

pytest-ec2
last release: Oct 22, 2019, *status*: 3 - Alpha, *requires*: N/A

Pytest execution on EC2 instance

pytest-echo
last release: Apr 27, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=8.3.3

pytest plugin that allows to dump environment variables, package version and generic attributes

pytest-edit
last release: Nov 17, 2024, *status*: N/A, *requires*: pytest

Edit the source code of a failed test with `pytest –edit`.

pytest-egg
last release: Apr 17, 2026, *status*: N/A, *requires*: pytest>=7.0

A pytest plugin that prints egg ascii art at the end of a test run.

pytest-ekstazi
last release: Sep 10, 2022, *status*: N/A, *requires*: pytest

Pytest plugin to select test using Ekstazi algorithm

pytest-elastic-reporter
last release: Mar 13, 2026, *status*: N/A, *requires*: pytest>=7.0

pytest-elasticsearch
last release: Feb 16, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.4.0

Elasticsearch fixtures and fixture factories for Pytest.

pytest-elasticsearch-test
last release: Apr 20, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.0

Elasticsearch fixtures and fixture factories for Pytest.

pytest-elegant
last release: Apr 13, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin that provides elegant, beautiful test output

pytest-elements
last release: Jan 13, 2021, *status*: N/A, *requires*: pytest (>=5.4,<6.0)

Tool to help automate user interfaces

pytest-eliot
last release: Aug 31, 2022, *status*: 1 - Planning, *requires*: pytest (>=5.4.0)

An eliot plugin for pytest.

pytest-elk-reporter
last release: Jul 25, 2024, *status*: 4 - Beta, *requires*: pytest>=3.5.0

A simple plugin to use with pytest

pytest-email
last release: Jul 08, 2020, *status*: N/A, *requires*: pytest

Send execution result email

pytest-embedded
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0

A pytest plugin that designed for embedded testing.

pytest-embedded-arduino
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Make pytest-embedded plugin work with Arduino.

pytest-embedded-idf
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Make pytest-embedded plugin work with ESP-IDF.

pytest-embedded-jtag
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Make pytest-embedded plugin work with JTAG.

pytest-embedded-nuttx
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Make pytest-embedded plugin work with NuttX.

pytest-embedded-qemu
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Make pytest-embedded plugin work with QEMU.

pytest-embedded-serial
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Make pytest-embedded plugin work with Serial.

pytest-embedded-serial-esp
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Make pytest-embedded plugin work with Espressif target boards.

pytest-embedded-wokwi
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Make pytest-embedded plugin work with the Wokwi CLI.

pytest-embrace
last release: Mar 25, 2023, *status*: N/A, *requires*: pytest (>=7.0,<8.0)

💝 Dataclasses-as-tests. Describe the runtime once and multiply coverage with no boilerplate.

pytest-emoji
last release: Feb 19, 2019, *status*: 4 - Beta, *requires*: pytest (>=4.2.1)

A pytest plugin that adds emojis to your test result report

pytest-emoji-output
last release: Apr 09, 2023, *status*: 4 - Beta, *requires*: pytest (==7.0.1)

Pytest plugin to represent test output with emoji support

pytest-enabler
last release: May 16, 2025, *status*: 5 - Production/Stable, *requires*: pytest!=8.1.*,>=6; extra == "test"

Enable installed pytest plugins

pytest-encode
last release: Nov 06, 2021, *status*: N/A, *requires*: N/A

set your encoding and logger

pytest-encode-kane
last release: Nov 16, 2021, *status*: N/A, *requires*: pytest

set your encoding and logger

pytest-encoding
last release: Aug 11, 2023, *status*: N/A, *requires*: pytest

set your encoding and logger

pytest_energy_reporter
last release: Mar 28, 2024, *status*: 3 - Alpha, *requires*: pytest<9.0.0,>=8.1.1

An energy estimation reporter for pytest

pytest-enhanced-reports
last release: Dec 15, 2022, *status*: N/A, *requires*: N/A

Enhanced test reports for pytest

pytest-enhancements
last release: Oct 30, 2019, *status*: 4 - Beta, *requires*: N/A

Improvements for pytest (rejected upstream)

pytest-env
last release: Mar 12, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.2

pytest plugin that allows you to add environment variables.

pytest-envfiles
last release: Oct 08, 2015, *status*: 3 - Alpha, *requires*: N/A

A py.test plugin that parses environment files before running tests

pytest-env-info
last release: Nov 25, 2017, *status*: 4 - Beta, *requires*: pytest (>=3.1.1)

Push information about the running pytest into envvars

pytest-environment
last release: Mar 17, 2024, *status*: 1 - Planning, *requires*: N/A

Pytest Environment

pytest-envraw
last release: Aug 27, 2020, *status*: 4 - Beta, *requires*: pytest (>=2.6.0)

py.test plugin that allows you to add environment variables.

pytest-envvars
last release: Jun 13, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=3.0.0)

Pytest plugin to validate use of envvars on your tests

pytest-envx
last release: Jun 28, 2025, *status*: 4 - Beta, *requires*: pytest>=8.4.1

Pytest plugin for managing environment variables with interpolation and .env file support.

pytest-env-yaml
last release: Apr 02, 2019, *status*: N/A, *requires*: N/A

pytest-ephemeral-container
last release: Apr 14, 2026, *status*: N/A, *requires*: pytest

Spawn epehemeral containers in pytest

pytest-eradicate
last release: Sep 08, 2020, *status*: N/A, *requires*: pytest (>=2.4.2)

pytest plugin to check for commented out code

pytest_erp
last release: Jan 13, 2015, *status*: N/A, *requires*: N/A

py.test plugin to send test info to report portal dynamically

pytest-error
last release: Dec 06, 2025, *status*: 4 - Beta, *requires*: pytest>=8.4

A decorator for testing exceptions with pytest

pytest-error-for-skips
last release: Dec 19, 2019, *status*: 4 - Beta, *requires*: pytest (>=4.6)

Pytest plugin to treat skipped tests a test failure

pytest-errxfail
last release: Jan 06, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

pytest plugin to mark a test as xfailed if it fails with the specified error message in the captured output

pytest-essentials
last release: May 19, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0

A Pytest plugin providing essential utilities like soft assertions.

pytest-eth
last release: Aug 14, 2020, *status*: 1 - Planning, *requires*: N/A

PyTest plugin for testing Smart Contracts for Ethereum Virtual Machine (EVM).

pytest-ethereum
last release: Jun 24, 2019, *status*: 3 - Alpha, *requires*: pytest (==3.3.2); extra == 'dev'

pytest-ethereum: Pytest library for ethereum projects.

pytest-eucalyptus
last release: Jun 28, 2022, *status*: N/A, *requires*: pytest (>=4.2.0)

Pytest Plugin for BDD

pytest-eval
last release: Feb 11, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

LLM testing for humans.

pytest-evals
last release: Feb 02, 2025, *status*: N/A, *requires*: pytest>=7.0.0

A pytest plugin for running and analyzing LLM evaluation tests

pytest-eventlet
last release: Oct 04, 2021, *status*: N/A, *requires*: pytest ; extra == 'dev'

Applies eventlet monkey-patch as a pytest plugin.

pytest-everyfunc
last release: Apr 30, 2025, *status*: 4 - Beta, *requires*: pytest

A pytest plugin to detect completely untested functions using coverage

pytest_evm
last release: Sep 23, 2024, *status*: 4 - Beta, *requires*: pytest<9.0.0,>=8.1.1

The testing package containing tools to test Web3-based projects

pytest_exact_fixtures
last release: Feb 04, 2019, *status*: N/A, *requires*: N/A

Parse queries in Lucene and Elasticsearch syntaxes

pytest-examples
last release: May 06, 2025, *status*: N/A, *requires*: pytest>=7

Pytest plugin for testing examples in docstrings and markdown files.

pytest-exasol-backend
last release: Mar 03, 2026, *status*: N/A, *requires*: pytest<9,>=7

pytest-exasol-extension
last release: Feb 06, 2026, *status*: N/A, *requires*: pytest<9,>=7

pytest-exasol-itde
last release: Nov 22, 2024, *status*: N/A, *requires*: pytest<9,>=7

pytest-exasol-saas
last release: Nov 22, 2024, *status*: N/A, *requires*: pytest<9,>=7

pytest-exasol-slc
last release: Mar 17, 2026, *status*: N/A, *requires*: pytest<9,>=7

pytest-excel
last release: Jul 22, 2025, *status*: 5 - Production/Stable, *requires*: pytest

pytest plugin for generating excel reports

pytest-exceptional
last release: Mar 16, 2017, *status*: 4 - Beta, *requires*: N/A

Better exceptions

pytest-exception-script
last release: Aug 04, 2020, *status*: 3 - Alpha, *requires*: pytest

Walk your code through exception script to check it's resiliency to failures.

pytest-executable
last release: Oct 07, 2023, *status*: N/A, *requires*: pytest <8,>=5

pytest plugin for testing executables

pytest-execution-timer
last release: Dec 24, 2021, *status*: 4 - Beta, *requires*: N/A

A timer for the phases of Pytest's execution.

pytest-exit-code
last release: May 06, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A pytest plugin that overrides the built-in exit codes to retain more information about the test results.

pytest-exit-status
last release: Jan 25, 2025, *status*: N/A, *requires*: pytest>=8.0.0

Enhance.

pytest-expect
last release: Apr 21, 2016, *status*: 4 - Beta, *requires*: N/A

py.test plugin to store test expectations and mark tests based on them

pytest-expectdir
last release: Mar 19, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=5.0)

A pytest plugin to provide initial/expected directories, and check a test transforms the initial directory to the expected one

pytest-expected
last release: Feb 26, 2025, *status*: N/A, *requires*: pytest

Record and play back your expectations

pytest-expecter
last release: Sep 18, 2022, *status*: 5 - Production/Stable, *requires*: N/A

Better testing with expecter and pytest.

pytest-expectr
last release: Oct 05, 2018, *status*: N/A, *requires*: pytest (>=2.4.2)

This plugin is used to expect multiple assert using pytest framework.

pytest-expect-test
last release: Apr 10, 2023, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A fixture to support expect tests in pytest

pytest-experiments
last release: Dec 13, 2021, *status*: 4 - Beta, *requires*: pytest (>=6.2.5,<7.0.0)

A pytest plugin to help developers of research-oriented software projects keep track of the results of their numerical experiments.

pytest-explicit
last release: Jun 15, 2021, *status*: 5 - Production/Stable, *requires*: pytest

A Pytest plugin to ignore certain marked tests by default

pytest-exploratory
last release: Sep 18, 2024, *status*: N/A, *requires*: pytest>=6.2

Interactive console for pytest.

pytest-explorer
last release: Aug 01, 2023, *status*: N/A, *requires*: N/A

terminal ui for exploring and running tests

pytest-ext
last release: Mar 31, 2024, *status*: N/A, *requires*: pytest>=5.3

pytest plugin for automation test

pytest-extended-mock
last release: Mar 12, 2025, *status*: N/A, *requires*: pytest<9.0.0,>=8.3.5

a pytest extension for easy mock setup

pytest-extensions
last release: Aug 17, 2022, *status*: 4 - Beta, *requires*: pytest ; extra == 'testing'

A collection of helpers for pytest to ease testing

pytest-external-blockers
last release: Oct 05, 2021, *status*: N/A, *requires*: pytest

a special outcome for tests that are blocked for external reasons

pytest_extra
last release: Aug 14, 2014, *status*: N/A, *requires*: N/A

Some helpers for writing tests with pytest.

pytest-extra-durations
last release: Apr 21, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A pytest plugin to get durations on a per-function basis and per module basis.

pytest-extra-markers
last release: Mar 05, 2023, *status*: 4 - Beta, *requires*: pytest

Additional pytest markers to dynamically enable/disable tests viia CLI flags

pytest-f3ts
last release: Mar 09, 2026, *status*: N/A, *requires*: pytest<8.0.0,>=7.2.1

Pytest Plugin for communicating test results and information to a FixturFab Test Runner GUI

pytest-fabric
last release: Sep 12, 2018, *status*: 5 - Production/Stable, *requires*: N/A

Provides test utilities to run fabric task tests by using docker containers

pytest-factory
last release: Sep 06, 2020, *status*: 3 - Alpha, *requires*: pytest (>4.3)

Use factories for test setup with py.test

pytest-factoryboy
last release: Jul 01, 2025, *status*: 6 - Mature, *requires*: pytest>=7.0

Factory Boy support for pytest.

pytest-factoryboy-fixtures
last release: Jun 25, 2020, *status*: N/A, *requires*: N/A

Generates pytest fixtures that allow the use of type hinting

pytest-factoryboy-state
last release: Mar 22, 2022, *status*: 5 - Production/Stable, *requires*: pytest (>=5.0)

Simple factoryboy random state management

pytest-fahhh
last release: Apr 16, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

A pytest plugin that plays the fahhh meme sound when a test fails.

pytest-failed-screen-record
last release: Jan 05, 2023, *status*: 4 - Beta, *requires*: pytest (>=7.1.2d,<8.0.0)

Create a video of the screen when pytest fails

pytest-failed-screenshot
last release: Apr 21, 2021, *status*: N/A, *requires*: N/A

Test case fails,take a screenshot,save it,attach it to the allure

pytest-failed-to-verify
last release: Aug 08, 2019, *status*: 5 - Production/Stable, *requires*: pytest (>=4.1.0)

A pytest plugin that helps better distinguishing real test failures from setup flakiness.

pytest-fail-slow
last release: Jun 01, 2024, *status*: N/A, *requires*: pytest>=7.0

Fail tests that take too long to run

pytest-failure-tracker
last release: Jul 17, 2024, *status*: N/A, *requires*: pytest>=6.0.0

A pytest plugin for tracking test failures over multiple runs

pytest-faker
last release: Dec 19, 2016, *status*: 6 - Mature, *requires*: N/A

Faker integration with the pytest framework.

pytest-falcon
last release: Sep 07, 2016, *status*: 4 - Beta, *requires*: N/A

Pytest helpers for Falcon.

pytest-fantasy
last release: Mar 14, 2019, *status*: N/A, *requires*: N/A

Pytest plugin for Flask Fantasy Framework

pytest-fastapi
last release: Dec 27, 2020, *status*: N/A, *requires*: N/A

pytest-fastapi-deps
last release: Jul 20, 2022, *status*: 5 - Production/Stable, *requires*: pytest

A fixture which allows easy replacement of fastapi dependencies for testing

pytest-fastcollect
last release: Nov 19, 2025, *status*: N/A, *requires*: pytest>=7.0.0

A high-performance pytest plugin that replaces test collection with a Rust-based implementation

pytest-fastest
last release: Oct 04, 2023, *status*: 4 - Beta, *requires*: pytest (>=4.4)

Use SCM and coverage to run only needed tests

pytest-fast-first
last release: Jan 19, 2023, *status*: 3 - Alpha, *requires*: pytest

Pytest plugin that runs fast tests first

pytest-faulthandler
last release: Jul 04, 2019, *status*: 6 - Mature, *requires*: pytest (>=5.0)

py.test plugin that activates the fault handler module for tests (dummy package)

pytest-fauna
last release: Jan 03, 2025, *status*: N/A, *requires*: N/A

A collection of helpful test fixtures for Fauna DB.

pytest-fauxfactory
last release: Dec 06, 2017, *status*: 5 - Production/Stable, *requires*: pytest (>=3.2)

Integration of fauxfactory into pytest.

pytest-figleaf
last release: Jan 18, 2010, *status*: 5 - Production/Stable, *requires*: N/A

py.test figleaf coverage plugin

pytest-file
last release: Mar 18, 2024, *status*: 1 - Planning, *requires*: N/A

Pytest File

pytest-filecov
last release: Jun 27, 2021, *status*: 4 - Beta, *requires*: pytest

A pytest plugin to detect unused files

pytest-filedata
last release: Apr 29, 2024, *status*: 5 - Production/Stable, *requires*: N/A

easily load test data from files

pytest-filemarker
last release: Dec 01, 2020, *status*: N/A, *requires*: pytest

A pytest plugin that runs marked tests when files change.

pytest-file-watcher
last release: Mar 23, 2023, *status*: N/A, *requires*: pytest

Pytest-File-Watcher is a CLI tool that watches for changes in your code and runs pytest on the changed files.

pytest-filter-case
last release: Nov 05, 2020, *status*: N/A, *requires*: N/A

run test cases filter by mark

pytest-filterfixtures
last release: Jan 09, 2026, *status*: N/A, *requires*: pytest>=9.0.2

pytest plugin to execute or ignore tests based on fixtures

pytest-filter-subpackage
last release: Mar 04, 2024, *status*: 5 - Production/Stable, *requires*: pytest >=4.6

Pytest plugin for filtering based on sub-packages

pytest-find-dependencies
last release: Jul 16, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=6.2.4

A pytest plugin to find dependencies between tests

pytest-finer-verdicts
last release: Jun 18, 2020, *status*: N/A, *requires*: pytest (>=5.4.3)

A pytest plugin to treat non-assertion failures as test errors.

pytest-firefox
last release: Feb 28, 2025, *status*: N/A, *requires*: N/A

pytest-firestore
last release: Mar 10, 2026, *status*: N/A, *requires*: pytest>=7.0

A Pytest fixture for managing Google Cloud Firestore emulator

pytest-fixedpoint
last release: Mar 12, 2026, *status*: N/A, *requires*: pytest>=7.0

Pytest plugin for recording and replaying deterministic function calls

pytest-fixkit
last release: Apr 02, 2026, *status*: 5 - Production/Stable, *requires*: pytest

A very micro http framework.

pytest-fixture-cache
last release: Jan 25, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Smart fixture caching for pytest with SQLite storage

pytest-fixturecheck
last release: Jun 02, 2025, *status*: 3 - Alpha, *requires*: pytest>=6.0.0

A pytest plugin to check fixture validity before test execution

pytest-fixture-classes
last release: Oct 12, 2025, *status*: 5 - Production/Stable, *requires*: N/A

Fixtures as classes that work well with dependency injection, autocompletetion, type checkers, and language servers

pytest-fixture-collect
last release: Jul 25, 2025, *status*: N/A, *requires*: pytest; extra == "test"

A utility to collect pytest fixture file paths.

pytest-fixturecollection
last release: Feb 22, 2024, *status*: 4 - Beta, *requires*: pytest >=3.5.0

A pytest plugin to collect tests based on fixtures being used by tests

pytest-fixture-config
last release: Oct 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Fixture configuration utils for py.test

pytest-fixture-forms
last release: Dec 06, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=7.0.0

A pytest plugin for creating fixtures that holds different forms between tests.

pytest-fixture-maker
last release: Sep 21, 2021, *status*: N/A, *requires*: N/A

Pytest plugin to load fixtures from YAML files

pytest-fixture-marker
last release: Oct 11, 2020, *status*: 5 - Production/Stable, *requires*: N/A

A pytest plugin to add markers based on fixtures used.

pytest-fixture-order
last release: Oct 22, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=3.0

pytest plugin to control fixture evaluation order

pytest-fixture-ref
last release: Nov 17, 2022, *status*: 4 - Beta, *requires*: N/A

Lets users reference fixtures without name matching magic.

pytest-fixture-remover
last release: Feb 14, 2024, *status*: 5 - Production/Stable, *requires*: N/A

A LibCST codemod to remove pytest fixtures applied via the usefixtures decorator, as well as its parametrizations.

pytest-fixture-rtttg
last release: Feb 23, 2022, *status*: N/A, *requires*: pytest (>=7.0.1,<8.0.0)

Warn or fail on fixture name clash

pytest-fixtures
last release: May 01, 2019, *status*: 5 - Production/Stable, *requires*: N/A

Common fixtures for pytest

pytest-fixtures-fixtures
last release: Nov 06, 2025, *status*: 4 - Beta, *requires*: pytest>=8.4.1

Handy fixtues to access your fixtures from your _pytest tests.

pytest-fixture-timing
last release: Dec 11, 2025, *status*: N/A, *requires*: pytest>=7.0

Tiny plugin to report total duration per fixture

pytest-fixture-tools
last release: Apr 30, 2025, *status*: 6 - Mature, *requires*: pytest

Plugin for pytest which provides tools for fixtures

pytest-fixture-typecheck
last release: Aug 24, 2021, *status*: N/A, *requires*: pytest

A pytest plugin to assert type annotations at runtime.

pytest-fkit
last release: Apr 16, 2026, *status*: N/A, *requires*: pytest>=6.0.0

A pytest plugin that prevents crashes from killing your test suite

pytest-flake8
last release: Nov 09, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7.0

pytest plugin to check FLAKE8 requirements

pytest-flake8-path
last release: Sep 09, 2025, *status*: 5 - Production/Stable, *requires*: pytest

A pytest fixture for testing flake8 plugins.

pytest-flake8-v2
last release: Mar 01, 2022, *status*: 5 - Production/Stable, *requires*: pytest (>=7.0)

pytest plugin to check FLAKE8 requirements

pytest-flake-detection
last release: Nov 29, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Continuously runs your tests to detect flaky tests

pytest-flakefighters
last release: Mar 05, 2026, *status*: N/A, *requires*: pytest>=6.2.0

Pytest plugin implementing flaky test failure detection and classification.

pytest-flakefinder
last release: Oct 26, 2022, *status*: 4 - Beta, *requires*: pytest (>=2.7.1)

Runs tests multiple times to expose flakiness.

pytest-flakehunter
last release: Apr 07, 2026, *status*: N/A, *requires*: pytest>=7.0

Re-run tests N times, visualize failure heatmaps, and get AI root cause hypotheses

pytest-flakemark
last release: Mar 23, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

FLAKEMARK — Differential execution tracer that finds the exact file, line, and root cause of any flaky test.

pytest-flakes
last release: Dec 02, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=5)

pytest plugin to check source code with pyflakes

pytest-flakiness
last release: Mar 24, 2026, *status*: N/A, *requires*: pytest>=9.0.2

Pytest reporter for Flakiness.io

pytest-flaptastic
last release: Mar 17, 2019, *status*: N/A, *requires*: N/A

Flaptastic py.test plugin

pytest-flask
last release: Oct 23, 2023, *status*: 5 - Production/Stable, *requires*: pytest >=5.2

A set of py.test fixtures to test Flask applications.

pytest-flask-ligand
last release: Apr 25, 2023, *status*: 4 - Beta, *requires*: pytest (~=7.3)

Pytest fixtures and helper functions to use for testing flask-ligand microservices.

pytest-flask-sqlalchemy
last release: Apr 30, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.2.1)

A pytest plugin for preserving test isolation in Flask-SQlAlchemy using database transactions.

pytest-flask-sqlalchemy-transactions
last release: Aug 02, 2018, *status*: 4 - Beta, *requires*: pytest (>=3.2.1)

Run tests in transactions using pytest, Flask, and SQLalchemy.

pytest-flexreport
last release: Apr 15, 2023, *status*: 4 - Beta, *requires*: pytest

pytest-fluent
last release: Aug 14, 2024, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin in order to provide logs via fluentd

pytest-fluentbit
last release: Jun 16, 2023, *status*: 4 - Beta, *requires*: pytest (>=7.0.0)

A pytest plugin in order to provide logs via fluentbit

pytest-fly
last release: Apr 18, 2026, *status*: 3 - Alpha, *requires*: pytest

pytest runner and observer

pytest-flyte
last release: May 03, 2021, *status*: N/A, *requires*: pytest

Pytest fixtures for simplifying Flyte integration testing

pytest-fmu-filter
last release: Jun 23, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin to filter fmus

pytest-focus
last release: May 04, 2019, *status*: 4 - Beta, *requires*: pytest

A pytest plugin that alerts user of failed test cases with screen notifications

pytest-forbid
last release: Mar 07, 2023, *status*: N/A, *requires*: pytest (>=7.2.2,<8.0.0)

pytest-forcefail
last release: May 15, 2018, *status*: 4 - Beta, *requires*: N/A

py.test plugin to make the test failing regardless of pytest.mark.xfail

pytest-forger
last release: Mar 09, 2026, *status*: N/A, *requires*: pytest>=7.4.0

Automatic test scaffolding and mock generation for Python

pytest-forward-compatability
last release: Sep 06, 2020, *status*: N/A, *requires*: N/A

A name to avoid typosquating pytest-foward-compatibility

pytest-forward-compatibility
last release: Sep 29, 2020, *status*: N/A, *requires*: N/A

A pytest plugin to shim pytest commandline options for fowards compatibility

pytest-frappe
last release: Jul 30, 2024, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Pytest Frappe Plugin - A set of pytest fixtures to test Frappe applications

pytest-freethreaded
last release: Oct 03, 2024, *status*: 5 - Production/Stable, *requires*: pytest

pytest plugin for running parallel tests

pytest-freeze
last release: Jan 27, 2026, *status*: N/A, *requires*: N/A

Pytest plugin to simplify writing freeze tests.

pytest-freezeblaster
last release: Oct 13, 2025, *status*: N/A, *requires*: pytest>=6.2.5

Wrap tests with fixtures in freeze_time

pytest-freezegun
last release: Jul 19, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.0.0)

Wrap tests with fixtures in freeze_time

pytest-freezer
last release: Dec 12, 2024, *status*: N/A, *requires*: pytest>=3.6

Pytest plugin providing a fixture interface for spulec/freezegun

pytest-freeze-reqs
last release: Apr 29, 2021, *status*: N/A, *requires*: N/A

Check if requirement files are frozen

pytest-frozen-uuids
last release: Apr 17, 2022, *status*: N/A, *requires*: pytest (>=3.0)

Deterministically frozen UUID's for your tests

pytest_ftpserver
last release: Feb 10, 2026, *status*: 5 - Production/Stable, *requires*: pytest

A PyTest plugin which provides an FTP fixture for your tests

pytest-func-cov
last release: Apr 15, 2021, *status*: 3 - Alpha, *requires*: pytest (>=5)

Pytest plugin for measuring function coverage

pytest-funcnodes
last release: Dec 21, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Testing plugin for funcnodes

pytest-funparam
last release: Dec 02, 2021, *status*: 4 - Beta, *requires*: pytest >=4.6.0

An alternative way to parametrize test cases.

pytest-fv
last release: Jun 06, 2025, *status*: N/A, *requires*: pytest

pytest extensions to support running functional-verification jobs

pytest-fxa
last release: Aug 28, 2018, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for Firefox Accounts

pytest-fxa-mte
last release: Mar 23, 2026, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for Firefox Accounts

pytest-fxtest
last release: Oct 27, 2020, *status*: N/A, *requires*: N/A

pytest-fzf
last release: Jan 06, 2025, *status*: 4 - Beta, *requires*: pytest>=6.0.0

fzf-based test selector for pytest

pytest_gae
last release: Aug 03, 2016, *status*: 3 - Alpha, *requires*: N/A

pytest plugin for apps written with Google's AppEngine

pytest-gak
last release: Apr 10, 2025, *status*: N/A, *requires*: N/A

A Pytest plugin and command line tool for interactive testing with Pytest

pytest-gather-fixtures
last release: Aug 18, 2024, *status*: N/A, *requires*: pytest>=7.0.0

set up asynchronous pytest fixtures concurrently

pytest-gc
last release: Feb 01, 2018, *status*: N/A, *requires*: N/A

The garbage collector plugin for py.test

pytest-gcov
last release: Feb 01, 2018, *status*: 3 - Alpha, *requires*: N/A

Uses gcov to measure test coverage of a C library

pytest-gcppubsub
last release: Mar 10, 2026, *status*: N/A, *requires*: pytest>=7.0

A Pytest fixture for managing Google Cloud Platform PubSub emulator

pytest-gcpsecretmanager
last release: Feb 18, 2026, *status*: N/A, *requires*: pytest>=7.0

A PyTest plugin for mocking GCP's Secret Manager

pytest-gcs
last release: Jan 24, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=6.2

GCS fixtures and fixture factories for Pytest.

pytest-gee
last release: Oct 16, 2025, *status*: 3 - Alpha, *requires*: pytest

The Python plugin for your GEE based packages.

pytest-gevent
last release: Feb 25, 2020, *status*: N/A, *requires*: pytest

Ensure that gevent is properly patched when invoking pytest

pytest-gherkin
last release: Jul 27, 2019, *status*: 3 - Alpha, *requires*: pytest (>=5.0.0)

A flexible framework for executing BDD gherkin tests

pytest-gh-log-group
last release: Jan 11, 2022, *status*: 3 - Alpha, *requires*: pytest

pytest plugin for gh actions

pytest-ghostinspector
last release: May 17, 2016, *status*: 3 - Alpha, *requires*: N/A

For finding/executing Ghost Inspector tests

pytest-girder
last release: Apr 08, 2026, *status*: N/A, *requires*: pytest>=3.6

A set of pytest fixtures for testing Girder applications.

pytest-git
last release: Oct 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Git repository fixture for py.test

pytest-gitconfig
last release: Dec 28, 2025, *status*: 4 - Beta, *requires*: pytest>=7.1.2

Provide a Git config sandbox for testing

pytest-gitcov
last release: Jan 11, 2020, *status*: 2 - Pre-Alpha, *requires*: N/A

Pytest plugin for reporting on coverage of the last git commit.

pytest-git-diff
last release: Apr 02, 2024, *status*: N/A, *requires*: N/A

Pytest plugin that allows the user to select the tests affected by a range of git commits

pytest-git-fixtures
last release: Mar 11, 2021, *status*: 4 - Beta, *requires*: pytest

Pytest fixtures for testing with git.

pytest-github
last release: Mar 07, 2019, *status*: 5 - Production/Stable, *requires*: N/A

Plugin for py.test that associates tests with github issues using a marker.

pytest-github-actions-annotate-failures
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

pytest plugin to annotate failed tests with a workflow command for GitHub Actions

pytest-github-report
last release: Jun 03, 2022, *status*: 4 - Beta, *requires*: N/A

Generate a GitHub report using pytest in GitHub Workflows

pytest-gitignore
last release: Jul 17, 2015, *status*: 4 - Beta, *requires*: N/A

py.test plugin to ignore the same files as git

pytest-gitlab
last release: Oct 16, 2024, *status*: N/A, *requires*: N/A

Pytest Plugin for Gitlab

pytest-gitlabci-parallelized
last release: Mar 08, 2023, *status*: N/A, *requires*: N/A

Parallelize pytest across GitLab CI workers.

pytest-gitlab-code-quality
last release: Nov 23, 2025, *status*: N/A, *requires*: pytest>=8.1.1

Collects warnings while testing and generates a GitLab Code Quality Report.

pytest-gitlab-fold
last release: Dec 31, 2023, *status*: 4 - Beta, *requires*: pytest >=2.6.0

Folds output sections in GitLab CI build log

pytest-gitscope
last release: Sep 24, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

A pragmatic pytest plugin that runs only the tests that matter, and ship faster

pytest-git-selector
last release: Nov 17, 2022, *status*: N/A, *requires*: N/A

Utility to select tests that have had its dependencies modified (as identified by git diff)

pytest-glamor-allure
last release: Jan 30, 2026, *status*: 5 - Production/Stable, *requires*: pytest<=9.0.2

Extends allure-pytest functionality

pytest-glaze
last release: Apr 10, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

A thin, transparent coat that makes your test output shine.

pytest-glow-report
last release: Dec 08, 2025, *status*: 4 - Beta, *requires*: pytest>=6.0; extra == "dev"

Beautiful, glowing HTML test reports for PyTest and unittest.

pytest-gnupg-fixtures
last release: Mar 04, 2021, *status*: 4 - Beta, *requires*: pytest

Pytest fixtures for testing with gnupg.

pytest-golden
last release: Jan 06, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6.1.2

Plugin for pytest that offloads expected outputs to data files

pytest-goldie
last release: May 23, 2023, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A plugin to support golden tests with pytest.

pytest-google-chat
last release: Mar 27, 2022, *status*: 4 - Beta, *requires*: pytest

Notify google chat channel for test results

pytest-google-cloud-storage
last release: Sep 11, 2025, *status*: N/A, *requires*: pytest>=8.0.0

Pytest custom features, e.g. fixtures and various tests. Aimed to emulate Google Cloud Storage service

pytest-grader
last release: Aug 25, 2025, *status*: N/A, *requires*: pytest>=8

Pytest extension for scoring programming assignments.

pytest-gradescope
last release: Apr 29, 2025, *status*: N/A, *requires*: N/A

A pytest plugin for Gradescope integration

pytest-graphql-schema
last release: Oct 18, 2019, *status*: N/A, *requires*: N/A

Get graphql schema as fixture for pytest

pytest-greendots
last release: Feb 08, 2014, *status*: 3 - Alpha, *requires*: N/A

Green progress dots

pytest-greener
last release: Dec 24, 2025, *status*: N/A, *requires*: pytest<9.0.0,>=8.3.3

Pytest plugin for Greener

pytest-green-light
last release: Nov 03, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

Pytest plugin that gives SQLAlchemy async engines the green light - automatically fixes MissingGreenlet errors

pytest-greet
last release: Oct 21, 2025, *status*: N/A, *requires*: N/A

pytest-gremlins
last release: Apr 03, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

Fast-first mutation testing for pytest. Let the gremlins loose, see which ones survive.

pytest-group-by-class
last release: Jun 27, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=2.5)

A Pytest plugin for running a subset of your tests by splitting them in to groups of classes.

pytest-growl
last release: Jan 13, 2014, *status*: 5 - Production/Stable, *requires*: N/A

Growl notifications for pytest results.

pytest-grpc
last release: May 01, 2020, *status*: N/A, *requires*: pytest (>=3.6.0)

pytest plugin for grpc

pytest-grpc-aio
last release: Oct 28, 2025, *status*: N/A, *requires*: pytest>=3.6.0

pytest plugin for grpc.aio

pytest-grunnur
last release: Jul 26, 2024, *status*: N/A, *requires*: pytest>=6

Py.Test plugin for Grunnur-based packages.

pytest_gui_status
last release: Jan 23, 2016, *status*: N/A, *requires*: pytest

Show pytest status in gui

pytest-hammer
last release: Mar 27, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

tools such as db tools of pytest

pytest-hammertime
last release: Jul 28, 2018, *status*: N/A, *requires*: pytest

Display "🔨 " instead of "." for passed pytest tests.

pytest-hardware-test-report
last release: Apr 01, 2024, *status*: 4 - Beta, *requires*: pytest<9.0.0,>=8.0.0

A simple plugin to use with pytest

pytest-harmony
last release: Jan 17, 2023, *status*: N/A, *requires*: pytest (>=7.2.1,<8.0.0)

Chain tests and data with pytest

pytest-harvest
last release: Mar 16, 2024, *status*: 5 - Production/Stable, *requires*: N/A

Store data created during your pytest tests execution, and retrieve it at the end of the session, e.g. for applicative benchmarking purposes.

pytest-helm
last release: Feb 21, 2026, *status*: 3 - Alpha, *requires*: pytest>=8.0.0

Simple, ergonomic Helm manifest fixtures for pytest.

pytest-helm-charts
last release: Dec 23, 2025, *status*: 4 - Beta, *requires*: pytest<9,>=8.0.0

A plugin to provide different types and configs of Kubernetes clusters that can be used for testing.

pytest-helm-templates
last release: Aug 07, 2024, *status*: N/A, *requires*: pytest~=7.4.0; extra == "dev"

Pytest fixtures for unit testing the output of helm templates

pytest-helper
last release: May 31, 2019, *status*: 5 - Production/Stable, *requires*: N/A

Functions to help in using the pytest testing framework

pytest-helpers
last release: May 17, 2020, *status*: N/A, *requires*: pytest

pytest helpers

pytest-helpers-namespace
last release: Dec 29, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=6.0.0)

Pytest Helpers Namespace Plugin

pytest-henry
last release: Aug 29, 2023, *status*: N/A, *requires*: N/A

pytest-hidecaptured
last release: May 04, 2018, *status*: 4 - Beta, *requires*: pytest (>=2.8.5)

Hide captured output

pytest-himark
last release: Jun 05, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

This plugin aims to create markers automatically based on a json configuration.

pytest-historic
last release: Apr 08, 2020, *status*: N/A, *requires*: pytest

Custom report to display pytest historical execution records

pytest-historic-hook
last release: Apr 08, 2020, *status*: N/A, *requires*: pytest

Custom listener to store execution results into MYSQL DB, which is used for pytest-historic report

pytest-history
last release: Jan 14, 2024, *status*: N/A, *requires*: pytest (>=7.4.3,<8.0.0)

Pytest plugin to keep a history of your pytest runs

pytest-home
last release: Jul 28, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Home directory fixtures

pytest-homeassistant
last release: Aug 12, 2020, *status*: 4 - Beta, *requires*: N/A

A pytest plugin for use with homeassistant custom components.

pytest-homeassistant-custom-component
last release: Apr 18, 2026, *status*: 3 - Alpha, *requires*: pytest==9.0.0

Experimental package to automatically extract test plugins for Home Assistant custom components

pytest-homeassistant-custom-component-framework
last release: Apr 18, 2026, *status*: 3 - Alpha, *requires*: pytest==9.0.0

Experimental package to automatically extract test plugins for Home Assistant custom components

pytest-Honda-report
last release: Apr 11, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

Enterprise-grade pytest HTML report plugin with Chinese UI, API details, and historical trends

pytest-honey
last release: Jan 07, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A simple plugin to use with pytest

pytest-honors
last release: Mar 06, 2020, *status*: 4 - Beta, *requires*: N/A

Report on tests that honor constraints, and guard against regressions

pytest-hot-reloading
last release: Sep 23, 2024, *status*: N/A, *requires*: N/A

pytest-hot-test
last release: Dec 10, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A plugin that tracks test changes

pytest-houdini
last release: Jul 15, 2024, *status*: N/A, *requires*: pytest

pytest plugin for testing code in Houdini.

pytest-hoverfly
last release: Jan 30, 2023, *status*: N/A, *requires*: pytest (>=5.0)

Simplify working with Hoverfly from pytest

pytest-hoverfly-wrapper
last release: Feb 27, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=3.7.0)

Integrates the Hoverfly HTTP proxy into Pytest

pytest-hpfeeds
last release: Feb 28, 2023, *status*: 4 - Beta, *requires*: pytest (>=6.2.4,<7.0.0)

Helpers for testing hpfeeds in your python project

pytest-html
last release: Jan 19, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7

pytest plugin for generating HTML reports

pytest-html5
last release: Dec 18, 2025, *status*: N/A, *requires*: N/A

the best report for pytest

pytest-html-cn
last release: Aug 19, 2024, *status*: 5 - Production/Stable, *requires*: pytest!=6.0.0,>=5.0

pytest plugin for generating HTML reports

pytest-html-dashboard
last release: Nov 24, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Beautiful dashboard-style HTML reports for pytest with charts, error analysis, and visual insights

pytest-html-lee
last release: Jun 30, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=5.0)

optimized pytest plugin for generating HTML reports

pytest-html-merger
last release: Jul 12, 2024, *status*: N/A, *requires*: N/A

Pytest HTML reports merging utility

pytest-html-nova-act
last release: Mar 30, 2026, *status*: N/A, *requires*: N/A

A Pytest Plugin for Amazon Nova Act Python SDK.

pytest-html-object-storage
last release: Jan 17, 2024, *status*: 5 - Production/Stable, *requires*: N/A

Pytest report plugin for send HTML report on object-storage

pytest-html-plus
last release: Apr 12, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Generate Actionable, automatic screenshots, unified Mobile friendly Pytest HTML report in less than 3 seconds — no hooks, merge plugins, no config, xdist-ready.

pytest-html-profiling
last release: Feb 11, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=3.0)

Pytest plugin for generating HTML reports with per-test profiling and optionally call graph visualizations. Based on pytest-html by Dave Hunt.

pytest-html-report
last release: Jun 24, 2025, *status*: 4 - Beta, *requires*: pytest>=6.0

Enhanced HTML reporting for pytest with categories, specifications, and detailed logging

pytest-html-report-builder
last release: Apr 15, 2026, *status*: N/A, *requires*: pytest>=7.0

A pytest plugin that generates self-contained HTML automation reports with visual charts.

pytest-html-reporter
last release: Feb 13, 2022, *status*: N/A, *requires*: N/A

Generates a static html report based on pytest framework

pytest-html-report-merger
last release: May 22, 2024, *status*: N/A, *requires*: N/A

pytest-html-thread
last release: Dec 29, 2020, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for generating HTML reports

pytest-htmlx
last release: Sep 09, 2025, *status*: 4 - Beta, *requires*: pytest

Custom HTML report plugin for Pytest with charts and tables

pytest-http
last release: Aug 22, 2024, *status*: N/A, *requires*: pytest

Fixture “http” for http requests

pytest-httpbin
last release: Sep 18, 2024, *status*: 5 - Production/Stable, *requires*: pytest; extra == “test”

Easily test your HTTP library against a local copy of httpbin

pytest-httpchain
last release: Apr 02, 2026, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for HTTP testing using JSON files

pytest-httpchain-core
last release: Apr 02, 2026, *status*: N/A, *requires*: N/A

Shared base types for pytest-httpchain

pytest-httpchain-jsonref
last release: Apr 02, 2026, *status*: N/A, *requires*: N/A

JSON reference ($ref) support for pytest-httpchain

pytest-httpchain-mcp
last release: Apr 02, 2026, *status*: N/A, *requires*: N/A

MCP server for pytest-httpchain

pytest-httpchain-models
last release: Apr 02, 2026, *status*: N/A, *requires*: N/A

Pydantic models for pytest-httpchain

pytest-httpchain-templates
last release: Apr 02, 2026, *status*: N/A, *requires*: N/A

Templating support for pytest-httpchain

pytest-httpchain-userfunc
last release: Apr 02, 2026, *status*: N/A, *requires*: N/A

User functions support for pytest-httpchain

pytest-httpdbg
last release: Mar 29, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin to record HTTP(S) requests with stack trace.

pytest-http-mocker
last release: Oct 20, 2019, *status*: N/A, *requires*: N/A

Pytest plugin for http mocking (via https://github.com/vilus/mocker)

pytest-httpretty
last release: Feb 16, 2014, *status*: 3 - Alpha, *requires*: N/A

A thin wrapper of HTTPretty for pytest

pytest_httpserver
last release: Feb 14, 2026, *status*: 3 - Alpha, *requires*: N/A

pytest-httpserver is a httpserver for pytest

pytest-httptesting
last release: Dec 19, 2024, *status*: N/A, *requires*: pytest>=8.2.0

http_testing framework on top of pytest

pytest-httpx
last release: Apr 09, 2026, *status*: 5 - Production/Stable, *requires*: pytest==9.*

Send responses to httpx.

pytest-httpx-blockage
last release: Feb 16, 2023, *status*: N/A, *requires*: pytest (>=7.2.1)

Disable httpx requests during a test run

pytest-httpx-recorder
last release: Jan 04, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Recorder feature based on pytest_httpx, like recorder feature in responses.

pytest-hue
last release: May 09, 2019, *status*: N/A, *requires*: N/A

Visualise PyTest status via your Phillips Hue lights

pytest-human
last release: Jan 25, 2026, *status*: 4 - Beta, *requires*: pytest>=8

A beautiful nested pytest HTML test report

pytest-hy
last release: Feb 11, 2026, *status*: N/A, *requires*: pytest>=7.0

Pytest plugin for discovering and running Hy test files

pytest-hylang
last release: Mar 28, 2021, *status*: N/A, *requires*: pytest

Pytest plugin to allow running tests written in hylang

pytest-hypo-25
last release: Jan 12, 2020, *status*: 3 - Alpha, *requires*: N/A

help hypo module for pytest

pytest-hypothesis
last release: Feb 09, 2026, *status*: N/A, *requires*: N/A

pytest-iam
last release: Nov 02, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A fully functional OAUTH2 / OpenID Connect (OIDC) / SCIM server to be used in your testsuite

pytest-ibutsu
last release: Feb 23, 2026, *status*: 4 - Beta, *requires*: pytest

A plugin to sent pytest results to an Ibutsu server

pytest-icdiff
last release: Dec 05, 2023, *status*: 4 - Beta, *requires*: pytest

use icdiff for better error messages in pytest assertions

pytest-idapro
last release: Nov 03, 2018, *status*: N/A, *requires*: N/A

A pytest plugin for idapython. Allows a pytest setup to run tests outside and inside IDA in an automated manner by runnig pytest inside IDA and by mocking idapython api

pytest-idem
last release: Dec 13, 2023, *status*: 5 - Production/Stable, *requires*: N/A

A pytest plugin to help with testing idem projects

pytest-idempotent
last release: Jul 25, 2022, *status*: N/A, *requires*: N/A

Pytest plugin for testing function idempotence.

pytest-ignore-flaky
last release: Apr 20, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=6.0

ignore failures from flaky tests (pytest plugin)

pytest-ignore-test-results
last release: Feb 03, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.0

A pytest plugin to ignore test results.

pytest-image-diff
last release: Dec 31, 2024, *status*: 3 - Alpha, *requires*: pytest

pytest-image-snapshot
last release: Jul 16, 2025, *status*: 4 - Beta, *requires*: pytest>=3.5.0

A pytest plugin for image snapshot management and comparison.

pytest-impacted
last release: Apr 16, 2026, *status*: 4 - Beta, *requires*: pytest>=8.0.0

A pytest plugin that selectively runs tests impacted by code changes via git introspection, AST parsing, and dependency graph analysis.

pytest-impacted-rs
last release: Apr 16, 2026, *status*: 4 - Beta, *requires*: N/A

Rust-accelerated import parsing for pytest-impacted (ruff parser + rayon parallelism).

pytest-imply
last release: Mar 21, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

Pytest plugin for test implication — skip tests implied by stronger ones

pytest-import-check
last release: Jul 19, 2024, *status*: 3 - Alpha, *requires*: pytest>=8.1

pytest plugin to check whether Python modules can be imported

pytest-incremental
last release: Apr 24, 2021, *status*: 5 - Production/Stable, *requires*: N/A

an incremental test runner (pytest plugin)

pytest-in-docker
last release: Feb 09, 2026, *status*: 3 - Alpha, *requires*: pytest>=9.0.2

Seamlessly run pytest tests inside docker containers

pytest-infinity
last release: Jun 09, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.0.0

pytest-influx
last release: Oct 16, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.3.3

Pytest plugin for managing your influx instance between test runs

pytest-influxdb
last release: Apr 20, 2021, *status*: N/A, *requires*: N/A

Plugin for influxdb and pytest integration.

pytest-info-collector
last release: May 26, 2019, *status*: 3 - Alpha, *requires*: N/A

pytest plugin to collect information from tests

pytest-info-plugin
last release: Sep 14, 2023, *status*: N/A, *requires*: N/A

Get executed interface information in pytest interface automation framework

pytest-informative-node
last release: Apr 25, 2019, *status*: 4 - Beta, *requires*: N/A

display more node ininformation.

pytest-infrahouse
last release: Mar 17, 2026, *status*: 4 - Beta, *requires*: pytest~=9.0

A set of fixtures to use with pytest

pytest-infrastructure
last release: Apr 12, 2020, *status*: 4 - Beta, *requires*: N/A

pytest stack validation prior to testing executing

pytest-ini
last release: Apr 26, 2022, *status*: N/A, *requires*: N/A

Reuse pytest.ini to store env variables

pytest-initry
last release: Apr 30, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.1.1

Plugin for sending automation test data from Pytest to the initry

pytest-inject
last release: Nov 25, 2025, *status*: N/A, *requires*: pytest>=6.0.0

A pytest plugin that allows you to inject arguments into fixtures and parametrized tests using pytest command-line options.

pytest-inline
last release: Oct 24, 2024, *status*: 4 - Beta, *requires*: pytest<9.0,>=7.0

A pytest plugin for writing inline tests

pytest-inline-snapshot
last release: Nov 09, 2025, *status*: N/A, *requires*: N/A

inline-snapshot is the package you are looking for

pytest-inline-tdd
last release: Mar 09, 2026, *status*: 4 - Beta, *requires*: pytest<9.0,>=7.0

A pytest plugin for writing inline tests

pytest-inmanta
last release: Nov 18, 2025, *status*: 5 - Production/Stable, *requires*: pytest

A py.test plugin providing fixtures to simplify inmanta modules testing.

pytest-inmanta-extensions
last release: Mar 03, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Inmanta tests package

pytest-inmanta-lsm
last release: Feb 12, 2026, *status*: 5 - Production/Stable, *requires*: N/A

Common fixtures used in inmanta LSM related modules

pytest-inmanta-srlinux
last release: Apr 22, 2025, *status*: 3 - Alpha, *requires*: N/A

Pytest library to facilitate end to end testing of inmanta projects

pytest-inmanta-yang
last release: Oct 28, 2025, *status*: 4 - Beta, *requires*: pytest

Common fixtures used in inmanta yang related modules

pytest-Inomaly
last release: Feb 13, 2018, *status*: 4 - Beta, *requires*: N/A

A simple image diff plugin for pytest

pytest-in-robotframework
last release: Nov 23, 2024, *status*: N/A, *requires*: pytest

The extension enables easy execution of pytest tests within the Robot Framework environment.

pytest-insper
last release: Mar 21, 2024, *status*: N/A, *requires*: pytest

Pytest plugin for courses at Insper

pytest-insta
last release: Nov 22, 2025, *status*: N/A, *requires*: pytest>=9.0.0

A practical snapshot testing plugin for pytest

pytest-instafail
last release: Mar 31, 2023, *status*: 4 - Beta, *requires*: pytest (>=5)

pytest plugin to show failures instantly

pytest-instrument
last release: Apr 05, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=5.1.0)

pytest plugin to instrument tests

pytest-insubprocess
last release: Dec 08, 2025, *status*: 4 - Beta, *requires*: pytest>=7.4

A pytest plugin to execute test cases in a subprocess

pytest-integration
last release: Nov 17, 2022, *status*: N/A, *requires*: N/A

Organizing pytests by integration or not

pytest-integration-mark
last release: May 22, 2023, *status*: N/A, *requires*: pytest (>=5.2)

Automatic integration test marking and excluding plugin for pytest

pytest-intent
last release: Dec 17, 2025, *status*: N/A, *requires*: pytest<10.0.0,>=9.0.0

A pytest plugin for tracking requirement coverage.

pytest-interactive
last release: Nov 30, 2017, *status*: 3 - Alpha, *requires*: N/A

A pytest plugin for console based interactive test selection just after the collection phase

pytest-intercept-remote
last release: May 24, 2021, *status*: 4 - Beta, *requires*: pytest (>=4.6)

Pytest plugin for intercepting outgoing connection requests during pytest run.

pytest-interface-tester
last release: Feb 11, 2026, *status*: 4 - Beta, *requires*: pytest

Pytest plugin for checking charm relation interface protocol compliance.

pytest-invenio
last release: Mar 16, 2026, *status*: 5 - Production/Stable, *requires*: pytest<9.0.0,>=6

Pytest fixtures for Invenio.

pytest-involve
last release: Feb 02, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Run tests covering a specific file or changeset

pytest-iovis
last release: Nov 06, 2024, *status*: 4 - Beta, *requires*: pytest>=7.1.0

A Pytest plugin to enable Jupyter Notebook testing with Papermill

pytest-ipdb
last release: Mar 20, 2013, *status*: 2 - Pre-Alpha, *requires*: N/A

A py.test plug-in to enable drop to ipdb debugger on test failure.

pytest-ipso
last release: Mar 24, 2026, *status*: N/A, *requires*: pytest

pytest plugin for running ipso notebook cell tests

pytest-ipynb
last release: Jan 29, 2019, *status*: 3 - Alpha, *requires*: N/A

THIS PROJECT IS ABANDONED

pytest-ipynb2
last release: Mar 09, 2025, *status*: N/A, *requires*: pytest

Pytest plugin to run tests in Jupyter Notebooks

pytest-ipywidgets
last release: Feb 24, 2026, *status*: N/A, *requires*: pytest

pytest-isolate
last release: Sep 08, 2025, *status*: 4 - Beta, *requires*: pytest

Run pytest tests in isolated subprocesses

pytest-isolated
last release: Mar 04, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

Run marked pytest tests in grouped subprocesses (cross-platform).

pytest-isolate-mpi
last release: Feb 24, 2025, *status*: 4 - Beta, *requires*: pytest>=5

pytest-isolate-mpi allows for MPI-parallel tests being executed in a segfault and MPI_Abort safe manner

pytest-isort
last release: Mar 05, 2024, *status*: 5 - Production/Stable, *requires*: pytest (>=5.0)

py.test plugin to check import ordering using isort

pytest-it
last release: Jan 29, 2024, *status*: 4 - Beta, *requires*: N/A

Pytest plugin to display test reports as a plaintext spec, inspired by Rspec: https://github.com/mattduck/pytest-it.

pytest-item-dict
last release: Nov 14, 2024, *status*: 4 - Beta, *requires*: pytest>=8.3.0

Get a hierarchical dict of session.items

pytest-iterassert
last release: May 11, 2020, *status*: 3 - Alpha, *requires*: N/A

Nicer list and iterable assertion messages for pytest

pytest-iteration
last release: Aug 22, 2024, *status*: N/A, *requires*: pytest

Add iteration mark for tests

pytest-iters
last release: May 24, 2022, *status*: N/A, *requires*: N/A

A contextmanager pytest fixture for handling multiple mock iters

pytest_jar_yuan
last release: Dec 12, 2022, *status*: N/A, *requires*: N/A

A allure and pytest used package

pytest-jasmine
last release: Nov 04, 2017, *status*: 1 - Planning, *requires*: N/A

Run jasmine tests from your pytest test suite

pytest-jax-bench
last release: Apr 06, 2026, *status*: N/A, *requires*: pytest>=7

Pytest plugin to profile jitted JAX functions (compile time, runtime, memory).

pytest-jelastic
last release: Nov 16, 2022, *status*: N/A, *requires*: pytest (>=7.2.0,<8.0.0)

Pytest plugin defining the necessary command-line options to pass to pytests testing a Jelastic environment.

pytest-jest
last release: May 22, 2018, *status*: 4 - Beta, *requires*: pytest (>=3.3.2)

A custom jest-pytest oriented Pytest reporter

pytest-jinja
last release: Oct 04, 2022, *status*: 3 - Alpha, *requires*: pytest (>=6.2.5,<7.0.0)

A plugin to generate customizable jinja-based HTML reports in pytest

pytest-jinja-check
last release: Mar 14, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Pytest plugin to lint Jinja2 templates in FastAPI applications

pytest-jira
last release: Mar 19, 2026, *status*: 4 - Beta, *requires*: pytest>=2.2.4

py.test JIRA integration plugin, using markers

pytest-jira-xfail
last release: Jul 09, 2024, *status*: N/A, *requires*: pytest>=7.2.0

Plugin skips (xfail) tests if unresolved Jira issue(s) linked

pytest-jira-xray
last release: Oct 11, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.4

pytest plugin to integrate tests with JIRA XRAY

pytest-job-selection
last release: Jan 30, 2023, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A pytest plugin for load balancing test suites

pytest-jobserver
last release: Feb 02, 2026, *status*: 5 - Production/Stable, *requires*: pytest

Limit parallel tests with posix jobserver.

pytest-joke
last release: Oct 08, 2019, *status*: 4 - Beta, *requires*: pytest (>=4.2.1)

Test failures are better served with humor.

pytest-jscov
last release: Apr 04, 2026, *status*: N/A, *requires*: pytest

Pytest plugin for JavaScript coverage via Playwright CDP

pytest-json
last release: Jan 18, 2016, *status*: 4 - Beta, *requires*: N/A

Generate JSON test reports

pytest-json-ctrf
last release: Mar 31, 2026, *status*: N/A, *requires*: pytest>6.0.0

Pytest plugin to generate json report in CTRF (Common Test Report Format)

pytest-json-fixtures
last release: Mar 14, 2023, *status*: 4 - Beta, *requires*: N/A

JSON output for the –fixtures flag

pytest-jsonlint
last release: Aug 04, 2016, *status*: N/A, *requires*: N/A

UNKNOWN

pytest-json-report
last release: Mar 15, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.8.0)

A pytest plugin to report test results as JSON files

pytest-json-report-wip
last release: Jul 23, 2025, *status*: 4 - Beta, *requires*: pytest >=3.8.0

A pytest plugin to report test results as JSON files

pytest-jsonschema
last release: Nov 07, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=6.2.0

A pytest plugin to perform JSONSchema validations

pytest-jsonschema-snapshot
last release: Mar 29, 2026, *status*: N/A, *requires*: pytest

Pytest plugin for automatic JSON Schema generation and validation from examples

pytest-jtr
last release: Jul 21, 2024, *status*: N/A, *requires*: pytest<8.0.0,>=7.1.2

pytest plugin supporting json test report output

pytest-jubilant
last release: Apr 07, 2026, *status*: N/A, *requires*: pytest>=8.3.5

Add your description here

pytest-junit-logging
last release: Nov 27, 2025, *status*: 4 - Beta, *requires*: pytest>=6.0

A pytest plugin for embedding log output into JUnit XML reports

pytest-junit-xray-xml
last release: Jan 01, 2025, *status*: 4 - Beta, *requires*: pytest

Export test results in an augmented JUnit format for usage with Xray ()

pytest-jupyter
last release: Oct 16, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0

A pytest plugin for testing Jupyter libraries and extensions.

pytest-jupyter-deploy
last release: Apr 13, 2026, *status*: 3 - Alpha, *requires*: pytest>=8.3.5

Pytest plugin for E2E testing of jupyter-deploy templates

pytest-jupyterhub
last release: Apr 25, 2023, *status*: 5 - Production/Stable, *requires*: pytest

A reusable JupyterHub pytest plugin

pytest-just
last release: Mar 22, 2026, *status*: 3 - Alpha, *requires*: pytest>=8.0.0

A pytest plugin for testing justfile recipes

pytest-jux
last release: Jan 08, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.4

A pytest plugin for signing and publishing JUnit XML test reports to the Jux REST API

pytest-k8s
last release: Jul 07, 2025, *status*: N/A, *requires*: pytest>=8.4.1

Kubernetes-based testing for pytest

pytest-kafka
last release: Aug 14, 2024, *status*: N/A, *requires*: pytest

Zookeeper, Kafka server, and Kafka consumer fixtures for Pytest

pytest-kafka-broker
last release: Apr 03, 2026, *status*: N/A, *requires*: N/A

Pytest plugin to run a single-broker Kafka cluster

pytest-kafkavents
last release: Sep 08, 2021, *status*: 4 - Beta, *requires*: pytest

A plugin to send pytest events to Kafka

pytest-kairos
last release: Aug 08, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=5.0.0

Pytest plugin with random number generation, reproducibility, and test repetition

pytest-kasima
last release: Jan 26, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=7.2.1,<8.0.0)

Display horizontal lines above and below the captured standard output for easy viewing.

pytest-kedge
last release: Jan 10, 2026, *status*: N/A, *requires*: pytest>=7.0.0

Agent-friendly structured test data collector for pytest

pytest-keep-together
last release: Dec 07, 2022, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin to customize test ordering by running all 'related' tests together

pytest-kexi
last release: Apr 29, 2022, *status*: N/A, *requires*: pytest (>=7.1.2,<8.0.0)

pytest-keyring
last release: Apr 10, 2026, *status*: N/A, *requires*: pytest>=8.0.2

A Pytest plugin to access the system's keyring to provide credentials for tests

pytest-kind
last release: Nov 30, 2022, *status*: 5 - Production/Stable, *requires*: N/A

Kubernetes test support with KIND for pytest

pytest-kivy
last release: Jul 06, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.6)

Kivy GUI tests fixtures using pytest

pytest-knows
last release: Aug 22, 2014, *status*: N/A, *requires*: N/A

A pytest plugin that can automaticly skip test case based on dependence info calculated by trace

pytest-konira
last release: Oct 09, 2011, *status*: N/A, *requires*: N/A

Run Konira DSL tests with py.test

pytest-kookit
last release: Sep 10, 2024, *status*: N/A, *requires*: N/A

Your simple but kooky integration testing with pytest

pytest-koopmans
last release: Nov 21, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A plugin for testing the koopmans package

pytest-krtech-common
last release: Nov 28, 2016, *status*: 4 - Beta, *requires*: N/A

pytest krtech common library

pytest-kubernetes
last release: Apr 14, 2026, *status*: N/A, *requires*: pytest<10.0.0,>=9.0.0

A lightweight pytest plugin for managing local Kubernetes clusters (minikube, k3d, kind)

pytest_kustomize
last release: Dec 08, 2025, *status*: N/A, *requires*: N/A

Parse and validate kustomize output

pytest-kuunda
last release: Feb 25, 2024, *status*: 4 - Beta, *requires*: pytest >=6.2.0

pytest plugin to help with test data setup for PySpark tests

pytest-kwparametrize
last release: Jan 22, 2021, *status*: N/A, *requires*: pytest (>=6)

Alternate syntax for @pytest.mark.parametrize with test cases as dictionaries and default value fallbacks

pytest-lambda
last release: May 27, 2024, *status*: 5 - Production/Stable, *requires*: pytest<9,>=3.6

Define pytest fixtures with lambda functions.

pytest-lamp
last release: Jan 06, 2017, *status*: 3 - Alpha, *requires*: N/A

pytest-langchain
last release: Feb 26, 2023, *status*: N/A, *requires*: pytest

Pytest-style test runner for langchain agents

pytest-language-server
last release: Apr 05, 2026, *status*: 4 - Beta, *requires*: N/A

A blazingly fast Language Server Protocol implementation for pytest

pytest-lark
last release: Nov 05, 2023, *status*: N/A, *requires*: N/A

Create fancy and clear HTML test reports.

pytest-latin-hypercube
last release: Jun 26, 2025, *status*: N/A, *requires*: pytest

Implementation of Latin Hypercube Sampling for pytest.

pytest-launchable
last release: Apr 05, 2023, *status*: N/A, *requires*: pytest (>=4.2.0)

Launchable Pytest Plugin

pytest-layab
last release: Oct 05, 2020, *status*: 5 - Production/Stable, *requires*: N/A

Pytest fixtures for layab.

pytest-lazy-fixture
last release: Feb 01, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.2.5)

It helps to use fixtures in pytest.mark.parametrize

pytest-lazy-fixtures
last release: Sep 16, 2025, *status*: N/A, *requires*: pytest>=7

Allows you to use fixtures in @pytest.mark.parametrize.

pytest-ldap
last release: Aug 18, 2020, *status*: N/A, *requires*: pytest

python-ldap fixtures for pytest

pytest-leak-finder
last release: Dec 19, 2025, *status*: 4 - Beta, *requires*: pytest>=3.5.0

Find the test that's leaking before the one that fails

pytest-leaks
last release: Nov 27, 2019, *status*: 1 - Planning, *requires*: N/A

A pytest plugin to trace resource leaks.

pytest-leaping
last release: Mar 27, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A simple plugin to use with pytest

pytest-leela
last release: Apr 09, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Type-aware mutation testing for Python — fast, opinionated, pytest-native

pytest-leo-interface
last release: Mar 19, 2025, *status*: N/A, *requires*: N/A

Pytest extension tool for leo projects.

pytest-level
last release: Oct 21, 2019, *status*: N/A, *requires*: pytest

Select tests of a given level or lower

pytest-lf-skip
last release: Feb 27, 2026, *status*: 4 - Beta, *requires*: pytest>=8.3.5

A pytest plugin which makes `–last-failed` skip instead of deselect tests.

pytest-libfaketime
last release: Apr 12, 2024, *status*: 4 - Beta, *requires*: pytest>=3.0.0

A python-libfaketime plugin for pytest

pytest-libiio
last release: Apr 03, 2026, *status*: N/A, *requires*: pytest>=3.5.0

A pytest plugin for testing libiio based devices

pytest-libnotify
last release: Apr 02, 2021, *status*: 3 - Alpha, *requires*: pytest

Pytest plugin that shows notifications about the test run

pytest-ligo
last release: Jan 16, 2020, *status*: 4 - Beta, *requires*: N/A

pytest-lineno
last release: Dec 04, 2020, *status*: N/A, *requires*: pytest

A pytest plugin to show the line numbers of test functions

pytest-line-profiler
last release: Aug 10, 2023, *status*: 4 - Beta, *requires*: pytest >=3.5.0

Profile code executed by pytest

pytest-line-profiler-apn
last release: Dec 05, 2022, *status*: N/A, *requires*: pytest (>=3.5.0)

Profile code executed by pytest

pytest-line-runner
last release: Feb 08, 2026, *status*: N/A, *requires*: N/A

Run pytest tests by line number instead of exact test name

pytest-lisa
last release: Jan 21, 2021, *status*: 3 - Alpha, *requires*: pytest (>=6.1.2,<7.0.0)

Pytest plugin for organizing tests.

pytest-listener
last release: Nov 29, 2024, *status*: 5 - Production/Stable, *requires*: pytest

A simple network listener

pytest-litf
last release: Jan 18, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.1.1)

A pytest plugin that stream output in LITF format

pytest-litter
last release: Nov 23, 2023, *status*: 4 - Beta, *requires*: pytest >=6.1

Pytest plugin which verifies that tests do not modify file trees.

pytest-live
last release: Mar 08, 2020, *status*: N/A, *requires*: pytest

Live results for pytest

pytest-liveview
last release: Mar 09, 2026, *status*: N/A, *requires*: pytest>=7.0

Pytest plugin that shows a real-time test dashboard in a local web server

pytest-llm
last release: Oct 03, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

pytest-llm: A pytest plugin for testing LLM outputs with success rate thresholds.

pytest-llm-agent
last release: Dec 16, 2025, *status*: N/A, *requires*: pytest>=9.0.2

LLM Agent for working with pytest

pytest-llm-assert
last release: Mar 31, 2026, *status*: 3 - Alpha, *requires*: pytest>=9.0

Simple LLM-powered assertions for any pytest test

pytest-llmeval
last release: Mar 19, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A pytest plugin to evaluate/benchmark LLM prompts

pytest-llm-report
last release: Jan 21, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

Human-friendly pytest test reports with optional LLM annotations

pytest-llm-rubric
last release: Apr 07, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.2

A pytest plugin for rubric-based LLM-as-judge testing with auto-discovery and preflight

pytest-llmtest
last release: Mar 08, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0; extra == "dev"

The pytest for LLMs — fast, Pydantic-based assertions for AI applications

pytest-lobster
last release: Jul 26, 2025, *status*: N/A, *requires*: pytest>=7.0

Pytest to generate lobster tracing files

pytest-local-badge
last release: Jan 15, 2023, *status*: N/A, *requires*: pytest (>=6.1.0)

Generate local badges (shields) reporting your test suite status.

pytest-localftpserver
last release: Nov 16, 2025, *status*: 5 - Production/Stable, *requires*: pytest

A PyTest plugin which provides an FTP fixture for your tests

pytest-localserver
last release: Nov 24, 2025, *status*: 4 - Beta, *requires*: N/A

pytest plugin to test server connections locally.

pytest-localstack
last release: Jun 07, 2023, *status*: 4 - Beta, *requires*: pytest (>=6.0.0,<7.0.0)

Pytest plugin for AWS integration tests

pytest-lock
last release: Feb 03, 2024, *status*: N/A, *requires*: pytest (>=7.4.3,<8.0.0)

pytest-lock is a pytest plugin that allows you to "lock" the results of unit tests, storing them in a local cache. This is particularly useful for tests that are resource-intensive or don't need to be run every time. When the tests are run subsequently, pytest-lock will compare the current results with the locked results and issue a warning if there are any discrepancies.

pytest-lockable
last release: Sep 08, 2025, *status*: 5 - Production/Stable, *requires*: pytest

lockable resource plugin for pytest

pytest-locker
last release: Dec 20, 2024, *status*: N/A, *requires*: pytest>=5.4

Used to lock object during testing. Essentially changing assertions from being hard coded to asserting that nothing changed

pytest-loco
last release: Mar 08, 2026, *status*: 3 - Alpha, *requires*: pytest<10.0.0,>=9.0.2

Another one YAML-based DSL for testing

pytest-loco-allure
last release: Mar 08, 2026, *status*: 3 - Alpha, *requires*: N/A

Allure support for pytest-loco

pytest-loco-http
last release: Feb 25, 2026, *status*: 3 - Alpha, *requires*: N/A

HTTP support for pytest-loco

pytest-loco-json
last release: Feb 25, 2026, *status*: 3 - Alpha, *requires*: N/A

JSON support for pytest-loco

pytest-log
last release: Aug 15, 2021, *status*: N/A, *requires*: pytest (>=3.8)

print log

pytest-logbook
last release: Nov 23, 2015, *status*: 5 - Production/Stable, *requires*: pytest (>=2.8)

py.test plugin to capture logbook log messages

pytest-logdog
last release: Jun 15, 2021, *status*: 1 - Planning, *requires*: pytest (>=6.2.0)

Pytest plugin to test logging

pytest-logfest
last release: Jul 21, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Pytest plugin providing three logger fixtures with basic or full writing to log files

pytest-log-filter
last release: Nov 13, 2025, *status*: N/A, *requires*: pytest

Ignore some loggers' log for pytest

pytest-logger
last release: Mar 10, 2024, *status*: 5 - Production/Stable, *requires*: pytest (>=3.2)

Plugin configuring handlers for loggers from Python logging module.

pytest-logger-db
last release: Sep 14, 2025, *status*: N/A, *requires*: N/A

Add your description here

pytest-logging
last release: Nov 04, 2015, *status*: 4 - Beta, *requires*: N/A

Configures logging and allows tweaking the log level with a py.test flag

pytest-logging-end-to-end-test-tool
last release: Sep 23, 2022, *status*: N/A, *requires*: pytest (>=7.1.2,<8.0.0)

pytest-logging-strict
last release: May 20, 2025, *status*: 3 - Alpha, *requires*: pytest

pytest fixture logging configured from packaged YAML

pytest-logikal
last release: Mar 16, 2026, *status*: 5 - Production/Stable, *requires*: pytest==9.0.2

Common testing environment

pytest-log-report
last release: Dec 26, 2019, *status*: N/A, *requires*: N/A

Package for creating a pytest test run reprot

pytest-logscanner
last release: Sep 30, 2024, *status*: 4 - Beta, *requires*: pytest>=8.2.2

Pytest plugin for logscanner (A logger for python logging outputting to easily viewable (and filterable) html files. Good for people not grep savey, and color higlighting and quickly changing filters might even bye useful for commandline wizards.)

pytest-loguru
last release: Mar 20, 2024, *status*: 5 - Production/Stable, *requires*: pytest; extra == "test"

Pytest Loguru

pytest-loop
last release: Oct 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

pytest plugin for looping tests

pytest-lsp
last release: Oct 25, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=8.0

A pytest plugin for end-to-end testing of language servers

pytest-lw-realtime-result
last release: Mar 13, 2025, *status*: N/A, *requires*: pytest>=3.5.0

Pytest plugin to generate realtime test results to a file

pytest-manifest
last release: Apr 07, 2025, *status*: N/A, *requires*: pytest

PyTest plugin for recording and asserting against a manifest file

pytest-manual-marker
last release: Aug 04, 2022, *status*: 3 - Alpha, *requires*: pytest>=7

pytest marker for marking manual tests

pytest-mark-ac
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: pytest<10,>=8.4

Provides a marker to reference acceptance criteria from PyTest tests through annotations

pytest-mark-count
last release: Nov 13, 2024, *status*: 4 - Beta, *requires*: pytest>=8.0.0

Get a count of the number of tests marked, unmarked, and unique tests if tests have multiple markers

pytest-markdir
last release: Feb 01, 2026, *status*: N/A, *requires*: pytest<10,>=8.0

pytest-markdoctest
last release: Jul 22, 2022, *status*: 4 - Beta, *requires*: pytest (>=6)

A pytest plugin to doctest your markdown files

pytest-markdown
last release: Jan 15, 2021, *status*: 4 - Beta, *requires*: pytest (>=6.0.1,<7.0.0)

Test your markdown docs with pytest

pytest-markdown-docs
last release: Mar 23, 2026, *status*: N/A, *requires*: pytest>=7.0.0

Run markdown code fences through pytest

pytest-markdown-report
last release: Jan 10, 2026, *status*: N/A, *requires*: pytest>=7.0

Token-efficient markdown test reports for LLM-based TDD agents

pytest-marker-bugzilla
last release: Apr 02, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=2.2.4

py.test bugzilla integration plugin, using markers

pytest-markers-presence
last release: Oct 30, 2024, *status*: 4 - Beta, *requires*: pytest>=6.0

A simple plugin to detect missed pytest tags and markers"

pytest-mark-filter
last release: May 11, 2025, *status*: N/A, *requires*: pytest>=8.3.0

Filter pytest marks by name using match kw

pytest-markfiltration
last release: Nov 08, 2011, *status*: 3 - Alpha, *requires*: N/A

UNKNOWN

pytest-mark-integration
last release: Jan 13, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Pytest plugin for automatic integration test marking and management

pytest-mark-manage
last release: Aug 15, 2024, *status*: N/A, *requires*: pytest

用例标签化管理

pytest-mark-no-py3
last release: May 17, 2019, *status*: N/A, *requires*: pytest

pytest plugin and bowler codemod to help migrate tests to Python 3

pytest-marks
last release: Nov 23, 2012, *status*: 3 - Alpha, *requires*: N/A

UNKNOWN

pytest-mask-secrets
last release: Dec 17, 2025, *status*: N/A, *requires*: N/A

Pytest plugin to hide sensitive data in test reports

pytest-matcher
last release: Aug 07, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Easy way to match captured `pytest` output against expectations stored in files

pytest-matchers
last release: Dec 19, 2025, *status*: N/A, *requires*: pytest<10.0,>=7.0

Matchers for pytest

pytest-match-skip
last release: May 15, 2019, *status*: 4 - Beta, *requires*: pytest (>=4.4.1)

Skip matching marks. Matches partial marks using wildcards.

pytest-mat-report
last release: Jan 20, 2021, *status*: N/A, *requires*: N/A

this is report

pytest-matrix
last release: Jun 24, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=5.4.3,<6.0.0)

Provide tools for generating tests from combinations of fixtures.

pytest-maxcov
last release: Sep 24, 2023, *status*: N/A, *requires*: pytest (>=7.4.0,<8.0.0)

Compute the maximum coverage available through pytest with the minimum execution time cost

pytest-max-warnings
last release: Oct 23, 2024, *status*: 4 - Beta, *requires*: pytest>=8.3.3

A Pytest plugin to exit non-zero exit code when the configured maximum warnings has been exceeded.

pytest-maybe-context
last release: Apr 16, 2023, *status*: N/A, *requires*: pytest (>=7,<8)

Simplify tests with warning and exception cases.

pytest-maybe-raises
last release: May 27, 2022, *status*: N/A, *requires*: pytest ; extra == 'dev'

Pytest fixture for optional exception testing.

pytest-mccabe
last release: Jul 22, 2020, *status*: 3 - Alpha, *requires*: pytest (>=5.4.0)

pytest plugin to run the mccabe code complexity checker.

pytest-mcp
last release: Jul 07, 2025, *status*: N/A, *requires*: pytest>=8.4.0

Pytest-style framework for evaluating Model Context Protocol (MCP) servers.

pytest-mcp-tools
last release: Mar 12, 2026, *status*: N/A, *requires*: pytest>=7.0.0; extra == "test"

pytest-md
last release: Jul 11, 2019, *status*: 3 - Alpha, *requires*: pytest (>=4.2.1)

Plugin for generating Markdown reports for pytest results

pytest-md-report
last release: May 02, 2025, *status*: 4 - Beta, *requires*: pytest!=6.0.0,<9,>=3.3.2

A pytest plugin to generate test outcomes reports with markdown table format.

pytest-meilisearch
last release: Oct 08, 2024, *status*: N/A, *requires*: pytest>=7.4.3

Pytest helpers for testing projects using Meilisearch

pytest-memlog
last release: May 03, 2023, *status*: N/A, *requires*: pytest (>=7.3.0,<8.0.0)

Log memory usage during tests

pytest-memprof
last release: Mar 29, 2019, *status*: 4 - Beta, *requires*: N/A

Estimates memory consumption of test functions

pytest-memray
last release: Aug 18, 2025, *status*: N/A, *requires*: pytest>=7.2

A simple plugin to use with pytest

pytest-menu
last release: Oct 04, 2017, *status*: 3 - Alpha, *requires*: pytest (>=2.4.2)

A pytest plugin for console based interactive test selection just after the collection phase

pytest-mercurial
last release: Nov 21, 2020, *status*: 1 - Planning, *requires*: N/A

pytest plugin to write integration tests for projects using Mercurial Python internals

pytest-mergify
last release: Apr 07, 2026, *status*: N/A, *requires*: pytest>=6.0.0

Pytest plugin for Mergify

pytest-mesh
last release: Aug 05, 2022, *status*: N/A, *requires*: pytest (==7.1.2)

pytest_mesh插件

pytest-message
last release: Aug 04, 2022, *status*: N/A, *requires*: pytest (>=6.2.5)

Pytest plugin for sending report message of marked tests execution

pytest-messenger
last release: Nov 24, 2022, *status*: 5 - Production/Stable, *requires*: N/A

Pytest to Slack reporting plugin

pytest-metadata
last release: Feb 12, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

pytest plugin for test session metadata

pytest-metaexport
last release: Jun 24, 2025, *status*: N/A, *requires*: pytest>=7.1.0

Pytest plugin for exporting custom test metadata to JSON.

pytest-metrics
last release: Apr 04, 2020, *status*: N/A, *requires*: pytest

Custom metrics report for pytest

pytest-mfd-config
last release: Jul 11, 2025, *status*: N/A, *requires*: pytest<9,>=7.2.1

Pytest Plugin that handles test and topology configs and all their belongings like helper fixtures.

pytest-mfd-logging
last release: Nov 14, 2025, *status*: N/A, *requires*: pytest<9,>=7.2.1

Module for handling PyTest logging.

pytest-mh
last release: Oct 16, 2025, *status*: N/A, *requires*: pytest

Pytest multihost plugin

pytest-mimesis
last release: Mar 21, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=4.2)

Mimesis integration with the pytest test runner

pytest-mimic
last release: Apr 24, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Easily record function calls while testing

pytest-minecraft
last release: Apr 06, 2022, *status*: N/A, *requires*: pytest (>=6.0.1)

A pytest plugin for running tests against Minecraft releases

pytest-mini
last release: Feb 06, 2023, *status*: N/A, *requires*: pytest (>=7.2.0,<8.0.0)

A plugin to test mp

pytest-minio-mock
last release: Aug 06, 2025, *status*: N/A, *requires*: pytest>=5.0.0

A pytest plugin for mocking Minio S3 interactions

pytest-mirror
last release: Jul 30, 2025, *status*: 4 - Beta, *requires*: N/A

A pluggy-based pytest plugin and CLI tool for ensuring your test suite mirrors your source code structure

pytest-missing-fixtures
last release: Oct 14, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Pytest plugin that creates missing fixtures

pytest-missing-modules
last release: Nov 17, 2025, *status*: N/A, *requires*: pytest>=8.3.2

Pytest plugin to easily fake missing modules

pytest-mitmproxy
last release: Nov 13, 2024, *status*: N/A, *requires*: pytest>=7.0

pytest plugin for mitmproxy tests

pytest-mitmproxy-plugin
last release: Apr 10, 2025, *status*: 4 - Beta, *requires*: pytest>=7.2.0

Use MITM Proxy in autotests with full control from code

pytest-ml
last release: May 04, 2019, *status*: 4 - Beta, *requires*: N/A

Test your machine learning!

pytest-mocha
last release: Apr 02, 2020, *status*: 4 - Beta, *requires*: pytest (>=5.4.0)

pytest plugin to display test execution output like a mochajs

pytest-mock
last release: Sep 16, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=6.2.5

Thin-wrapper around the mock package for easier use with pytest

pytest-mock-api
last release: Feb 13, 2019, *status*: 1 - Planning, *requires*: pytest (>=4.0.0)

A mock API server with configurable routes and responses available as a fixture.

pytest-mock-generator
last release: May 16, 2022, *status*: 5 - Production/Stable, *requires*: N/A

A pytest fixture wrapper for https://pypi.org/project/mock-generator

pytest-mock-helper
last release: Jan 24, 2018, *status*: N/A, *requires*: pytest

Help you mock HTTP call and generate mock code

pytest-mockito
last release: Feb 10, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6

Base fixtures for mockito

pytest-mockllm
last release: Dec 22, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Zero-config pytest plugin for mocking LLM APIs - OpenAI, Anthropic, Gemini, LangChain & more

pytest-mockredis
last release: Jan 02, 2018, *status*: 2 - Pre-Alpha, *requires*: N/A

An in-memory mock of a Redis server that runs in a separate thread. This is to be used for unit-tests that require a Redis database.

pytest-mock-resources
last release: Sep 17, 2025, *status*: N/A, *requires*: pytest>=1.0

A pytest plugin for easily instantiating reproducible mock resources.

pytest-mock-server
last release: Jan 09, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Mock server plugin for pytest

pytest-mockservers
last release: Mar 31, 2020, *status*: N/A, *requires*: pytest (>=4.3.0)

A set of fixtures to test your requests to HTTP/UDP servers

pytest-mocktcp
last release: Oct 11, 2022, *status*: N/A, *requires*: pytest

A pytest plugin for testing TCP clients

pytest-mock-unity-catalog
last release: Mar 12, 2026, *status*: N/A, *requires*: pytest

Unity Catalog pyspark fixtures

pytest-modalt
last release: Feb 27, 2024, *status*: 4 - Beta, *requires*: pytest >=6.2.0

Massively distributed pytest runs using modal.com

pytest-model-lib
last release: Feb 22, 2026, *status*: N/A, *requires*: N/A

pytest plugin for model-lib

pytest-modern
last release: Aug 19, 2025, *status*: 4 - Beta, *requires*: pytest>=8

A more modern pytest

pytest-modified-env
last release: Jan 29, 2022, *status*: 4 - Beta, *requires*: N/A

Pytest plugin to fail a test if it leaves modified `os.environ` afterwards.

pytest-modifyjunit
last release: Jan 10, 2019, *status*: N/A, *requires*: N/A

Utility for adding additional properties to junit xml for IDM QE

pytest-molecule
last release: Mar 29, 2022, *status*: 5 - Production/Stable, *requires*: pytest (>=7.0.0)

PyTest Molecule Plugin :: discover and run molecule tests

pytest-molecule-JC
last release: Jul 18, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=7.0.0)

PyTest Molecule Plugin :: discover and run molecule tests

pytest-mongo
last release: Feb 04, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.4

MongoDB process and client fixtures plugin for Pytest.

pytest-mongodb
last release: May 16, 2023, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for MongoDB fixtures

pytest-mongodb-nono
last release: Jan 07, 2025, *status*: N/A, *requires*: N/A

pytest plugin for MongoDB

pytest-mongodb-ry
last release: Sep 25, 2025, *status*: N/A, *requires*: N/A

pytest plugin for MongoDB

pytest-mongo-docker
last release: Mar 05, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.4

A tiny plugin for pytest which runs MongoDB in Docker

pytest-monitor
last release: Jun 25, 2023, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin for analyzing resource usage.

pytest-monkeyplus
last release: Sep 18, 2012, *status*: 5 - Production/Stable, *requires*: N/A

pytest's monkeypatch subclass with extra functionalities

pytest-monkeytype
last release: Jul 29, 2020, *status*: 4 - Beta, *requires*: N/A

pytest-monkeytype: Generate Monkeytype annotations from your pytest tests.

pytest-moto
last release: Aug 28, 2015, *status*: 1 - Planning, *requires*: N/A

Fixtures for integration tests of AWS services,uses moto mocking library.

pytest-moto-fixtures
last release: Nov 17, 2025, *status*: 1 - Planning, *requires*: pytest<9.1,>=8.3; extra == "pytest"

Fixtures for testing code that interacts with AWS

pytest-motor
last release: Jul 21, 2021, *status*: 3 - Alpha, *requires*: pytest

A pytest plugin for motor, the non-blocking MongoDB driver.

pytest-mp
last release: May 23, 2018, *status*: 4 - Beta, *requires*: pytest

A test batcher for multiprocessed Pytest runs

pytest-mpi
last release: Jan 08, 2022, *status*: 3 - Alpha, *requires*: pytest

pytest plugin to collect information from tests

pytest-mpiexec
last release: Jul 29, 2024, *status*: 3 - Alpha, *requires*: pytest

pytest plugin for running individual tests with mpiexec

pytest-mpi-tmweigand
last release: Dec 27, 2025, *status*: 3 - Alpha, *requires*: pytest

forked pytest plugin to collect information from tests

pytest-mpl
last release: Mar 25, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6.2.5

pytest plugin to help with testing figures output from Matplotlib

pytest-mpl-oggm
last release: Mar 09, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6.2.5

pytest plugin to help with testing figures output from Matplotlib - OGGM fork

pytest-mproc
last release: Mar 27, 2026, *status*: 4 - Beta, *requires*: pytest>=6

low-startup-overhead, scalable, distributed-testing pytest plugin

pytest-mqtt
last release: Jan 28, 2026, *status*: 5 - Production/Stable, *requires*: pytest<10; extra == "test"

pytest-mqtt supports testing systems based on MQTT

pytest-multihost
last release: Apr 07, 2020, *status*: 4 - Beta, *requires*: N/A

Utility for writing multi-host tests for pytest

pytest-multilog
last release: Dec 28, 2025, *status*: N/A, *requires*: pytest

Multi-process logs handling and other helpers for pytest

pytest-multithreading
last release: Aug 05, 2024, *status*: N/A, *requires*: N/A

a pytest plugin for th and concurrent testing

pytest-multithreading-allure
last release: Nov 25, 2022, *status*: N/A, *requires*: N/A

pytest_multithreading_allure

pytest-mutagen
last release: Jul 24, 2020, *status*: N/A, *requires*: pytest (>=5.4)

Add the mutation testing feature to pytest

pytest-my-cool-lib
last release: Nov 02, 2023, *status*: N/A, *requires*: pytest (>=7.1.3,<8.0.0)

pytest-my-plugin
last release: Jan 27, 2025, *status*: N/A, *requires*: pytest>=6.0

A pytest plugin that does awesome things

pytest-mypy
last release: Apr 02, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.0

A Pytest Plugin for Mypy

pytest-mypyd
last release: Aug 20, 2019, *status*: 4 - Beta, *requires*: pytest (<4.7,>=2.8) ; python_version < "3.5"

Mypy static type checker plugin for Pytest

pytest-mypy-plugins
last release: Apr 17, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

pytest plugin for writing tests for mypy plugins

pytest-mypy-plugins-shim
last release: Feb 14, 2025, *status*: N/A, *requires*: pytest>=6.0.0

Substitute for "pytest-mypy-plugins" for Python implementations which aren't supported by mypy.

pytest-mypy-runner
last release: Apr 23, 2024, *status*: N/A, *requires*: pytest>=8.0

Run the mypy static type checker as a pytest test case

pytest-mypy-testing
last release: Jan 26, 2026, *status*: N/A, *requires*: pytest>=8

Pytest plugin to check mypy output

pytest-mysql
last release: Apr 12, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.4

MySQL process and client fixtures for pytest

pytest-nb
last release: Jul 26, 2025, *status*: N/A, *requires*: pytest==8.4.1

Seedable Jupyter Notebook testing tool

pytest-nb-as-test
last release: Feb 25, 2026, *status*: 4 - Beta, *requires*: pytest<9.1.0,>=7.0.0; python_version < "3.14"

Use notebooks as pytests. Keep your notebooks working.

pytest-nbgrader
last release: Mar 31, 2026, *status*: 3 - Alpha, *requires*: pytest>=8

Pytest plugin for using with nbgrader and generating test cases.

pytest-ndb
last release: Apr 28, 2024, *status*: N/A, *requires*: pytest

pytest notebook debugger

pytest-needle
last release: Dec 10, 2018, *status*: 4 - Beta, *requires*: pytest (<5.0.0,>=3.0.0)

pytest plugin for visual testing websites using selenium

pytest-neo
last release: Jan 08, 2022, *status*: 3 - Alpha, *requires*: pytest (>=6.2.0)

pytest-neo is a plugin for pytest that shows tests like screen of Matrix.

pytest-neon
last release: Feb 05, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

Pytest plugin for Neon database branch isolation in tests

pytest-neos
last release: Sep 10, 2024, *status*: 5 - Production/Stable, *requires*: pytest<8.0,>=7.2; extra == "dev"

Pytest plugin for neos

pytest-netconf
last release: Nov 03, 2025, *status*: N/A, *requires*: N/A

A pytest plugin that provides a mock NETCONF (RFC6241/RFC6242) server for local testing.

pytest-netdut
last release: Oct 09, 2025, *status*: N/A, *requires*: pytest>=3.5.0

"Automated software testing for switches using pytest"

pytest-network
last release: May 07, 2020, *status*: N/A, *requires*: N/A

A simple plugin to disable network on socket level.

pytest-network-endpoints
last release: Mar 06, 2022, *status*: N/A, *requires*: pytest

Network endpoints plugin for pytest

pytest-never-sleep
last release: May 05, 2021, *status*: 3 - Alpha, *requires*: pytest (>=3.5.1)

pytest plugin helps to avoid adding tests without mock `time.sleep`

pytest-nginx
last release: May 03, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=3.0.0

nginx fixture for pytest

pytest-nginx-iplweb
last release: Mar 01, 2019, *status*: 5 - Production/Stable, *requires*: N/A

nginx fixture for pytest - iplweb temporary fork

pytest-ngrok
last release: Jan 20, 2022, *status*: 3 - Alpha, *requires*: pytest

pytest-ngsfixtures
last release: Sep 06, 2019, *status*: 2 - Pre-Alpha, *requires*: pytest (>=5.0.0)

pytest ngs fixtures

pytest-nhsd-apim
last release: Feb 13, 2026, *status*: N/A, *requires*: pytest<9.0.0,>=8.2.0

Pytest plugin accessing NHSDigital's APIM proxies

pytest-nice
last release: May 04, 2019, *status*: 4 - Beta, *requires*: pytest

A pytest plugin that alerts user of failed test cases with screen notifications

pytest-nice-parametrize
last release: Apr 17, 2021, *status*: 5 - Production/Stable, *requires*: N/A

A small snippet for nicer PyTest's Parametrize

pytest_nlcov
last release: Aug 05, 2024, *status*: N/A, *requires*: N/A

Pytest plugin to get the coverage of the new lines (based on git diff) only

pytest-nocustom
last release: Aug 05, 2024, *status*: 5 - Production/Stable, *requires*: N/A

Run all tests without custom markers

pytest-node-dependency
last release: Apr 10, 2024, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for controlling execution flow

pytest-nodev
last release: Jul 21, 2016, *status*: 4 - Beta, *requires*: pytest (>=2.8.1)

Test-driven source code search for Python.

pytest-nogarbage
last release: Feb 24, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=4.6.0

Ensure a test produces no garbage

pytest-no-problem
last release: Jan 11, 2026, *status*: N/A, *requires*: pytest>=7.0

Pytest plugin to tell you when there's no problem

pytest-nose-attrib
last release: Aug 13, 2023, *status*: N/A, *requires*: N/A

pytest plugin to use nose @attrib marks decorators and pick tests based on attributes and partially uses nose-attrib plugin approach

pytest_notebook
last release: Nov 28, 2023, *status*: 4 - Beta, *requires*: pytest>=3.5.0

A pytest plugin for testing Jupyter Notebooks.

pytest-notice
last release: Nov 05, 2020, *status*: N/A, *requires*: N/A

Send pytest execution result email

pytest-notification
last release: Jun 19, 2020, *status*: N/A, *requires*: pytest (>=4)

A pytest plugin for sending a desktop notification and playing a sound upon completion of tests

pytest-notifier
last release: Jun 12, 2020, *status*: 3 - Alpha, *requires*: pytest

A pytest plugin to notify test result

pytest-notifier-plugin
last release: Dec 22, 2025, *status*: N/A, *requires*: pytest>=7.0

Pytest plugin для отправки нотификаций в различные каналы связи о статуе прохождения тестов.

pytest_notify
last release: Jul 05, 2017, *status*: N/A, *requires*: pytest>=3.0.0

Get notifications when your tests ends

pytest-notimplemented
last release: Aug 27, 2019, *status*: N/A, *requires*: pytest (>=5.1,<6.0)

Pytest markers for not implemented features and tests.

pytest-notion
last release: Aug 07, 2019, *status*: N/A, *requires*: N/A

A PyTest Reporter to send test runs to Notion.so

pytest-nunit
last release: Feb 26, 2024, *status*: 5 - Production/Stable, *requires*: N/A

A pytest plugin for generating NUnit3 test result XML output

pytest-oar
last release: May 12, 2025, *status*: N/A, *requires*: pytest>=6.0.1

PyTest plugin for the OAR testing framework

pytest-oarepo
last release: Apr 01, 2026, *status*: N/A, *requires*: pytest>=7.1.2; extra == "dev"

pytest-object-getter
last release: Jul 31, 2022, *status*: 5 - Production/Stable, *requires*: pytest

Import any object from a 3rd party module while mocking its namespace on demand.

pytest-ochrus
last release: Feb 21, 2018, *status*: 4 - Beta, *requires*: N/A

pytest results data-base and HTML reporter

pytest-odc
last release: Aug 04, 2023, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A pytest plugin for simplifying ODC database tests

pytest-odoo
last release: May 20, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=8

py.test plugin to run Odoo tests

pytest-odoo-fixtures
last release: Jun 25, 2019, *status*: N/A, *requires*: N/A

Project description

pytest-oduit
last release: Feb 11, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8

py.test plugin to run Odoo tests

pytest-oerp
last release: Feb 28, 2012, *status*: 3 - Alpha, *requires*: N/A

pytest plugin to test OpenERP modules

pytest-offline
last release: Mar 09, 2023, *status*: 1 - Planning, *requires*: pytest (>=7.0.0,<8.0.0)

pytest-ogsm-plugin
last release: May 16, 2023, *status*: N/A, *requires*: N/A

针对特定项目定制化插件，优化了pytest报告展示方式,并添加了项目所需特定参数

pytest-ok
last release: Apr 01, 2019, *status*: 4 - Beta, *requires*: N/A

The ultimate pytest output plugin

pytest-once
last release: Oct 10, 2025, *status*: 3 - Alpha, *requires*: pytest>=8.4.0

xdist-safe 'run once' fixture decorator for pytest (setup/teardown across workers)

pytest-only
last release: May 27, 2024, *status*: 5 - Production/Stable, *requires*: pytest<9,>=3.6.0

Use @pytest.mark.only to run a single test

pytest-only-markers
last release: Mar 17, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.1

A pytest plugin that isolates test execution to only tests decorated with ONLY* markers, stripping all other markers from matching items.

pytest-oof
last release: Dec 11, 2023, *status*: 4 - Beta, *requires*: N/A

A Pytest plugin providing structured, programmatic access to a test run's results

pytest-oot
last release: Sep 18, 2016, *status*: 4 - Beta, *requires*: N/A

Run object-oriented tests in a simple format

pytest-openapi
last release: Apr 12, 2026, *status*: N/A, *requires*: pytest>=7.0.0

`pytest –openapi` - an opinionated, lightweight black-box contract tester against a live API using its OpenAPI specification as the source of truth

pytest-openfiles
last release: Jun 05, 2024, *status*: 3 - Alpha, *requires*: pytest>=4.6

Pytest plugin for detecting inadvertent open file handles

pytest-open-html
last release: Mar 31, 2025, *status*: N/A, *requires*: pytest>=6.0

Auto-open HTML reports after pytest runs

pytest-opentelemetry
last release: Apr 25, 2025, *status*: N/A, *requires*: pytest

A pytest plugin for instrumenting test runs via OpenTelemetry

pytest-opentmi
last release: Feb 09, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=5.0

pytest plugin for publish results to opentmi

pytest-operator
last release: Sep 28, 2022, *status*: N/A, *requires*: pytest

Fixtures for Charmed Operators

pytest-optional
last release: Oct 07, 2015, *status*: N/A, *requires*: N/A

include/exclude values of fixtures in pytest

pytest-optional-tests
last release: Jul 21, 2025, *status*: 4 - Beta, *requires*: pytest; extra == "dev"

Easy declaration of optional tests (i.e., that are not run by default)

pytest-orchestration
last release: Jul 18, 2019, *status*: N/A, *requires*: N/A

A pytest plugin for orchestrating tests

pytest-order
last release: Aug 22, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=5.0; python_version < "3.10"

pytest plugin to run your tests in a specific order

pytest-ordered
last release: Nov 09, 2025, *status*: N/A, *requires*: pytest>=6.2.0

Declare the order in which tests should run in your pytest.ini

pytest-ordering
last release: Nov 14, 2018, *status*: 4 - Beta, *requires*: pytest

pytest plugin to run your tests in a specific order

pytest-order-modify
last release: Nov 04, 2022, *status*: N/A, *requires*: N/A

新增run_marker 来自定义用例的执行顺序

pytest-osxnotify
last release: May 15, 2015, *status*: N/A, *requires*: N/A

OS X notifications for py.test results.

pytest-ot
last release: Mar 21, 2024, *status*: N/A, *requires*: pytest; extra == "dev"

A pytest plugin for instrumenting test runs via OpenTelemetry

pytest-otel
last release: Apr 14, 2026, *status*: N/A, *requires*: pytest==9.0.3

OpenTelemetry plugin for Pytest

pytest-otelmark
last release: Sep 14, 2025, *status*: 3 - Alpha, *requires*: pytest>=8.3.5

Pytest plugin for otelmark.

pytest-override-env-var
last release: Feb 25, 2023, *status*: N/A, *requires*: N/A

Pytest mark to override a value of an environment variable.

pytest-owner
last release: Aug 19, 2024, *status*: N/A, *requires*: pytest

Add owner mark for tests

pytest-pact
last release: Jan 07, 2019, *status*: 4 - Beta, *requires*: N/A

A simple plugin to use with pytest

pytest-pagerduty
last release: Mar 22, 2025, *status*: N/A, *requires*: pytest<9.0.0,>=7.4.0

Pytest plugin for PagerDuty integration via automation testing.

pytest-pahrametahrize
last release: Nov 24, 2021, *status*: 4 - Beta, *requires*: pytest (>=6.0,<7.0)

Parametrize your tests with a Boston accent.

pytest-paia-blockly
last release: Mar 19, 2026, *status*: N/A, *requires*: pytest>=8.0

pytest plugin for PAIA Blockly: verify get_solution() against test cases

pytest-paraflow
last release: Feb 26, 2026, *status*: 3 - Alpha, *requires*: pytest>=9.0.0

Deterministic pytest test sharding across CI machines

pytest-parallel
last release: Oct 10, 2021, *status*: 3 - Alpha, *requires*: pytest (>=3.0.0)

a pytest plugin for parallel and concurrent testing

pytest-parallel-39
last release: Jul 12, 2021, *status*: 3 - Alpha, *requires*: pytest (>=3.0.0)

a pytest plugin for parallel and concurrent testing

pytest-parallelize-tests
last release: Jan 27, 2023, *status*: 4 - Beta, *requires*: N/A

pytest plugin that parallelizes test execution across multiple hosts

pytest-param
last release: Sep 11, 2016, *status*: 4 - Beta, *requires*: pytest (>=2.6.0)

pytest plugin to test all, first, last or random params

pytest-parametrization
last release: May 22, 2022, *status*: 5 - Production/Stable, *requires*: N/A

Simpler PyTest parametrization

pytest-parametrization-annotation
last release: Dec 10, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7

A pytest library for parametrizing tests using type hints.

pytest-parametrize
last release: Apr 03, 2026, *status*: 5 - Production/Stable, *requires*: pytest<10.0,>=8.3

pytest decorator for parametrizing test cases in a dict-way

pytest-parametrize-cases
last release: Mar 13, 2022, *status*: N/A, *requires*: pytest (>=6.1.2)

A more user-friendly way to write parametrized tests.

pytest-parametrized
last release: Dec 21, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Pytest decorator for parametrizing tests with default iterables.

pytest-parametrize-suite
last release: Jan 19, 2023, *status*: 5 - Production/Stable, *requires*: pytest

A simple pytest extension for creating a named test suite.

pytest_param_files
last release: Jul 29, 2023, *status*: N/A, *requires*: pytest

Create pytest parametrize decorators from external files.

pytest-params
last release: Mar 14, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

Simplified pytest test case parameters.

pytest-param-scope
last release: Oct 18, 2023, *status*: N/A, *requires*: pytest

pytest parametrize scope fixture workaround

pytest-parawtf
last release: Dec 03, 2018, *status*: 4 - Beta, *requires*: pytest (>=3.6.0)

Finally spell paramete?ri[sz]e correctly

pytest-park
last release: Mar 20, 2026, *status*: N/A, *requires*: N/A

Organise and analyse your pytest benchmarks

pytest-pass
last release: Dec 04, 2019, *status*: N/A, *requires*: N/A

Check out https://github.com/elilutsky/pytest-pass

pytest-passrunner
last release: Feb 10, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=4.6.0)

Pytest plugin providing the 'run_on_pass' marker

pytest-paste-config
last release: Sep 18, 2013, *status*: 3 - Alpha, *requires*: N/A

Allow setting the path to a paste config file

pytest-patch
last release: Apr 29, 2023, *status*: 3 - Alpha, *requires*: pytest (>=7.0.0)

An automagic `patch` fixture that can patch objects directly or by name.

pytest-patches
last release: Aug 30, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A contextmanager pytest fixture for handling multiple mock patches

pytest-patterns
last release: Oct 22, 2024, *status*: 4 - Beta, *requires*: pytest>=6

pytest plugin to make testing complicated long string output easy to write and easy to debug

pytest-pdb
last release: Jul 31, 2018, *status*: N/A, *requires*: N/A

pytest plugin which adds pdb helper commands related to pytest.

pytest-peach
last release: Apr 12, 2019, *status*: 4 - Beta, *requires*: pytest (>=2.8.7)

pytest plugin for fuzzing with Peach API Security

pytest-pep257
last release: Jul 09, 2016, *status*: N/A, *requires*: N/A

py.test plugin for pep257

pytest-pep723
last release: Apr 06, 2026, *status*: 4 - Beta, *requires*: pytest>=7

Pytest plugin to verify PEP 723 inline script metadata covers all imports.

pytest-pep8
last release: Apr 27, 2014, *status*: N/A, *requires*: N/A

pytest plugin to check PEP8 requirements

pytest-percent
last release: May 21, 2020, *status*: N/A, *requires*: pytest (>=5.2.0)

Change the exit code of pytest test sessions when a required percent of tests pass.

pytest-percents
last release: Mar 16, 2024, *status*: N/A, *requires*: N/A

pytest-perf
last release: May 20, 2024, *status*: 5 - Production/Stable, *requires*: pytest!=8.1.*,>=6; extra == "testing"

Run performance tests against the mainline code.

pytest-performance
last release: Sep 11, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=3.7.0)

A simple plugin to ensure the execution of critical sections of code has not been impacted

pytest-performancetotal
last release: Mar 24, 2026, *status*: 5 - Production/Stable, *requires*: N/A

A performance plugin for pytest

pytest-persistence
last release: Aug 21, 2024, *status*: N/A, *requires*: N/A

Pytest tool for persistent objects

pytest-pexpect
last release: Sep 10, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Pytest pexpect plugin.

pytest-pg
last release: May 18, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.4

A tiny plugin for pytest which runs PostgreSQL in Docker

pytest-pgsql
last release: May 13, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=3.0.0)

Pytest plugins and helpers for tests using a Postgres database.

pytest-phmdoctest
last release: Apr 15, 2022, *status*: 4 - Beta, *requires*: pytest (>=5.4.3)

pytest plugin to test Python examples in Markdown using phmdoctest.

pytest-phoenix-interface
last release: Mar 19, 2025, *status*: N/A, *requires*: N/A

Pytest extension tool for phoenix projects.

pytest-picked
last release: Nov 06, 2024, *status*: N/A, *requires*: pytest>=3.7.0

Run the tests related to the changed files

pytest-pickle-cache
last release: Feb 06, 2026, *status*: 4 - Beta, *requires*: pytest>=9

A pytest plugin for caching test results using pickle.

pytest-pigeonhole
last release: Jun 25, 2018, *status*: 5 - Production/Stable, *requires*: pytest (>=3.4)

pytest-pikachu
last release: Aug 05, 2021, *status*: 5 - Production/Stable, *requires*: pytest

Show surprise when tests are passing

pytest-pilot
last release: Dec 17, 2025, *status*: 5 - Production/Stable, *requires*: N/A

Slice in your test base thanks to powerful markers.

pytest-pingguo-pytest-plugin
last release: Oct 26, 2022, *status*: 4 - Beta, *requires*: N/A

pingguo test

pytest-pings
last release: Jun 29, 2019, *status*: 3 - Alpha, *requires*: pytest (>=5.0.0)

🦊 The pytest plugin for Firefox Telemetry 📊

pytest-pinned
last release: Sep 17, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A simple pytest plugin for pinning tests

pytest-pinpoint
last release: Sep 25, 2020, *status*: N/A, *requires*: pytest (>=4.4.0)

A pytest plugin which runs SBFL algorithms to detect faults.

pytest-pipeline
last release: Jan 24, 2017, *status*: 3 - Alpha, *requires*: N/A

Pytest plugin for functional testing of data analysispipelines

pytest-pitch
last release: Nov 02, 2023, *status*: 4 - Beta, *requires*: pytest >=7.3.1

runs tests in an order such that coverage increases as fast as possible

pytest-platform-adapter
last release: Mar 23, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6.2.5

Pytest集成自动化平台插件

pytest-platform-markers
last release: Sep 09, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.6.0)

Markers for pytest to skip tests on specific platforms

pytest-play
last release: Jun 12, 2019, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin that let you automate actions and assertions with test metrics reporting executing plain YAML files

pytest-playbook
last release: Jan 21, 2021, *status*: 3 - Alpha, *requires*: pytest (>=6.1.2,<7.0.0)

Pytest plugin for reading playbooks.

pytest-playwright
last release: Nov 24, 2025, *status*: N/A, *requires*: pytest<10.0.0,>=6.2.4

A pytest wrapper with fixtures for Playwright to automate web browsers

pytest-playwright-artifacts
last release: Mar 23, 2026, *status*: N/A, *requires*: N/A

Capture screenshots, HTML, and console logs on Playwright test failures

pytest_playwright_async
last release: Sep 28, 2024, *status*: N/A, *requires*: N/A

ASYNC Pytest plugin for Playwright

pytest-playwright-asyncio
last release: Nov 24, 2025, *status*: N/A, *requires*: pytest<10.0.0,>=6.2.4

A pytest wrapper with async fixtures for Playwright to automate web browsers

pytest-playwright-axe
last release: Apr 17, 2026, *status*: 5 - Production/Stable, *requires*: N/A

An axe-core integration for accessibility testing using Playwright Python.

pytest-playwright-enhanced
last release: Mar 24, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.0.0

A pytest plugin for playwright python

pytest-playwright-json
last release: Jan 06, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Generate Playwright-compatible JSON reports from pytest-playwright test runs

pytest-playwrights
last release: Dec 02, 2021, *status*: N/A, *requires*: N/A

A pytest wrapper with fixtures for Playwright to automate web browsers

pytest-playwright-snapshot
last release: Aug 19, 2021, *status*: N/A, *requires*: N/A

A pytest wrapper for snapshot testing with playwright

pytest-playwright-visual
last release: Apr 28, 2022, *status*: N/A, *requires*: N/A

A pytest fixture for visual testing with Playwright

pytest-playwright-visual-snapshot
last release: Feb 05, 2026, *status*: N/A, *requires*: N/A

Easy pytest visual regression testing using playwright

pytest-pl-grader
last release: Nov 12, 2025, *status*: 3 - Alpha, *requires*: pytest

A pytest plugin for autograding Python code. Designed for use with the PrairieLearn platform.

pytest-plone
last release: Apr 18, 2026, *status*: 3 - Alpha, *requires*: pytest<8.0.0

Pytest plugin to test Plone addons

pytest-plt
last release: Jan 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Fixtures for quickly making Matplotlib plots in tests

pytest-plugin-helpers
last release: Nov 23, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A plugin to help developing and testing other plugins

pytest-plugins
last release: Apr 05, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.1

A Python package for managing pytest plugins.

pytest-plugin-utils
last release: Mar 26, 2026, *status*: N/A, *requires*: N/A

Reusable configuration and artifact utilities for building pytest plugins

pytest-plus
last release: Feb 02, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.4.2

PyTest Plus Plugin :: extends pytest functionality

pytest-pmisc
last release: Mar 21, 2019, *status*: 5 - Production/Stable, *requires*: N/A

pytest-podman
last release: Feb 03, 2026, *status*: N/A, *requires*: N/A

Pytest plugin for Podman integration

pytest-pogo
last release: Jan 20, 2026, *status*: 4 - Beta, *requires*: pytest>=7

Pytest plugin for pogo-migrate

pytest-pointers
last release: Dec 26, 2022, *status*: N/A, *requires*: N/A

Pytest plugin to define functions you test with special marks for better navigation and reports

pytest-pokie
last release: Oct 19, 2023, *status*: 5 - Production/Stable, *requires*: N/A

Pokie plugin for pytest

pytest-polarion-cfme
last release: Nov 13, 2017, *status*: 3 - Alpha, *requires*: N/A

pytest plugin for collecting test cases and recording test results

pytest-polarion-collect
last release: Jun 18, 2020, *status*: 3 - Alpha, *requires*: pytest

pytest plugin for collecting polarion test cases data

pytest-polecat
last release: Aug 12, 2019, *status*: 4 - Beta, *requires*: N/A

Provides Polecat pytest fixtures

pytest-polymeric-report
last release: Feb 24, 2026, *status*: N/A, *requires*: N/A

A polymeric test report plugin for pytest

pytest-ponyorm
last release: Oct 31, 2018, *status*: N/A, *requires*: pytest (>=3.1.1)

PonyORM in Pytest

pytest-poo
last release: Mar 25, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=2.3.4)

Visualize your crappy tests

pytest-poo-fail
last release: Feb 12, 2015, *status*: 5 - Production/Stable, *requires*: N/A

Visualize your failed tests with poo

pytest-pook
last release: Feb 15, 2024, *status*: 4 - Beta, *requires*: pytest

Pytest plugin for pook

pytest-pop
last release: May 09, 2023, *status*: 5 - Production/Stable, *requires*: pytest

A pytest plugin to help with testing pop projects

pytest-porcochu
last release: Nov 28, 2024, *status*: 5 - Production/Stable, *requires*: N/A

Show surprise when tests are passing

pytest-portion
last release: Mar 04, 2026, *status*: 4 - Beta, *requires*: pytest>=3.5.0

Select a portion of the collected tests

pytest-postgres
last release: Mar 22, 2020, *status*: N/A, *requires*: pytest

Run PostgreSQL in Docker container in Pytest.

pytest-postgresql
last release: Jan 23, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.2

Postgresql fixtures and fixture factories for Pytest.

pytest-power
last release: Dec 31, 2020, *status*: N/A, *requires*: pytest (>=5.4)

pytest plugin with powerful fixtures

pytest-powerpack
last release: Jan 04, 2025, *status*: N/A, *requires*: pytest<9.0.0,>=8.1.1

A plugin containing extra batteries for pytest

pytest-prairielearn-grader
last release: Mar 26, 2026, *status*: 3 - Alpha, *requires*: pytest

A pytest plugin for autograding Python code. Designed for use with the PrairieLearn platform.

pytest-prefer-nested-dup-tests
last release: Apr 27, 2022, *status*: 4 - Beta, *requires*: pytest (>=7.1.1,<8.0.0)

A Pytest plugin to drop duplicated tests during collection, but will prefer keeping nested packages.

pytest-pretty
last release: Jun 04, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7

pytest plugin for printing summary data as I want it

pytest-pretty-terminal
last release: Jan 31, 2022, *status*: N/A, *requires*: pytest (>=3.4.1)

pytest plugin for generating prettier terminal output

pytest-pride
last release: Apr 02, 2016, *status*: 3 - Alpha, *requires*: N/A

Minitest-style test colors

pytest-print
last release: Feb 11, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.2

pytest-print adds the printer fixture you can use to print messages to the user (directly to the pytest runner, not stdout)

pytest-priority
last release: Aug 19, 2024, *status*: N/A, *requires*: pytest

pytest plugin for add priority for tests

pytest-proceed
last release: Oct 01, 2024, *status*: N/A, *requires*: pytest

pytest-profiles
last release: Dec 09, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.7.0)

pytest plugin for configuration profiles

pytest-profiling
last release: Nov 29, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Profiling plugin for py.test

pytest-progress
last release: Nov 11, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=6.0

pytest plugin for instant test progress status

pytest-prometheus
last release: Oct 03, 2017, *status*: N/A, *requires*: N/A

Report test pass / failures to a Prometheus PushGateway

pytest-prometheus-pushgateway
last release: Sep 27, 2022, *status*: 5 - Production/Stable, *requires*: pytest

Pytest report plugin for Zulip

pytest-prometheus-pushgw
last release: May 19, 2025, *status*: N/A, *requires*: pytest>=6.0.0

Pytest plugin to export test metrics to Prometheus Pushgateway

pytest-proofy
last release: Nov 13, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Pytest plugin for Proofy test reporting

pytest-prosper
last release: Sep 24, 2018, *status*: N/A, *requires*: N/A

Test helpers for Prosper projects

pytest-prysk
last release: Dec 10, 2024, *status*: 4 - Beta, *requires*: pytest>=7.3.2

Pytest plugin for prysk

pytest-pspec
last release: Jun 02, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.0.0)

A rspec format reporter for Python ptest

pytest-psqlgraph
last release: Oct 19, 2021, *status*: 4 - Beta, *requires*: pytest (>=6.0)

pytest plugin for testing applications that use psqlgraph

pytest-pt
last release: Nov 21, 2025, *status*: 5 - Production/Stable, *requires*: pytest

pytest plugin to use *.pt files as tests

pytest-ptera
last release: Mar 01, 2022, *status*: N/A, *requires*: pytest (>=6.2.4,<7.0.0)

Use ptera probes in tests

pytest-publish
last release: Jun 04, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.0.0

pytest-pudb
last release: Oct 25, 2018, *status*: 3 - Alpha, *requires*: pytest (>=2.0)

Pytest PuDB debugger integration

pytest-pudb-resurrected
last release: Mar 12, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Pytest PuDB debugger integration

pytest-pumpkin-spice
last release: Sep 18, 2022, *status*: 4 - Beta, *requires*: N/A

A pytest plugin that makes your test reporting pumpkin-spiced

pytest-purkinje
last release: Oct 28, 2017, *status*: 2 - Pre-Alpha, *requires*: N/A

py.test plugin for purkinje test runner

pytest-pusher
last release: Jan 06, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=3.6)

pytest plugin for push report to minio

pytest-pvcr
last release: Feb 25, 2026, *status*: 3 - Alpha, *requires*: pytest>=3.5.0

PyTest Process VCR

pytest-pve-cloud
last release: Feb 27, 2026, *status*: N/A, *requires*: pytest==8.4.2

pytest-pw-config-gen
last release: Mar 14, 2026, *status*: N/A, *requires*: pytest>=7.4; extra == "dev"

Generate pytest-playwright configuration files (pytest.ini, pyproject.toml, conftest.py) via CLI

pytest-py125
last release: Dec 03, 2022, *status*: N/A, *requires*: N/A

pytest-pycharm
last release: Aug 13, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=2.3)

Plugin for py.test to enter PyCharm debugger on uncaught exceptions

pytest-pycodestyle
last release: Jul 20, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0

pytest plugin to run pycodestyle

pytest-pydantic-schema-sync
last release: Aug 29, 2024, *status*: N/A, *requires*: pytest>=6

Pytest plugin to synchronise Pydantic model schemas with JSONSchema files

pytest-pydev
last release: Nov 15, 2017, *status*: 3 - Alpha, *requires*: N/A

py.test plugin to connect to a remote debug server with PyDev or PyCharm.

pytest-pydocstyle
last release: Oct 09, 2024, *status*: 3 - Alpha, *requires*: pytest>=7.0

pytest plugin to run pydocstyle

pytest-pyeval
last release: Apr 13, 2026, *status*: N/A, *requires*: pytest>=8.0

pytest plugin integrating pydantic-evals

pytest-pylembic
last release: Jul 22, 2025, *status*: 3 - Alpha, *requires*: N/A

This package provides pytest plugin for validating Alembic migrations using the pylembic package.

pytest-pylint
last release: Oct 06, 2023, *status*: 5 - Production/Stable, *requires*: pytest >=7.0

pytest plugin to check source code with pylint

pytest-pylyzer
last release: Feb 15, 2025, *status*: 4 - Beta, *requires*: N/A

A pytest plugin for pylyzer

pytest-pymysql-autorecord
last release: Sep 02, 2022, *status*: N/A, *requires*: N/A

Record PyMySQL queries and mock with the stored data.

pytest-pyodide
last release: Apr 17, 2026, *status*: N/A, *requires*: pytest

Pytest plugin for testing applications that use Pyodide

pytest-pypi
last release: Mar 04, 2018, *status*: 3 - Alpha, *requires*: N/A

Easily test your HTTP library against a local copy of pypi

pytest-pypom-navigation
last release: Feb 18, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.0.7)

Core engine for cookiecutter-qa and pytest-play packages

pytest-pyppeteer
last release: Apr 28, 2022, *status*: N/A, *requires*: pytest (>=6.2.5,<7.0.0)

A plugin to run pyppeteer in pytest

pytest-pyq
last release: Mar 10, 2020, *status*: 5 - Production/Stable, *requires*: N/A

Pytest fixture “q” for pyq

pytest-pyramid
last release: Sep 30, 2025, *status*: 5 - Production/Stable, *requires*: pytest

pytest_pyramid - provides fixtures for testing pyramid applications with pytest test suite

pytest-pyramid-server
last release: Oct 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Pyramid server fixture for py.test

pytest-pyreport
last release: May 05, 2024, *status*: N/A, *requires*: pytest

PyReport is a lightweight reporting plugin for Pytest that provides concise HTML report

pytest-pyright
last release: Jan 26, 2024, *status*: 4 - Beta, *requires*: pytest >=7.0.0

Pytest plugin for type checking code with Pyright

pytest-pyspark-plugin
last release: Nov 23, 2025, *status*: 4 - Beta, *requires*: pytest>=8.0.0

Pytest pyspark plugin (p3)

pytest-pyspec
last release: Nov 18, 2025, *status*: 5 - Production/Stable, *requires*: pytest<10,>=9

The pytest-pyspec plugin transforms pytest output into a beautiful, readable format similar to RSpec. It provides semantic meaning to your tests by organizing them into descriptive hierarchies, using the prefixes `Describe`/`Test`, `With`/`Without`/`When`, and `test`/`it`, while allowing docstrings and decorators to override the descriptions.

pytest-pystack
last release: Nov 16, 2024, *status*: N/A, *requires*: pytest>=3.5.0

Plugin to run pystack after a timeout for a test suite.

pytest-pytestdb
last release: Sep 14, 2025, *status*: N/A, *requires*: N/A

Add your description here

pytest-pytestrail
last release: Aug 27, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.8.0)

Pytest plugin for interaction with TestRail

pytest-pytestrail-internal
last release: Jun 12, 2025, *status*: 4 - Beta, *requires*: pytest>=3.8.0

Pytest plugin for interaction with TestRail, Pytest plugin for TestRail (internal fork from: https://github.com/tolstislon/pytest-pytestrail with PR #25 fix)

pytest-pythonhashseed
last release: Nov 16, 2025, *status*: 4 - Beta, *requires*: pytest>=3.0.0

Pytest plugin to set PYTHONHASHSEED env var.

pytest-pythonpath
last release: Feb 10, 2022, *status*: 5 - Production/Stable, *requires*: pytest (<7,>=2.5.2)

pytest plugin for adding to the PYTHONPATH from command line or configs.

pytest-python-test-engineer-sort
last release: May 13, 2024, *status*: N/A, *requires*: pytest>=6.2.0

Sort plugin for Pytest

pytest-pytorch
last release: May 25, 2021, *status*: 4 - Beta, *requires*: pytest

pytest plugin for a better developer experience when working with the PyTorch test suite

pytest-pyvenv
last release: Feb 27, 2024, *status*: N/A, *requires*: pytest ; extra == 'test'

A package for create venv in tests

pytest-pyvista
last release: Apr 13, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Pytest-pyvista package.

pytest-qanova
last release: Sep 05, 2024, *status*: 3 - Alpha, *requires*: pytest

A pytest plugin to collect test information

pytest-qaseio
last release: Dec 10, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.2.2

Pytest plugin for Qase.io integration

pytest-qasync
last release: Jul 12, 2021, *status*: 4 - Beta, *requires*: pytest (>=5.4.0)

Pytest support for qasync.

pytest-qatouch
last release: Feb 14, 2023, *status*: 4 - Beta, *requires*: pytest (>=6.2.0)

Pytest plugin for uploading test results to your QA Touch Testrun.

pytest-qfield
last release: Apr 06, 2026, *status*: N/A, *requires*: N/A

A pytest plugin for testing QField qml plugins

pytest-qgis
last release: Apr 01, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6.0

A pytest plugin for testing QGIS python plugins

pytest-qml
last release: Dec 02, 2020, *status*: 4 - Beta, *requires*: pytest (>=6.0.0)

Run QML Tests with pytest

pytest-qr
last release: Nov 25, 2021, *status*: 4 - Beta, *requires*: N/A

pytest plugin to generate test result QR codes

pytest-qt
last release: Jul 01, 2025, *status*: 5 - Production/Stable, *requires*: pytest

pytest support for PyQt and PySide applications

pytest-qt-app
last release: Oct 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

QT app fixture for py.test

pytest-quantum
last release: Mar 22, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.0

A cross-framework pytest plugin for quantum program testing

pytest-quarantine
last release: Nov 24, 2019, *status*: 5 - Production/Stable, *requires*: pytest (>=4.6)

A plugin for pytest to manage expected test failures

pytest-quickcheck
last release: Nov 05, 2022, *status*: 4 - Beta, *requires*: pytest (>=4.0)

pytest plugin to generate random data inspired by QuickCheck

pytest_quickify
last release: Jun 14, 2019, *status*: N/A, *requires*: pytest

Run test suites with pytest-quickify.

pytest-rabbitmq
last release: Mar 22, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6.2

RabbitMQ process and client fixtures for pytest

pytest-race
last release: Jun 07, 2022, *status*: 4 - Beta, *requires*: N/A

Race conditions tester for pytest

pytest-rage
last release: Oct 21, 2011, *status*: 3 - Alpha, *requires*: N/A

pytest plugin to implement PEP712

pytest-rail
last release: May 02, 2022, *status*: N/A, *requires*: pytest (>=3.6)

pytest plugin for creating TestRail runs and adding results

pytest-railflow-testrail-reporter
last release: Jun 29, 2022, *status*: 5 - Production/Stable, *requires*: pytest

Generate json reports along with specified metadata defined in test markers.

pytest-raises
last release: Apr 23, 2020, *status*: N/A, *requires*: pytest (>=3.2.2)

An implementation of pytest.raises as a pytest.mark fixture

pytest-raisesregexp
last release: Dec 18, 2015, *status*: N/A, *requires*: N/A

Simple pytest plugin to look for regex in Exceptions

pytest-raisin
last release: Feb 06, 2022, *status*: N/A, *requires*: pytest

Plugin enabling the use of exception instances with pytest.raises

pytest-random
last release: Apr 28, 2013, *status*: 3 - Alpha, *requires*: N/A

py.test plugin to randomize tests

pytest-randomly
last release: Sep 12, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin to randomly order tests and control random.seed.

pytest-randomness
last release: May 30, 2019, *status*: 3 - Alpha, *requires*: N/A

Pytest plugin about random seed management

pytest-random-num
last release: Oct 19, 2020, *status*: 5 - Production/Stable, *requires*: N/A

Randomise the order in which pytest tests are run with some control over the randomness

pytest-random-order
last release: Jun 22, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Randomise the order in which pytest tests are run with some control over the randomness

pytest-ranking
last release: Apr 08, 2025, *status*: 4 - Beta, *requires*: pytest>=7.4.3

A Pytest plugin for faster fault detection via regression test prioritization

pytest-rca-report
last release: Aug 04, 2025, *status*: N/A, *requires*: N/A

Interactive RCA report generator for pytest runs, with AI-based analysis and visual dashboard

pytest-readable
last release: Mar 23, 2026, *status*: 3 - Alpha, *requires*: pytest<10.0,>=9.0

Pytest plugin that renders readable test specifications and exports documentation

pytest-readme
last release: Aug 01, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Test your README.md file

pytest-reana
last release: Mar 03, 2026, *status*: 3 - Alpha, *requires*: N/A

Pytest fixtures for REANA.

pytest-recap
last release: Jun 16, 2025, *status*: N/A, *requires*: pytest>=6.2.0

Capture your test sessions. Recap the results.

pytest-recorder
last release: Apr 13, 2026, *status*: N/A, *requires*: pytest>=8.4.1

Pytest plugin, meant to facilitate unit tests writing for tools consumming Web APIs.

pytest-recording
last release: May 08, 2025, *status*: 4 - Beta, *requires*: pytest>=3.5.0

A pytest plugin powered by VCR.py to record and replay HTTP traffic

pytest-recordings
last release: Aug 13, 2020, *status*: N/A, *requires*: N/A

Provides pytest plugins for reporting request/response traffic, screenshots, and more to ReportPortal

pytest-record-video
last release: Oct 31, 2024, *status*: N/A, *requires*: N/A

用例执行过程中录制视频

pytest-redis
last release: Feb 28, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.4.0

Redis fixtures and fixture factories for Pytest.

pytest-redislite
last release: Apr 05, 2022, *status*: 4 - Beta, *requires*: pytest

Pytest plugin for testing code using Redis

pytest-redmine
last release: Mar 19, 2018, *status*: 1 - Planning, *requires*: N/A

Pytest plugin for redmine

pytest-ref
last release: Nov 23, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A plugin to store reference files to ease regression testing

pytest-reference-formatter
last release: Oct 01, 2019, *status*: 4 - Beta, *requires*: N/A

Conveniently run pytest with a dot-formatted test reference.

pytest-regex
last release: May 29, 2023, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Select pytest tests with regular expressions

pytest-regex-dependency
last release: Jun 12, 2022, *status*: N/A, *requires*: pytest

Management of Pytest dependencies via regex patterns

pytest-regressions
last release: Feb 10, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=6.2.0

Easy to use fixtures to write regression tests.

pytest-regtest
last release: Apr 18, 2026, *status*: N/A, *requires*: pytest>7.2

pytest plugin for snapshot regression testing

pytest-relative-order
last release: May 17, 2021, *status*: 4 - Beta, *requires*: N/A

a pytest plugin that sorts tests using “before” and “after” markers

pytest-relative-path
last release: Nov 13, 2025, *status*: N/A, *requires*: pytest

Handle relative path in pytest options or ini configs

pytest-relaxed
last release: Mar 29, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7

Relaxed test discovery/organization for pytest

pytest_relay
last release: Jan 31, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin to relay test information and control from and to pytest

pytest-relay-run
last release: Jan 31, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A pytest wrapper using for pytest-relay with pytest-relay-ws to control pytest executions.

pytest_relay_ws
last release: Jan 31, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

An extension plugin to pytest-relay to relay pytest information via websockets

pytest-remaster
last release: Apr 09, 2026, *status*: 3 - Alpha, *requires*: pytest>=7

Pytest plugin for golden master (characterisation) testing with automatic expected file regeneration.

pytest-remfiles
last release: Jul 01, 2019, *status*: 5 - Production/Stable, *requires*: N/A

Pytest plugin to create a temporary directory with remote files

pytest-remotedata
last release: Sep 26, 2023, *status*: 5 - Production/Stable, *requires*: pytest >=4.6

Pytest plugin for controlling remote data access.

pytest-remote-response
last release: Apr 26, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=4.6)

Pytest plugin for capturing and mocking connection requests.

pytest-remove-stale-bytecode
last release: Nov 19, 2025, *status*: 4 - Beta, *requires*: pytest

py.test plugin to remove stale byte code files.

pytest-reorder
last release: May 31, 2018, *status*: 4 - Beta, *requires*: pytest

Reorder tests depending on their paths and names.

pytest-repeat
last release: Apr 07, 2025, *status*: 5 - Production/Stable, *requires*: pytest

pytest plugin for repeating tests

pytest-repeated
last release: Feb 24, 2026, *status*: N/A, *requires*: pytest>=7.0.0

A pytest module for very basic statistical tests. Repeat test multiple times and pass if the underlying test passes a threshold.

pytest_repeater
last release: Feb 09, 2018, *status*: 1 - Planning, *requires*: N/A

py.test plugin for repeating single test multiple times.

pytest-replay
last release: Dec 23, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Saves previous test runs and allow re-execute previous pytest runs to reproduce crashes or flaky tests

pytest-repo-health
last release: Dec 09, 2025, *status*: 3 - Alpha, *requires*: pytest

A pytest plugin to report on repository standards conformance

pytest-report
last release: May 11, 2016, *status*: 4 - Beta, *requires*: N/A

Creates json report that is compatible with atom.io's linter message format

pytest-reporter
last release: Feb 28, 2024, *status*: 4 - Beta, *requires*: pytest

Generate Pytest reports with templates

pytest-reporter-html
last release: Mar 15, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0

Pytest plugin that generates rich HTML test reports with step tracking, log capture, and interactive filtering

pytest-reporter-html1
last release: Oct 10, 2025, *status*: 4 - Beta, *requires*: N/A

A basic HTML report template for Pytest

pytest-reporter-html-dots
last release: Apr 26, 2025, *status*: N/A, *requires*: N/A

A basic HTML report for pytest using Jinja2 template engine.

pytest-reporter-plus
last release: Jul 16, 2025, *status*: N/A, *requires*: N/A

Lightweight enhanced HTML reporter for Pytest

pytest-report-extras
last release: Dec 24, 2025, *status*: N/A, *requires*: pytest>=8.4.0

Pytest plugin to enhance pytest-html and allure reports by adding comments, screenshots, webpage sources and attachments.

pytest-reportinfra
last release: Aug 11, 2019, *status*: 3 - Alpha, *requires*: N/A

Pytest plugin for reportinfra

pytest-reporting
last release: Oct 25, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A plugin to report summarized results in a table format

pytest-reportlog
last release: Nov 11, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Replacement for the –resultlog option, focused in simplicity and extensibility

pytest-report-me
last release: Dec 31, 2020, *status*: N/A, *requires*: pytest

A pytest plugin to generate report.

pytest-report-parameters
last release: Jun 18, 2020, *status*: 3 - Alpha, *requires*: pytest (>=2.4.2)

pytest plugin for adding tests' parameters to junit report

pytest-reportportal
last release: Apr 15, 2026, *status*: N/A, *requires*: N/A

Agent for Reporting results of tests to the Report Portal

pytest-report-stream
last release: Oct 22, 2023, *status*: 4 - Beta, *requires*: N/A

A pytest plugin which allows to stream test reports at runtime

pytest-repo-structure
last release: Mar 18, 2024, *status*: 1 - Planning, *requires*: N/A

Pytest Repo Structure

pytest-req
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.4.2

pytest requests plugin

pytest-reqcov
last release: Jul 04, 2025, *status*: 3 - Alpha, *requires*: pytest>=6.0

A pytest plugin for requirement coverage tracking

pytest-reqs
last release: May 12, 2019, *status*: N/A, *requires*: pytest (>=2.4.2)

pytest plugin to check pinned requirements

pytest-requests
last release: Jun 24, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A simple plugin to use with pytest

pytest-requestselapsed
last release: Aug 14, 2022, *status*: N/A, *requires*: N/A

collect and show http requests elapsed time

pytest-requests-futures
last release: Jul 06, 2022, *status*: 5 - Production/Stable, *requires*: pytest

Pytest Plugin to Mock Requests Futures

pytest-requirements
last release: Mar 10, 2026, *status*: N/A, *requires*: pytest

pytest plugin for using custom markers to relate tests to requirements and usecases

pytest-requires
last release: Dec 21, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A pytest plugin to elegantly skip tests with optional requirements

pytest-reqyaml
last release: Aug 16, 2025, *status*: N/A, *requires*: pytest>=8.4.1

This is a plugin where generate requests test cases from yaml.

pytest-reraise
last release: Sep 20, 2022, *status*: 5 - Production/Stable, *requires*: pytest (>=4.6)

Make multi-threaded pytest test cases fail when they should

pytest-rerun
last release: Jul 08, 2019, *status*: N/A, *requires*: pytest (>=3.6)

Re-run only changed files in specified branch

pytest-rerun-all
last release: Jul 30, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

Rerun testsuite for a certain time or iterations

pytest-rerunclassfailures
last release: Apr 24, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7.2

pytest rerun class failures plugin

pytest-rerunfailures
last release: Oct 10, 2025, *status*: 5 - Production/Stable, *requires*: pytest!=8.2.2,>=7.4

pytest plugin to re-run tests to eliminate flaky failures

pytest-rerunfailures-all-logs
last release: Mar 07, 2022, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin to re-run tests to eliminate flaky failures

pytest-reserial
last release: Dec 30, 2025, *status*: 4 - Beta, *requires*: pytest

Pytest fixture for recording and replaying serial port traffic.

pytest-resilient-circuits
last release: Nov 13, 2025, *status*: N/A, *requires*: pytest~=7.0

Resilient Circuits fixtures for PyTest

pytest-resource
last release: Nov 14, 2018, *status*: 4 - Beta, *requires*: N/A

Load resource fixture plugin to use with pytest

pytest-resource-path
last release: Sep 18, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=3.5.0

Provides path for uniform access to test resources in isolated directory

pytest-resource-usage
last release: Nov 06, 2022, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

Pytest plugin for reporting running time and peak memory usage

pytest-respect
last release: Apr 08, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.0.0

Pytest plugin to load resource files relative to test code and to expect values to match them.

pytest-responsemock
last release: Mar 10, 2022, *status*: 5 - Production/Stable, *requires*: N/A

Simplified requests calls mocking for pytest

pytest-responses
last release: Oct 11, 2022, *status*: N/A, *requires*: pytest (>=2.5)

py.test integration for responses

pytest-rest-api
last release: Aug 08, 2022, *status*: N/A, *requires*: pytest (>=7.1.2,<8.0.0)

pytest-restrict
last release: Feb 09, 2026, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin to restrict the test types allowed

pytest-resttest
last release: Jan 01, 2026, *status*: 5 - Production/Stable, *requires*: N/A

A REST API testing framework for Python, as plugin for pytest. Uses simple and readable YAML files for specifying test cases.

pytest-result-log
last release: Jan 10, 2024, *status*: N/A, *requires*: pytest>=7.2.0

A pytest plugin that records the start, end, and result information of each use case in a log file

pytest-result-notify
last release: Apr 27, 2025, *status*: N/A, *requires*: pytest>=8.3.5

Default template for PDM package

pytest-results
last release: Oct 08, 2025, *status*: 4 - Beta, *requires*: pytest

Easily spot regressions in your tests.

pytest-result-sender
last release: Apr 20, 2023, *status*: N/A, *requires*: pytest>=7.3.1

pytest-result-sender-fanrenbufan
last release: Mar 12, 2026, *status*: N/A, *requires*: pytest>=8.4.2

Default template for PDM package

pytest-result-sender-jms
last release: May 22, 2025, *status*: N/A, *requires*: pytest>=8.3.5

Default template for PDM package

pytest-result-sender-lj
last release: Dec 17, 2024, *status*: N/A, *requires*: pytest>=8.3.4

Default template for PDM package

pytest-result-sender-lyt
last release: Mar 14, 2025, *status*: N/A, *requires*: pytest>=8.3.5

Default template for PDM package

pytest-result-sender-misszhang
last release: Mar 21, 2025, *status*: N/A, *requires*: pytest>=8.3.5

Default template for PDM package

pytest-result-sender-r
last release: Dec 26, 2025, *status*: N/A, *requires*: pytest>=8.4.2

Default template for PDM package

pytest-resume
last release: Apr 22, 2023, *status*: 4 - Beta, *requires*: pytest (>=7.0)

A Pytest plugin to resuming from the last run test

pytest-rethinkdb
last release: Jul 24, 2016, *status*: 4 - Beta, *requires*: N/A

A RethinkDB plugin for pytest.

pytest-retry
last release: Jan 19, 2025, *status*: N/A, *requires*: pytest>=7.0.0

Adds the ability to retry flaky tests in CI environments

pytest-retry-class
last release: Nov 24, 2024, *status*: N/A, *requires*: pytest>=5.3

A pytest plugin to rerun entire class on failure

pytest-reusable-testcases
last release: Apr 28, 2023, *status*: N/A, *requires*: N/A

pytest-revealtype-injector
last release: Feb 03, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

Pytest plugin for replacing reveal_type() calls inside test functions with static and runtime type checking result comparison, for confirming type annotation validity.

pytest-reverse
last release: Sep 09, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Pytest plugin to reverse test order.

pytest-review
last release: Apr 12, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

A pytest plugin that reviews the quality of your tests

pytest-rich
last release: Dec 12, 2024, *status*: 4 - Beta, *requires*: pytest>=7.0

Leverage rich for richer test session output

pytest-richer
last release: Oct 27, 2023, *status*: 3 - Alpha, *requires*: pytest

Pytest plugin providing a Rich based reporter.

pytest-rich-reporter
last release: Feb 17, 2022, *status*: 1 - Planning, *requires*: pytest (>=5.0.0)

A pytest plugin using Rich for beautiful test result formatting.

pytest-richtrace
last release: Jun 20, 2023, *status*: N/A, *requires*: N/A

A pytest plugin that displays the names and information of the pytest hook functions as they are executed.

pytest-ringo
last release: Sep 27, 2017, *status*: 3 - Alpha, *requires*: N/A

pytest plugin to test webapplications using the Ringo webframework

pytest-rmsis
last release: Aug 10, 2022, *status*: N/A, *requires*: pytest (>=5.3.5)

Sycronise pytest results to Jira RMsis

pytest-rmysql
last release: Aug 17, 2025, *status*: N/A, *requires*: pytest>=8.4.1

This is a plugin which is able to connet MySQL easyly.

pytest-rng
last release: Aug 08, 2019, *status*: 5 - Production/Stable, *requires*: pytest

Fixtures for seeding tests and making randomness reproducible

pytest-roast
last release: Nov 09, 2022, *status*: 5 - Production/Stable, *requires*: pytest

pytest plugin for ROAST configuration override and fixtures

pytest_robotframework
last release: Dec 22, 2025, *status*: N/A, *requires*: pytest<10,>=7

a pytest plugin that can run both python and robotframework tests while generating robot reports for them

pytest-rocketchat
last release: Apr 18, 2021, *status*: 5 - Production/Stable, *requires*: N/A

Pytest to Rocket.Chat reporting plugin

pytest-rotest
last release: Sep 08, 2019, *status*: N/A, *requires*: pytest (>=3.5.0)

Pytest integration with rotest

pytest-route-coverage
last release: Apr 02, 2026, *status*: N/A, *requires*: pytest>=7.2.2

pytest plugin to generate reports on routes coverage for web applications.

pytest-routes
last release: Dec 01, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0

Property-based smoke testing for ASGI application routes

pytest-rpc
last release: Feb 22, 2019, *status*: 4 - Beta, *requires*: pytest (~=3.6)

Extend py.test for RPC OpenStack testing.

pytest-r-snapshot
last release: Jan 02, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

A pytest plugin for snapshot testing against R code outputs

pytest-rst
last release: Feb 22, 2026, *status*: N/A, *requires*: N/A

Test code from RST documents with pytest

pytest-rt
last release: May 05, 2022, *status*: N/A, *requires*: N/A

pytest data collector plugin for Testgr

pytest-rts
last release: May 17, 2021, *status*: N/A, *requires*: pytest

Coverage-based regression test selection (RTS) plugin for pytest

pytest-ruff
last release: Jun 19, 2025, *status*: 4 - Beta, *requires*: pytest>=5

pytest plugin to check ruff requirements.

pytest-run-changed
last release: Apr 02, 2021, *status*: 3 - Alpha, *requires*: pytest

Pytest plugin that runs changed tests only

pytest-runfailed
last release: Mar 24, 2016, *status*: N/A, *requires*: N/A

implement a –failed option for pytest

pytest-run-parallel
last release: Jan 14, 2026, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A simple pytest plugin to run tests concurrently

pytest-run-subprocess
last release: Nov 12, 2022, *status*: 5 - Production/Stable, *requires*: pytest

Pytest Plugin for running and testing subprocesses.

pytest-runtime-types
last release: Feb 09, 2023, *status*: N/A, *requires*: pytest

Checks type annotations on runtime while running tests.

pytest-runtime-xfail
last release: Oct 10, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=5.0.0

Call runtime_xfail() to mark running test as xfail.

pytest-runtime-yoyo
last release: Jun 12, 2023, *status*: N/A, *requires*: pytest (>=7.2.0)

run case mark timeout

pytest-rxdist
last release: Mar 17, 2026, *status*: 1 - Planning, *requires*: N/A

Rust-accelerated next-generation execution engine for pytest (planned).

pytest-saccharin
last release: Oct 31, 2022, *status*: 3 - Alpha, *requires*: N/A

pytest-saccharin is a updated fork of pytest-sugar, a plugin for pytest that changes the default look and feel of pytest (e.g. progressbar, show tests that fail instantly).

pytest-salt
last release: Jan 27, 2020, *status*: 4 - Beta, *requires*: N/A

Pytest Salt Plugin

pytest-salt-containers
last release: Nov 09, 2016, *status*: 4 - Beta, *requires*: N/A

A Pytest plugin that builds and creates docker containers

pytest-salt-factories
last release: Jul 08, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.4.0

Pytest Salt Plugin

pytest-salt-from-filenames
last release: Jan 29, 2019, *status*: 4 - Beta, *requires*: pytest (>=4.1)

Simple PyTest Plugin For Salt's Test Suite Specifically

pytest-salt-runtests-bridge
last release: Dec 05, 2019, *status*: 4 - Beta, *requires*: pytest (>=4.1)

Simple PyTest Plugin For Salt's Test Suite Specifically

pytest-sample-argvalues
last release: May 07, 2024, *status*: N/A, *requires*: pytest

A utility function to help choose a random sample from your argvalues in pytest.

pytest-sanic
last release: Oct 25, 2021, *status*: N/A, *requires*: pytest (>=5.2)

a pytest plugin for Sanic

pytest-sanitizer
last release: Mar 16, 2025, *status*: 3 - Alpha, *requires*: pytest>=6.0.0

A pytest plugin to sanitize output for LLMs (personal tool, no warranty or liability)

pytest-sanity
last release: Dec 07, 2020, *status*: N/A, *requires*: N/A

pytest-sa-pg
last release: May 14, 2019, *status*: N/A, *requires*: N/A

pytest_sauce
last release: Jul 14, 2014, *status*: 3 - Alpha, *requires*: N/A

pytest_sauce provides sane and helpful methods worked out in clearcode to run py.test tests with selenium/saucelabs

pytest-sbase
last release: Apr 18, 2026, *status*: 5 - Production/Stable, *requires*: N/A

A complete web automation framework for end-to-end testing.

pytest-scenario
last release: Feb 06, 2017, *status*: 3 - Alpha, *requires*: N/A

pytest plugin for test scenarios

pytest-scenario-files
last release: Feb 17, 2026, *status*: 5 - Production/Stable, *requires*: pytest<10,>=7.4

A pytest plugin that generates unit test scenarios from data files.

pytest-scenarios
last release: Jan 03, 2026, *status*: N/A, *requires*: N/A

Add your description here

pytest-schedule
last release: Oct 31, 2024, *status*: N/A, *requires*: N/A

Automate and customize test scheduling effortlessly on local machines.

pytest-schema
last release: Feb 16, 2024, *status*: 5 - Production/Stable, *requires*: pytest >=3.5.0

👍 Validate return values against a schema-like object in testing

pytest-scim2-server
last release: Nov 14, 2025, *status*: 4 - Beta, *requires*: pytest>=8.3.4

SCIM2 server fixture for Pytest

pytest-screenshot-on-failure
last release: Jul 21, 2023, *status*: 4 - Beta, *requires*: N/A

Saves a screenshot when a test case from a pytest execution fails

pytest-scrutinize
last release: Aug 19, 2024, *status*: 4 - Beta, *requires*: pytest>=6

Scrutinize your pytest test suites for slow fixtures, tests and more.

pytest-securestore
last release: Nov 08, 2021, *status*: 4 - Beta, *requires*: N/A

An encrypted password store for use within pytest cases

pytest-select
last release: Jan 18, 2019, *status*: 3 - Alpha, *requires*: pytest (>=3.0)

A pytest plugin which allows to (de-)select tests from a file.

pytest-selenium
last release: Feb 01, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=6.0.0

pytest plugin for Selenium

pytest-selenium-auto
last release: Nov 07, 2023, *status*: N/A, *requires*: pytest >= 7.0.0

pytest plugin to automatically capture screenshots upon selenium webdriver events

pytest-seleniumbase
last release: Apr 18, 2026, *status*: 5 - Production/Stable, *requires*: N/A

A complete web automation framework for end-to-end testing.

pytest-selenium-driver
last release: Mar 07, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

A zero-boilerplate Selenium WebDriver fixture for pytest

pytest-selenium-enhancer
last release: Apr 29, 2022, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for Selenium

pytest-selenium-pdiff
last release: Apr 06, 2017, *status*: 2 - Pre-Alpha, *requires*: N/A

A pytest package implementing perceptualdiff for Selenium tests.

pytest-selfie
last release: Dec 16, 2024, *status*: N/A, *requires*: pytest>=8.0.0

A pytest plugin for selfie snapshot testing.

pytest-semantic
last release: Nov 11, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

A pytest plugin for testing LLM outputs using semantic similarity matching

pytest-semantic-assert
last release: Jan 09, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

Pytest plugin for semantic LLM output assertions using embeddings. Test meaning, not strings.

pytest-semantic-llm
last release: Mar 10, 2026, *status*: N/A, *requires*: pytest>=7.0.0

Semantic assertions for pytest using LLMs

pytest-semantix
last release: Apr 13, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

pytest plugin for semantic LLM output testing — validate meaning, not just shape.

pytest-send-email
last release: Sep 02, 2024, *status*: N/A, *requires*: pytest

Send pytest execution result email

pytest-sentry
last release: Jul 01, 2025, *status*: N/A, *requires*: pytest

A pytest plugin to send testrun information to Sentry.io

pytest-sequence-markers
last release: May 23, 2023, *status*: 5 - Production/Stable, *requires*: N/A

Pytest plugin for sequencing markers for execution of tests

pytest-server
last release: Sep 09, 2024, *status*: N/A, *requires*: N/A

test server exec cmd

pytest-server-fixtures
last release: Nov 29, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Extensible server fixtures for py.test

pytest-serverless
last release: May 09, 2022, *status*: 4 - Beta, *requires*: N/A

Automatically mocks resources from serverless.yml in pytest using moto.

pytest-servers
last release: Dec 21, 2025, *status*: 3 - Alpha, *requires*: pytest>=6.2

pytest servers

pytest-service
last release: Aug 06, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=6.0.0

pytest-services
last release: Jul 16, 2025, *status*: 6 - Mature, *requires*: pytest

Services plugin for pytest testing framework

pytest-session2file
last release: Jan 26, 2021, *status*: 3 - Alpha, *requires*: pytest

pytest-session2file (aka: pytest-session_to_file for v0.1.0 - v0.1.2) is a py.test plugin for capturing and saving to file the stdout of py.test.

pytest-session-fixture-globalize
last release: May 15, 2018, *status*: 4 - Beta, *requires*: N/A

py.test plugin to make session fixtures behave as if written in conftest, even if it is written in some modules

pytest-session_to_file
last release: Oct 01, 2015, *status*: 3 - Alpha, *requires*: N/A

pytest-session_to_file is a py.test plugin for capturing and saving to file the stdout of py.test.

pytest-setupinfo
last release: Jan 23, 2023, *status*: N/A, *requires*: N/A

Displaying setup info during pytest command run

pytest-sftpserver
last release: Sep 16, 2019, *status*: 4 - Beta, *requires*: N/A

py.test plugin to locally test sftp server connections.

pytest-shard
last release: Dec 11, 2020, *status*: 4 - Beta, *requires*: pytest

pytest-shard-cloudc
last release: Apr 02, 2026, *status*: 4 - Beta, *requires*: pytest

Shard tests to support parallelism across multiple machines.

pytest-shard-fork
last release: Jun 13, 2025, *status*: 4 - Beta, *requires*: pytest

Shard tests to support parallelism across multiple machines

pytest-shared-session-scope
last release: Oct 31, 2025, *status*: N/A, *requires*: pytest>=7.0.0

Pytest session-scoped fixture that works with xdist

pytest-share-hdf
last release: Sep 21, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Plugin to save test data in HDF files and retrieve them for comparison

pytest-sharkreport
last release: Jul 11, 2022, *status*: N/A, *requires*: pytest (>=3.5)

this is pytest report plugin.

pytest-shell
last release: Mar 27, 2022, *status*: N/A, *requires*: N/A

A pytest plugin to help with testing shell scripts / black box commands

pytest-shell-utilities
last release: Oct 22, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7.4.0

Pytest plugin to simplify running shell commands against the system

pytest-sheraf
last release: Feb 11, 2020, *status*: N/A, *requires*: pytest

Versatile ZODB abstraction layer - pytest fixtures

pytest-sherlock
last release: Aug 14, 2023, *status*: 5 - Production/Stable, *requires*: pytest >=3.5.1

pytest plugin help to find coupled tests

pytest-shortcuts
last release: Oct 29, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Expand command-line shortcuts listed in pytest configuration

pytest-shutil
last release: Nov 29, 2024, *status*: 5 - Production/Stable, *requires*: pytest

A goodie-bag of unix shell and environment tools for py.test

pytest-sigil
last release: Oct 21, 2025, *status*: N/A, *requires*: pytest<9.0.0,>=7.0.0

Proper fixture resource cleanup by handling signals

pytest-simbind
last release: Mar 28, 2024, *status*: N/A, *requires*: pytest>=7.0.0

Pytest plugin to operate with objects generated by Simbind tool.

pytest-simplehttpserver
last release: Jun 24, 2021, *status*: 4 - Beta, *requires*: N/A

Simple pytest fixture to spin up an HTTP server

pytest-simple-plugin
last release: Nov 27, 2019, *status*: N/A, *requires*: N/A

Simple pytest plugin

pytest-simple-settings
last release: Nov 17, 2020, *status*: 4 - Beta, *requires*: pytest

simple-settings plugin for pytest

pytest-simplified
last release: Jan 19, 2026, *status*: 4 - Beta, *requires*: pytest<9.0.0,>=8.3.5

A PyTest plugin to simplify testing classes.

pytest-single-file-logging
last release: May 05, 2016, *status*: 4 - Beta, *requires*: pytest (>=2.8.1)

Allow for multiple processes to log to a single file

pytest-skill-engineering
last release: Apr 18, 2026, *status*: 3 - Alpha, *requires*: pytest>=9.0

The testing framework for skill engineering. Test tool descriptions, prompt templates, agent skills, and custom agents with real LLMs. AI analyzes results and tells you what to fix.

pytest-skip
last release: Sep 12, 2025, *status*: 3 - Alpha, *requires*: pytest

A pytest plugin which allows to (de-)select or skip tests from a file.

pytest-skip-markers
last release: Aug 09, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7.1.0

Pytest Salt Plugin

pytest-skipper
last release: Mar 26, 2017, *status*: 3 - Alpha, *requires*: pytest (>=3.0.6)

A plugin that selects only tests with changes in execution path

pytest-skippy
last release: Jan 27, 2018, *status*: 3 - Alpha, *requires*: pytest (>=2.3.4)

Automatically skip tests that don't need to run!

pytest-skip-slow
last release: Feb 09, 2023, *status*: N/A, *requires*: pytest>=6.2.0

A pytest plugin to skip `@pytest.mark.slow` tests by default.

pytest-skipuntil
last release: Nov 25, 2023, *status*: 4 - Beta, *requires*: pytest >=3.8.0

A simple pytest plugin to skip flapping test with deadline

pytest-slack
last release: Dec 15, 2020, *status*: 5 - Production/Stable, *requires*: N/A

Pytest to Slack reporting plugin

pytest-slow
last release: Sep 28, 2021, *status*: N/A, *requires*: N/A

A pytest plugin to skip `@pytest.mark.slow` tests by default.

pytest-slowest-first
last release: Dec 11, 2022, *status*: 4 - Beta, *requires*: N/A

Sort tests by their last duration, slowest first

pytest-slow-first
last release: Jan 30, 2024, *status*: 4 - Beta, *requires*: pytest >=3.5.0

Prioritize running the slowest tests first.

pytest-slow-last
last release: Mar 16, 2025, *status*: 4 - Beta, *requires*: pytest>=3.5.0

Run tests in order of execution time (faster tests first)

pytest-smartcollect
last release: Oct 04, 2018, *status*: N/A, *requires*: pytest (>=3.5.0)

A plugin for collecting tests that touch changed code

pytest-smartcov
last release: Sep 30, 2017, *status*: 3 - Alpha, *requires*: N/A

Smart coverage plugin for pytest.

pytest-smart-debugger-backend
last release: Sep 17, 2025, *status*: N/A, *requires*: N/A

Backend server for Pytest Smart Debugger

pytest-smart-rerun
last release: Oct 12, 2025, *status*: 3 - Alpha, *requires*: N/A

A Pytest plugin for intelligent retrying of flaky tests.

pytest-smell
last release: Jun 26, 2022, *status*: N/A, *requires*: N/A

Automated bad smell detection tool for Pytest

pytest-smoke
last release: Nov 09, 2025, *status*: 4 - Beta, *requires*: pytest<10,>=7.0.0

Pytest plugin for smoke testing

pytest-smtp
last release: Feb 20, 2021, *status*: N/A, *requires*: pytest

Send email with pytest execution result

pytest-smtp4dev
last release: Jun 27, 2023, *status*: 5 - Production/Stable, *requires*: N/A

Plugin for smtp4dev API

pytest-smtpd
last release: May 15, 2023, *status*: N/A, *requires*: pytest

An SMTP server for testing built on aiosmtpd

pytest-smtp-test-server
last release: Dec 03, 2023, *status*: 2 - Pre-Alpha, *requires*: pytest (>=7.4.3,<8.0.0)

pytest plugin for using `smtp-test-server` as a fixture

pytest-snail
last release: Nov 04, 2019, *status*: 3 - Alpha, *requires*: pytest (>=5.0.1)

Plugin for adding a marker to slow running tests.

pytest-snap
last release: Aug 25, 2025, *status*: N/A, *requires*: pytest>=8.0.0

A text-based snapshot testing library implemented as a pytest plugin

pytest-snapcheck
last release: Sep 07, 2025, *status*: N/A, *requires*: pytest>=8.0

Minimal deterministic test-run snapshot capture for pytest.

pytest-snapci
last release: Nov 12, 2015, *status*: N/A, *requires*: N/A

py.test plugin for Snap-CI

pytest-snapmock
last release: Nov 15, 2024, *status*: N/A, *requires*: N/A

Snapshots for your mocks.

pytest-snapshot
last release: Apr 23, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.0.0)

A plugin for snapshot testing with pytest.

pytest-snapshot-with-message-generator
last release: Jul 25, 2023, *status*: 4 - Beta, *requires*: pytest (>=3.0.0)

A plugin for snapshot testing with pytest.

pytest-snmpserver
last release: May 12, 2021, *status*: N/A, *requires*: N/A

pytest-snob
last release: Jan 12, 2025, *status*: N/A, *requires*: pytest

A pytest plugin that only selects meaningful python tests to run.

pytest-snowflake-bdd
last release: Jan 05, 2022, *status*: 4 - Beta, *requires*: pytest (>=6.2.0)

Setup test data and run tests on snowflake in BDD style!

pytest-socket
last release: Jan 28, 2024, *status*: 4 - Beta, *requires*: pytest (>=6.2.5)

Pytest Plugin to disable socket calls during tests

pytest-sofaepione
last release: Aug 17, 2022, *status*: N/A, *requires*: N/A

Test the installation of SOFA and the SofaEpione plugin.

pytest-soft-assert
last release: Dec 07, 2025, *status*: N/A, *requires*: pytest>=8.4.0

Pytest plugin for soft assertions.

pytest-soft-assertions
last release: May 05, 2020, *status*: 3 - Alpha, *requires*: pytest

pytest-solidity
last release: Jan 15, 2022, *status*: 1 - Planning, *requires*: pytest (<7,>=6.0.1) ; extra == 'tests'

A PyTest library plugin for Solidity language.

pytest-solr
last release: May 11, 2020, *status*: 3 - Alpha, *requires*: pytest (>=3.0.0)

Solr process and client fixtures for py.test.

pytest-sort
last release: Apr 04, 2026, *status*: N/A, *requires*: pytest>=7.4.0

Tools for sorting test cases

pytest-sorter
last release: Apr 20, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.1.1)

A simple plugin to first execute tests that historically failed more

pytest-sosu
last release: Aug 04, 2023, *status*: 2 - Pre-Alpha, *requires*: pytest

Unofficial PyTest plugin for Sauce Labs

pytest-sourceorder
last release: Sep 01, 2021, *status*: 4 - Beta, *requires*: pytest

Test-ordering plugin for pytest

pytest-spark
last release: May 21, 2025, *status*: 4 - Beta, *requires*: pytest

pytest plugin to run the tests with support of pyspark.

pytest-spawner
last release: Jul 31, 2015, *status*: 4 - Beta, *requires*: N/A

py.test plugin to spawn process and communicate with them.

pytest-spec
last release: Feb 22, 2026, *status*: N/A, *requires*: pytest; extra == "test"

Library pytest-spec is a pytest plugin to display test execution output like a SPECIFICATION.

pytest-spec2md
last release: Apr 10, 2024, *status*: N/A, *requires*: pytest>7.0

Library pytest-spec2md is a pytest plugin to create a markdown specification while running pytest.

pytest-speed
last release: Jan 22, 2023, *status*: 3 - Alpha, *requires*: pytest>=7

Modern benchmarking library for python with pytest integration.

pytest-sphinx
last release: Jan 21, 2026, *status*: 4 - Beta, *requires*: pytest>=8.1.1

Doctest plugin for pytest with support for Sphinx-specific doctest-directives

pytest-spiratest
last release: Feb 09, 2026, *status*: N/A, *requires*: pytest>=3.0.0

Exports unit tests as test runs in Spira (SpiraTest/Team/Plan)

pytest-splinter
last release: Sep 09, 2022, *status*: 6 - Mature, *requires*: pytest (>=3.0.0)

Splinter plugin for pytest testing framework

pytest-splinter4
last release: Feb 01, 2024, *status*: 6 - Mature, *requires*: pytest >=8.0.0

Pytest plugin for the splinter automation library

pytest-split
last release: Feb 03, 2026, *status*: 4 - Beta, *requires*: pytest<10,>=5

Pytest plugin which splits the test suite to equally sized sub suites based on test execution time.

pytest-split-ct
last release: Jan 23, 2026, *status*: 4 - Beta, *requires*: pytest<10,>=5

Pytest plugin which splits the test suite to equally sized sub suites based on test execution time.

pytest-split-ext
last release: Sep 23, 2023, *status*: 4 - Beta, *requires*: pytest (>=5,<8)

Pytest plugin which splits the test suite to equally sized sub suites based on test execution time.

pytest-splitio
last release: Sep 22, 2020, *status*: N/A, *requires*: pytest (<7,>=5.0)

Split.io SDK integration for e2e tests

pytest-split-ng
last release: Jan 05, 2026, *status*: 4 - Beta, *requires*: pytest<10,>=5

Pytest plugin which splits the test suite to equally sized sub suites based on test execution time.

pytest-split-tests
last release: Jul 30, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=2.5)

A Pytest plugin for running a subset of your tests by splitting them in to equally sized groups. Forked from Mark Adams' original project pytest-test-groups.

pytest-split-tests-tresorit
last release: Feb 22, 2021, *status*: 1 - Planning, *requires*: N/A

pytest-split-v2
last release: Jan 14, 2026, *status*: 4 - Beta, *requires*: pytest<10,>=5

Pytest plugin which splits the test suite to equally sized sub suites based on test execution time.

pytest-splunk-addon
last release: Apr 01, 2026, *status*: N/A, *requires*: pytest<8,>5.4.0

A Dynamic test tool for Splunk Apps and Add-ons

pytest-splunk-addon-ui-smartx
last release: Mar 25, 2026, *status*: N/A, *requires*: N/A

Library to support testing Splunk Add-on UX

pytest-splunk-env
last release: Oct 22, 2020, *status*: N/A, *requires*: pytest (>=6.1.1,<7.0.0)

pytest fixtures for interaction with Splunk Enterprise and Splunk Cloud

pytest-sqitch
last release: Apr 06, 2020, *status*: 4 - Beta, *requires*: N/A

sqitch for pytest

pytest-sqlalchemy
last release: Apr 19, 2025, *status*: 3 - Alpha, *requires*: pytest>=8.0

pytest plugin with sqlalchemy related fixtures

pytest-sqlalchemy-mock
last release: Aug 10, 2024, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

pytest sqlalchemy plugin for mock

pytest-sqlalchemy-session
last release: May 19, 2023, *status*: 4 - Beta, *requires*: pytest (>=7.0)

A pytest plugin for preserving test isolation that use SQLAlchemy.

pytest-sql-bigquery
last release: Dec 19, 2019, *status*: N/A, *requires*: pytest

Yet another SQL-testing framework for BigQuery provided by pytest plugin

pytest-sqlfluff
last release: Dec 21, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A pytest plugin to use sqlfluff to enable format checking of sql files.

pytest-sqlguard
last release: Jun 06, 2025, *status*: 4 - Beta, *requires*: pytest>=7

Pytest fixture to record and check SQL Queries made by SQLAlchemy

pytest-squadcast
last release: Feb 22, 2022, *status*: 5 - Production/Stable, *requires*: pytest

Pytest report plugin for Squadcast

pytest-srcpaths
last release: Oct 15, 2021, *status*: N/A, *requires*: pytest>=6.2.0

Add paths to sys.path

pytest-ssh
last release: May 27, 2019, *status*: N/A, *requires*: pytest

pytest plugin for ssh command run

pytest-start-from
last release: Apr 11, 2016, *status*: N/A, *requires*: N/A

Start pytest run from a given point

pytest-static
last release: May 25, 2025, *status*: 3 - Alpha, *requires*: pytest<8.0.0,>=7.4.3

pytest-static

pytest-stats
last release: Jul 18, 2024, *status*: N/A, *requires*: pytest>=8.0.0

Collects tests metadata for future analysis, easy to extend for any data store

pytest-statsd
last release: Nov 30, 2018, *status*: 5 - Production/Stable, *requires*: pytest (>=3.0.0)

pytest plugin for reporting to graphite

pytest-status
last release: Aug 22, 2024, *status*: N/A, *requires*: pytest

Add status mark for tests

pytest-stderr-db
last release: Sep 14, 2025, *status*: N/A, *requires*: N/A

Add your description here

pytest-stdout-db
last release: Sep 14, 2025, *status*: N/A, *requires*: N/A

Add your description here

pytest-stepfunctions
last release: May 08, 2021, *status*: 4 - Beta, *requires*: pytest

A small description

pytest-step-logger
last release: Mar 11, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Live Rich-rendered step trees in pytest terminal output

pytest-steps
last release: Sep 23, 2021, *status*: 5 - Production/Stable, *requires*: N/A

Create step-wise / incremental tests in pytest.

pytest-stepthrough
last release: Aug 14, 2025, *status*: N/A, *requires*: N/A

Pause and wait for Enter after each test with –step

pytest-stepwise
last release: Dec 01, 2015, *status*: 4 - Beta, *requires*: N/A

Run a test suite one failing test at a time.

pytest-stf
last release: Sep 23, 2025, *status*: N/A, *requires*: pytest>=5.0

pytest plugin for openSTF

pytest-stochastic
last release: Feb 24, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0

A pytest plugin for principled stochastic unit testing using concentration inequalities

pytest-stochastics
last release: Dec 01, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.0.0

pytest plugin that allows selectively running tests several times and accepting *some* failures.

pytest-stoq
last release: Feb 09, 2021, *status*: 4 - Beta, *requires*: N/A

A plugin to pytest stoq

pytest-storage
last release: Sep 12, 2025, *status*: 3 - Alpha, *requires*: pytest>=8.4.2

Pytest plugin to store test artifacts

pytest-store
last release: Jul 30, 2025, *status*: 3 - Alpha, *requires*: pytest>=7.0.0

Pytest plugin to store values from test runs

pytest-streaming
last release: Jan 14, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.3.5

Plugin for testing pubsub, pulsar, and kafka systems with pytest locally and in ci/cd

pytest-stress
last release: Dec 07, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.6.0)

A Pytest plugin that allows you to loop tests for a user defined amount of time.

pytest-structlog
last release: Sep 10, 2025, *status*: N/A, *requires*: pytest

Structured logging assertions

pytest-structmpd
last release: Oct 17, 2018, *status*: N/A, *requires*: N/A

provide structured temporary directory

pytest-stub
last release: Apr 28, 2020, *status*: 5 - Production/Stable, *requires*: N/A

Stub packages, modules and attributes.

pytest-stubprocess
last release: Sep 17, 2018, *status*: 3 - Alpha, *requires*: pytest (>=3.5.0)

Provide stub implementations for subprocesses in Python tests

pytest-study
last release: Sep 26, 2017, *status*: 3 - Alpha, *requires*: pytest (>=2.0)

A pytest plugin to organize long run tests (named studies) without interfering the regular tests

pytest-subinterpreter
last release: Nov 25, 2023, *status*: N/A, *requires*: pytest>=7.0.0

Run pytest in a subinterpreter

pytest-subket
last release: Jul 31, 2025, *status*: 4 - Beta, *requires*: N/A

Pytest Plugin to disable socket calls during tests

pytest-subprocess
last release: Mar 21, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=4.0.0

A plugin to fake subprocess for pytest

pytest-subtesthack
last release: Jul 16, 2022, *status*: N/A, *requires*: N/A

A hack to explicitly set up and tear down fixtures.

pytest-subtests
last release: Oct 20, 2025, *status*: 4 - Beta, *requires*: pytest>=7.4

unittest subTest() support and subtests fixture

pytest-subunit
last release: Sep 17, 2023, *status*: N/A, *requires*: pytest (>=2.3)

pytest-subunit is a plugin for py.test which outputs testsresult in subunit format.

pytest-sugar
last release: Aug 23, 2025, *status*: 4 - Beta, *requires*: pytest>=6.2.0

pytest-sugar is a plugin for pytest that changes the default look and feel of pytest (e.g. progressbar, show tests that fail instantly).

pytest-suitemanager
last release: Apr 28, 2023, *status*: 4 - Beta, *requires*: N/A

A simple plugin to use with pytest

pytest-suite-timeout
last release: Jan 26, 2024, *status*: N/A, *requires*: pytest>=7.0.0

A pytest plugin for ensuring max suite time

pytest-supercov
last release: Jul 02, 2023, *status*: N/A, *requires*: N/A

Pytest plugin for measuring explicit test-file to source-file coverage

pytest-svn
last release: Oct 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

SVN repository fixture for py.test

pytest-swag
last release: Apr 14, 2026, *status*: 3 - Alpha, *requires*: pytest>=7.0

Generate OpenAPI documentation from pytest tests

pytest-symbols
last release: Nov 20, 2017, *status*: 3 - Alpha, *requires*: N/A

pytest-symbols is a pytest plugin that adds support for passing test environment symbols into pytest tests.

pytest-system-statistics
last release: Feb 16, 2022, *status*: 5 - Production/Stable, *requires*: pytest (>=6.0.0)

Pytest plugin to track and report system usage statistics

pytest-system-test-plugin
last release: Feb 03, 2022, *status*: N/A, *requires*: N/A

Pyst - Pytest System-Test Plugin

pytest_tagging
last release: Nov 08, 2024, *status*: N/A, *requires*: pytest>=7.1.3

a pytest plugin to tag tests

pytest-takeltest
last release: Sep 07, 2024, *status*: N/A, *requires*: N/A

Fixtures for ansible, testinfra and molecule

pytest-talisker
last release: Nov 28, 2021, *status*: N/A, *requires*: N/A

pytest-tally
last release: May 22, 2023, *status*: 4 - Beta, *requires*: pytest (>=6.2.5)

A Pytest plugin to generate realtime summary stats, and display them in-console using a text-based dashboard.

pytest-tap
last release: Jan 30, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=3.0

Test Anything Protocol (TAP) reporting plugin for pytest

pytest-tape
last release: Mar 17, 2021, *status*: 4 - Beta, *requires*: N/A

easy assertion with expected results saved to yaml files

pytest-target
last release: Jan 21, 2021, *status*: 3 - Alpha, *requires*: pytest (>=6.1.2,<7.0.0)

Pytest plugin for remote target orchestration.

pytest-taskgraph
last release: Apr 09, 2025, *status*: N/A, *requires*: pytest

Add your description here

pytest-tblineinfo
last release: Dec 01, 2015, *status*: 3 - Alpha, *requires*: pytest (>=2.0)

tblineinfo is a py.test plugin that insert the node id in the final py.test report when –tb=line option is used

pytest-tcpclient
last release: Nov 16, 2022, *status*: N/A, *requires*: pytest (<8,>=7.1.3)

A pytest plugin for testing TCP clients

pytest-tdd
last release: Aug 18, 2023, *status*: 4 - Beta, *requires*: N/A

run pytest on a python module

pytest-teamcity-logblock
last release: May 15, 2018, *status*: 4 - Beta, *requires*: N/A

py.test plugin to introduce block structure in teamcity build log, if output is not captured

pytest-teardown
last release: Apr 15, 2025, *status*: N/A, *requires*: pytest<9.0.0,>=7.4.1

pytest-telegram
last release: Apr 25, 2024, *status*: 5 - Production/Stable, *requires*: N/A

Pytest to Telegram reporting plugin

pytest-telegram-notifier
last release: Jun 27, 2023, *status*: 5 - Production/Stable, *requires*: N/A

Telegram notification plugin for Pytest

pytest-tempdir
last release: Oct 11, 2019, *status*: 4 - Beta, *requires*: pytest (>=2.8.1)

Predictable and repeatable tempdir support.

pytest-terra-fixt
last release: Sep 15, 2022, *status*: N/A, *requires*: pytest (==6.2.5)

Terraform and Terragrunt fixtures for pytest

pytest-terraform
last release: May 21, 2024, *status*: N/A, *requires*: pytest>=6.0

A pytest plugin for using terraform fixtures

pytest-terraform-fixture
last release: Nov 14, 2018, *status*: 4 - Beta, *requires*: N/A

generate terraform resources to use with pytest

pytest-test-analyzer
last release: Jun 14, 2025, *status*: 4 - Beta, *requires*: N/A

A powerful tool for analyzing pytest test files and generating detailed reports

pytest-testbook
last release: Dec 11, 2016, *status*: 3 - Alpha, *requires*: N/A

A plugin to run tests written in Jupyter notebook

pytest-test-categories
last release: Mar 04, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.4.2

A pytest plugin to enforce test timing constraints and size distributions.

pytest-testconfig
last release: Jan 11, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Test configuration plugin for pytest.

pytest-testcontainers-compose
last release: Feb 11, 2026, *status*: N/A, *requires*: N/A

Pytest plugin for Docker Compose

pytest-testdata
last release: Aug 30, 2024, *status*: N/A, *requires*: pytest

Get and load testdata in pytest projects

pytest-testdirectory
last release: May 02, 2023, *status*: 5 - Production/Stable, *requires*: pytest

A py.test plugin providing temporary directories in unit tests.

pytest-testdox
last release: Jul 22, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=4.6.0)

A testdox format reporter for pytest

pytest-test-grouping
last release: Feb 01, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=2.5)

A Pytest plugin for running a subset of your tests by splitting them in to equally sized groups.

pytest-test-groups
last release: May 08, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

A Pytest plugin for running a subset of your tests by splitting them in to equally sized groups.

pytest-testinel
last release: Mar 13, 2026, *status*: 4 - Beta, *requires*: pytest>=7

Testinel's pytest plugin captures structured test execution data directly from pytest and sends it to Testinel, where your test results become searchable, comparable, and actually useful.

pytest-testinfra
last release: Mar 30, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=6

Test infrastructures

pytest-testinfra-jpic
last release: Sep 21, 2023, *status*: 5 - Production/Stable, *requires*: N/A

Test infrastructures

pytest-testinfra-winrm-transport
last release: Sep 21, 2023, *status*: 5 - Production/Stable, *requires*: N/A

Test infrastructures

pytest-testit-parametrize
last release: Dec 04, 2024, *status*: 4 - Beta, *requires*: pytest>=8.3.3

A pytest plugin for uploading parameterized tests parameters into TMS TestIT

pytest-testlink-adaptor
last release: Dec 20, 2018, *status*: 4 - Beta, *requires*: pytest (>=2.6)

pytest reporting plugin for testlink

pytest-testmon
last release: Dec 01, 2025, *status*: 4 - Beta, *requires*: pytest<10,>=5

selects tests affected by changed files and methods

pytest-testmon-dev
last release: Mar 30, 2023, *status*: 4 - Beta, *requires*: pytest (<8,>=5)

selects tests affected by changed files and methods

pytest-testmon-oc
last release: Jun 01, 2022, *status*: 4 - Beta, *requires*: pytest (<8,>=5)

nOly selects tests affected by changed files and methods

pytest-testmon-skip-libraries
last release: Mar 03, 2023, *status*: 4 - Beta, *requires*: pytest (<8,>=5)

selects tests affected by changed files and methods

pytest-testobject
last release: Sep 24, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.1.1)

Plugin to use TestObject Suites with Pytest

pytest-testpluggy
last release: Jan 07, 2022, *status*: N/A, *requires*: pytest

set your encoding

pytest-testrail
last release: Jan 25, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

A pytest plugin for creating TestRail runs and adding results

pytest-testrail2
last release: Feb 10, 2023, *status*: N/A, *requires*: pytest (<8.0,>=7.2.0)

A pytest plugin to upload results to TestRail.

pytest-testrail-api
last release: Mar 17, 2025, *status*: N/A, *requires*: pytest

TestRail Api Python Client

pytest-testrail-api-client
last release: Dec 14, 2021, *status*: N/A, *requires*: pytest

TestRail Api Python Client

pytest-testrail-appetize
last release: Sep 29, 2021, *status*: N/A, *requires*: N/A

pytest plugin for creating TestRail runs and adding results

pytest-testrail-client
last release: Sep 29, 2020, *status*: 5 - Production/Stable, *requires*: N/A

pytest plugin for Testrail

pytest-testrail-e2e
last release: Oct 11, 2021, *status*: N/A, *requires*: pytest (>=3.6)

pytest plugin for creating TestRail runs and adding results

pytest-testrail-integrator
last release: Aug 01, 2022, *status*: N/A, *requires*: pytest (>=6.2.5)

Pytest plugin for sending report to testrail system.

pytest-testrail-ns
last release: Aug 12, 2022, *status*: N/A, *requires*: N/A

pytest plugin for creating TestRail runs and adding results

pytest-testrail-reporter
last release: Sep 10, 2018, *status*: N/A, *requires*: N/A

pytest-testrail-results
last release: Mar 04, 2024, *status*: N/A, *requires*: pytest >=7.2.0

A pytest plugin to upload results to TestRail.

pytest-testreport
last release: Dec 01, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

pytest-testreport-new
last release: Oct 07, 2023, *status*: 4 - Beta, *requires*: pytest >=3.5.0

pytest-testslide
last release: Jan 07, 2021, *status*: 5 - Production/Stable, *requires*: pytest (~=6.2)

TestSlide fixture for pytest

pytest-test-this
last release: Sep 15, 2019, *status*: 2 - Pre-Alpha, *requires*: pytest (>=2.3)

Plugin for py.test to run relevant tests, based on naively checking if a test contains a reference to the symbol you supply

pytest-test-tracer-for-pytest
last release: Jun 28, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin that allows coll test data for use on Test Tracer

pytest-test-tracer-for-pytest-bdd
last release: Aug 20, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin that allows coll test data for use on Test Tracer

pytest-test-utils
last release: Feb 08, 2024, *status*: N/A, *requires*: pytest >=3.9

pytest-tesults
last release: Nov 12, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=3.5.0

Tesults plugin for pytest

pytest-texts-score
last release: Dec 17, 2025, *status*: 4 - Beta, *requires*: pytest>=8.4.2

Texts content similarity scoring plugin

pytest-textual-snapshot
last release: Jan 23, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=8.0.0

Snapshot testing for Textual apps

pytest-tezos
last release: Jan 16, 2020, *status*: 4 - Beta, *requires*: N/A

pytest-ligo

pytest-tf
last release: May 29, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.2.1

Test your OpenTofu and Terraform config using a PyTest plugin

pytest-th2-bdd
last release: May 13, 2022, *status*: N/A, *requires*: N/A

pytest_th2_bdd

pytest-thawgun
last release: May 26, 2020, *status*: 3 - Alpha, *requires*: N/A

Pytest plugin for time travel

pytest-thread
last release: Jul 07, 2023, *status*: N/A, *requires*: N/A

pytest-threadleak
last release: Jul 03, 2022, *status*: 4 - Beta, *requires*: pytest (>=3.1.1)

Detects thread leaks

pytest-threadpool
last release: Mar 19, 2026, *status*: 4 - Beta, *requires*: pytest<=9.0.2,>=9.0.0

Parallel test execution using threads — true parallelism on free-threaded Python, concurrent I/O on standard builds

pytest-tick
last release: Aug 31, 2021, *status*: 5 - Production/Stable, *requires*: pytest (>=6.2.5,<7.0.0)

Ticking on tests

pytest_time
last release: Dec 01, 2025, *status*: 3 - Alpha, *requires*: pytest

pytest-timeassert-ethan
last release: Dec 25, 2023, *status*: N/A, *requires*: pytest

execution duration

pytest-timeit
last release: Oct 13, 2016, *status*: 4 - Beta, *requires*: N/A

A pytest plugin to time test function runs

pytest-timeout
last release: May 05, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

pytest plugin to abort hanging tests

pytest-timeouts
last release: Sep 21, 2019, *status*: 5 - Production/Stable, *requires*: N/A

Linux-only Pytest plugin to control durations of various test case execution phases

pytest-timer
last release: Dec 26, 2023, *status*: N/A, *requires*: pytest

A timer plugin for pytest

pytest-timestamper
last release: Mar 27, 2024, *status*: N/A, *requires*: N/A

Pytest plugin to add a timestamp prefix to the pytest output

pytest-timestamps
last release: Sep 11, 2023, *status*: N/A, *requires*: pytest (>=7.3,<8.0)

A simple plugin to view timestamps for each test

pytest-timing-plugin
last release: Jul 21, 2025, *status*: N/A, *requires*: N/A

pytest插件开发demo

pytest-tiny-api-client
last release: Jan 04, 2024, *status*: 5 - Production/Stable, *requires*: pytest

The companion pytest plugin for tiny-api-client

pytest-tinybird
last release: May 07, 2025, *status*: 4 - Beta, *requires*: pytest>=3.8.0

A pytest plugin to report test results to tinybird

pytest-tipsi-django
last release: Feb 05, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=6.0.0

Better fixtures for django

pytest-tipsi-testing
last release: Feb 04, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=3.3.0

Better fixtures management. Various helpers

pytest-tldr
last release: Nov 10, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin that limits the output to just the things you need.

pytest-tm4j-reporter
last release: Sep 01, 2020, *status*: N/A, *requires*: pytest

Cloud Jira Test Management (TM4J) PyTest reporter plugin

pytest-tmnet
last release: Mar 01, 2022, *status*: N/A, *requires*: N/A

A small example package

pytest-tmp-files
last release: Dec 08, 2023, *status*: N/A, *requires*: pytest

Utilities to create temporary file hierarchies in pytest.

pytest-tmpfs
last release: Aug 29, 2022, *status*: N/A, *requires*: pytest

A pytest plugin that helps you on using a temporary filesystem for testing.

pytest-tmreport
last release: Aug 12, 2022, *status*: N/A, *requires*: N/A

this is a vue-element ui report for pytest

pytest-tmux
last release: Sep 01, 2025, *status*: 4 - Beta, *requires*: N/A

A pytest plugin that enables tmux driven tests

pytest-todo
last release: May 23, 2019, *status*: 4 - Beta, *requires*: pytest

A small plugin for the pytest testing framework, marking TODO comments as failure

pytest-tomato
last release: Mar 01, 2019, *status*: 5 - Production/Stable, *requires*: N/A

pytest-toolbelt
last release: Aug 12, 2019, *status*: 3 - Alpha, *requires*: N/A

This is just a collection of utilities for pytest, but don't really belong in pytest proper.

pytest-toolbox
last release: Apr 07, 2018, *status*: N/A, *requires*: pytest (>=3.5.0)

Numerous useful plugins for pytest.

pytest-toolkit
last release: Jun 07, 2024, *status*: N/A, *requires*: N/A

Useful utils for testing

pytest-tools
last release: Oct 21, 2022, *status*: 4 - Beta, *requires*: N/A

Pytest tools

pytest-topo
last release: Jun 05, 2024, *status*: N/A, *requires*: pytest>=7.0.0

Topological sorting for pytest

pytest-tornado
last release: Jun 17, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=3.6)

A py.test plugin providing fixtures and markers to simplify testing of asynchronous tornado applications.

pytest-tornado5
last release: Nov 16, 2018, *status*: 5 - Production/Stable, *requires*: pytest (>=3.6)

A py.test plugin providing fixtures and markers to simplify testing of asynchronous tornado applications.

pytest-tornado-yen3
last release: Oct 15, 2018, *status*: 5 - Production/Stable, *requires*: N/A

A py.test plugin providing fixtures and markers to simplify testing of asynchronous tornado applications.

pytest-tornasync
last release: Jul 15, 2019, *status*: 3 - Alpha, *requires*: pytest (>=3.0)

py.test plugin for testing Python 3.5+ Tornado code

pytest-trace
last release: Jun 19, 2022, *status*: N/A, *requires*: pytest (>=4.6)

Save OpenTelemetry spans generated during testing

pytest-track
last release: Feb 26, 2021, *status*: 3 - Alpha, *requires*: pytest (>=3.0)

pytest-translate
last release: Apr 08, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

pytest terminal output in your language — 134 languages supported, zero configuration

pytest-translations
last release: Sep 11, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=7)

Test your translation files.

pytest-travis-fold
last release: Nov 29, 2017, *status*: 4 - Beta, *requires*: pytest (>=2.6.0)

Folds captured output sections in Travis CI build log

pytest-trello
last release: Nov 20, 2015, *status*: 5 - Production/Stable, *requires*: N/A

Plugin for py.test that integrates trello using markers

pytest-trepan
last release: Sep 11, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=4.0.0

Pytest plugin for trepan debugger.

pytest-trialtemp
last release: Jun 08, 2015, *status*: N/A, *requires*: N/A

py.test plugin for using the same _trial_temp working directory as trial

pytest-trio
last release: Nov 01, 2022, *status*: N/A, *requires*: pytest (>=7.2.0)

Pytest plugin for trio

pytest-trytond
last release: Nov 04, 2022, *status*: 4 - Beta, *requires*: pytest (>=5)

Pytest plugin for the Tryton server framework

pytest-tspwplib
last release: Jan 08, 2021, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

A simple plugin to use with tspwplib

pytest-tst
last release: Apr 27, 2022, *status*: N/A, *requires*: pytest (>=5.0.0)

Customize pytest options, output and exit code to make it compatible with tst

pytest-tstcls
last release: Mar 23, 2020, *status*: 5 - Production/Stable, *requires*: N/A

Test Class Base

pytest-tui
last release: Dec 08, 2023, *status*: 4 - Beta, *requires*: N/A

Text User Interface (TUI) and HTML report for Pytest test runs

pytest-tui-runner
last release: Dec 12, 2025, *status*: N/A, *requires*: pytest<=9.0.1,>=7.4

Textual-based terminal UI for running pytest tests

pytest-tuitest
last release: Apr 11, 2025, *status*: N/A, *requires*: pytest>=7.4.0

pytest plugin for testing TUI and regular command-line applications.

pytest-tutorials
last release: Mar 11, 2023, *status*: N/A, *requires*: N/A

pytest-twilio-conversations-client-mock
last release: Aug 02, 2022, *status*: N/A, *requires*: N/A

pytest-twisted
last release: Sep 10, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=2.3

A twisted plugin for pytest.

pytest-ty
last release: Mar 08, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin to run the ty type checker

pytest-typechecker
last release: Feb 04, 2022, *status*: N/A, *requires*: pytest (>=6.2.5,<7.0.0)

Run type checkers on specified test files

pytest-typed-schema-shot
last release: Jun 14, 2025, *status*: N/A, *requires*: pytest

Pytest plugin for automatic JSON Schema generation and validation from examples

pytest-typhoon-config
last release: Apr 07, 2022, *status*: 5 - Production/Stable, *requires*: N/A

A Typhoon HIL plugin that facilitates test parameter configuration at runtime

pytest-typhoon-polarion
last release: Feb 01, 2024, *status*: 4 - Beta, *requires*: N/A

Typhoontest plugin for Siemens Polarion

pytest-typhoon-xray
last release: Aug 15, 2023, *status*: 4 - Beta, *requires*: N/A

Typhoon HIL plugin for pytest

pytest-typing
last release: Mar 24, 2026, *status*: 3 - Alpha, *requires*: pytest

Test your types by running typecheckers on them.

pytest-typing-runner
last release: May 31, 2025, *status*: N/A, *requires*: N/A

Pytest plugin to make it easier to run and check python code against static type checkers

pytest-tytest
last release: May 25, 2020, *status*: 4 - Beta, *requires*: pytest (>=5.4.2)

Typhoon HIL plugin for pytest

pytest-tzshift
last release: Jun 25, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0

A Pytest plugin that transparently re-runs tests under a matrix of timezones and locales.

pytest-ubersmith
last release: Apr 13, 2015, *status*: N/A, *requires*: N/A

Easily mock calls to ubersmith at the `requests` level.

pytest-ui
last release: Jul 05, 2021, *status*: 4 - Beta, *requires*: pytest

Text User Interface for running python tests

pytest-ui-failed-screenshot
last release: Dec 06, 2022, *status*: N/A, *requires*: N/A

UI自动测试失败时自动截图，并将截图加入到测试报告中

pytest-ui-failed-screenshot-allure
last release: Dec 06, 2022, *status*: N/A, *requires*: N/A

UI自动测试失败时自动截图，并将截图加入到Allure测试报告中

pytest-uncollect-if
last release: Dec 26, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

A plugin to uncollect pytests tests rather than using skipif

pytest-unflakable
last release: Apr 30, 2024, *status*: 4 - Beta, *requires*: pytest>=6.2.0

Unflakable plugin for PyTest

pytest-unhandled-exception-exit-code
last release: Jun 22, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=2.3)

Plugin for py.test set a different exit code on uncaught exceptions

pytest-unique
last release: Mar 27, 2026, *status*: N/A, *requires*: pytest>=9.0.0

Pytest fixture to generate unique values.

pytest-unittest-filter
last release: Jan 12, 2019, *status*: 4 - Beta, *requires*: pytest (>=3.1.0)

A pytest plugin for filtering unittest-based test classes

pytest-unittest-id-runner
last release: Feb 09, 2025, *status*: N/A, *requires*: pytest>=6.0.0

A pytest plugin to run tests using unittest-style test IDs

pytest-unmagic
last release: Jul 14, 2025, *status*: 5 - Production/Stable, *requires*: pytest

Pytest fixtures with conventional import semantics

pytest-unmarked
last release: Aug 27, 2019, *status*: 5 - Production/Stable, *requires*: N/A

Run only unmarked tests

pytest-unordered
last release: Jun 03, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Test equality of unordered collections in pytest

pytest-unstable
last release: Sep 27, 2022, *status*: 4 - Beta, *requires*: N/A

Set a test as unstable to return 0 even if it failed

pytest-unused-fixtures
last release: Dec 23, 2025, *status*: 4 - Beta, *requires*: pytest>7.3.2

A pytest plugin to list unused fixtures after a test run.

pytest-unused-port
last release: Oct 22, 2025, *status*: N/A, *requires*: pytest

pytest fixture finding an unused local port

pytest-upload-report
last release: Jun 18, 2021, *status*: 5 - Production/Stable, *requires*: N/A

pytest-upload-report is a plugin for pytest that upload your test report for test results.

pytest-urllib3
last release: Mar 09, 2026, *status*: 3 - Alpha, *requires*: pytest>=7

A pytest plugin to mock urllib3 requests

pytest-utils
last release: Feb 02, 2023, *status*: 4 - Beta, *requires*: pytest (>=7.0.0,<8.0.0)

Some helpers for pytest.

pytest-uuid
last release: Feb 27, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin for mocking uuid.uuid4() calls

pytest-vagrant
last release: Sep 07, 2021, *status*: 5 - Production/Stable, *requires*: pytest

A py.test plugin providing access to vagrant.

pytest-valgrind
last release: May 19, 2021, *status*: N/A, *requires*: N/A

pytest-variables
last release: Feb 01, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

pytest plugin for providing variables to tests/fixtures

pytest-variant
last release: Jun 06, 2022, *status*: N/A, *requires*: N/A

Variant support for Pytest

pytest-vcr
last release: Apr 26, 2019, *status*: 5 - Production/Stable, *requires*: pytest (>=3.6.0)

Plugin for managing VCR.py cassettes

pytest-vcr-delete-on-fail
last release: Feb 16, 2024, *status*: 5 - Production/Stable, *requires*: pytest (>=8.0.0,<9.0.0)

A pytest plugin that automates vcrpy cassettes deletion on test failure.

pytest-vcrpandas
last release: Jan 12, 2019, *status*: 4 - Beta, *requires*: pytest

Test from HTTP interactions to dataframe processed.

pytest-vcs
last release: Sep 22, 2022, *status*: 4 - Beta, *requires*: N/A

pytest-venv
last release: Nov 23, 2023, *status*: 4 - Beta, *requires*: pytest

py.test fixture for creating a virtual environment

pytest-verbose-parametrize
last release: Nov 29, 2024, *status*: 5 - Production/Stable, *requires*: pytest

More descriptive output for parametrized py.test tests

pytest-verify
last release: Oct 25, 2025, *status*: 5 - Production/Stable, *requires*: N/A

A pytest plugin for snapshot verification with optional visual diff viewer.

pytest-ves
last release: Apr 18, 2026, *status*: 3 - Alpha, *requires*: N/A

pytest fixtures and builders for ONAP VES 7.x events (fault, heartbeat, measurement, and more).

pytest-vigil
last release: Apr 07, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=9.0.0

A pytest plugin for enhanced test reliability and monitoring

pytest-vimqf
last release: Feb 08, 2021, *status*: 4 - Beta, *requires*: pytest (>=6.2.2,<7.0.0)

A simple pytest plugin that will shrink pytest output when specified, to fit vim quickfix window.

pytest-virtualenv
last release: Nov 29, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Virtualenv fixture for py.test

pytest-visual
last release: Nov 28, 2024, *status*: 4 - Beta, *requires*: pytest>=7.0.0

pytest-vnc
last release: Nov 06, 2023, *status*: N/A, *requires*: pytest

VNC client for Pytest

pytest-voluptuous
last release: Jun 09, 2020, *status*: N/A, *requires*: pytest

Pytest plugin for asserting data against voluptuous schema.

pytest-vscodedebug
last release: Dec 04, 2020, *status*: 4 - Beta, *requires*: N/A

A pytest plugin to easily enable debugging tests within Visual Studio Code

pytest-vscode-pycharm-cls
last release: Feb 01, 2023, *status*: N/A, *requires*: pytest

A PyTest helper to enable start remote debugger on test start or failure or when pytest.set_trace is used.

pytest-vtestify
last release: Feb 04, 2025, *status*: N/A, *requires*: pytest

A pytest plugin for visual assertion using SSIM and image comparison.

pytest-vts
last release: Jun 05, 2019, *status*: N/A, *requires*: pytest (>=2.3)

pytest plugin for automatic recording of http stubbed tests

pytest-vulture
last release: Nov 25, 2024, *status*: N/A, *requires*: pytest>=7.0.0

A pytest plugin to checks dead code with vulture

pytest-vw
last release: Oct 07, 2015, *status*: 4 - Beta, *requires*: N/A

pytest-vw makes your failing test cases succeed under CI tools scrutiny

pytest-vyper
last release: May 28, 2020, *status*: 2 - Pre-Alpha, *requires*: N/A

Plugin for the vyper smart contract language.

pytest-wa-e2e-plugin
last release: Feb 18, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.5.0)

Pytest plugin for testing whatsapp bots with end to end tests

pytest-wake
last release: Nov 19, 2024, *status*: N/A, *requires*: pytest

pytest-warmup
last release: Apr 13, 2026, *status*: 3 - Alpha, *requires*: pytest<10,>=8.4

Batch preparation and distribution of expensive test resources for pytest.

pytest-watch
last release: May 20, 2018, *status*: N/A, *requires*: N/A

Local continuous test runner with pytest and watchdog.

pytest-watcher
last release: Jan 10, 2026, *status*: 4 - Beta, *requires*: N/A

Automatically rerun your tests on file modifications

pytest-watch-plugin
last release: Sep 12, 2024, *status*: N/A, *requires*: N/A

Placeholder for internal package

pytest_wdb
last release: Jul 04, 2016, *status*: N/A, *requires*: N/A

Trace pytest tests with wdb to halt on error with –wdb.

pytest-wdl
last release: Nov 17, 2020, *status*: 5 - Production/Stable, *requires*: N/A

Pytest plugin for testing WDL workflows.

pytest-web3-data
last release: Oct 04, 2023, *status*: 4 - Beta, *requires*: pytest

A pytest plugin to fetch test data from IPFS HTTP gateways during pytest execution.

pytest-webdriver
last release: Oct 17, 2024, *status*: 5 - Production/Stable, *requires*: pytest

Selenium webdriver fixture for py.test

pytest-webstage
last release: Sep 20, 2024, *status*: N/A, *requires*: pytest<9.0,>=7.0

Test web apps with pytest

pytest-webtestpilot
last release: Dec 28, 2025, *status*: N/A, *requires*: pytest>=9.0.2

Pytest plugin for running WebTestPilot JSON tests

pytest-wetest
last release: Nov 10, 2018, *status*: 4 - Beta, *requires*: N/A

Welian API Automation test framework pytest plugin

pytest-when
last release: Sep 25, 2025, *status*: N/A, *requires*: pytest>=7.3.1

Utility which makes mocking more readable and controllable

pytest-whirlwind
last release: Jun 12, 2020, *status*: N/A, *requires*: N/A

Testing Tornado.

pytest-wholenodeid
last release: Aug 26, 2015, *status*: 4 - Beta, *requires*: pytest (>=2.0)

pytest addon for displaying the whole node id for failures

pytest-win32consoletitle
last release: Aug 08, 2021, *status*: N/A, *requires*: N/A

Pytest progress in console title (Win32 only)

pytest-winnotify
last release: Apr 22, 2016, *status*: N/A, *requires*: N/A

Windows tray notifications for py.test results.

pytest-wirefracture
last release: Dec 31, 2025, *status*: N/A, *requires*: N/A

Pytest fixtures for wirefracture

pytest-wiremock
last release: Mar 27, 2022, *status*: N/A, *requires*: pytest (>=7.1.1,<8.0.0)

A pytest plugin for programmatically using wiremock in integration tests

pytest-wiretap
last release: Mar 18, 2025, *status*: N/A, *requires*: pytest

`pytest` plugin for recording call stacks

pytest-with-docker
last release: Nov 09, 2021, *status*: N/A, *requires*: pytest

pytest with docker helpers.

pytest-workaround-12888
last release: Jan 15, 2025, *status*: N/A, *requires*: N/A

forces an import of readline early in the process to work around pytest bug #12888

pytest-workflow
last release: Mar 18, 2024, *status*: 5 - Production/Stable, *requires*: pytest >=7.0.0

A pytest plugin for configuring workflow/pipeline tests using YAML files

pytest-xdist
last release: Jul 01, 2025, *status*: 5 - Production/Stable, *requires*: pytest>=7.0.0

pytest xdist plugin for distributed testing, most importantly across multiple CPUs

pytest-xdist-debug-for-graingert
last release: Jul 24, 2019, *status*: 5 - Production/Stable, *requires*: pytest (>=4.4.0)

pytest xdist plugin for distributed testing and loop-on-failing modes

pytest-xdist-forked
last release: Feb 10, 2020, *status*: 5 - Production/Stable, *requires*: pytest (>=4.4.0)

forked from pytest-xdist

pytest-xdist-gnumake
last release: Jun 22, 2025, *status*: N/A, *requires*: pytest

A small example package

pytest-xdist-load-testing
last release: Nov 22, 2025, *status*: 4 - Beta, *requires*: pytest>=8.4.2

xdist scheduler to repeately run tests

pytest-xdist-rate-limit
last release: Dec 31, 2025, *status*: 4 - Beta, *requires*: pytest>=8.4.2

Shared state management and rate limiting for pytest-xdist workers

pytest-xdist-tracker
last release: Nov 18, 2021, *status*: 3 - Alpha, *requires*: pytest (>=3.5.1)

pytest plugin helps to reproduce failures for particular xdist node

pytest-xdist-worker-stats
last release: Feb 16, 2026, *status*: 4 - Beta, *requires*: pytest>=7.0.0

A pytest plugin to list worker statistics after a xdist run.

pytest-xdocker
last release: Mar 27, 2026, *status*: N/A, *requires*: pytest>=9.0.0

Pytest fixture to run docker across test runs.

pytest-xfaillist
last release: Sep 17, 2021, *status*: N/A, *requires*: pytest (>=6.2.2,<7.0.0)

Maintain a xfaillist in an additional file to avoid merge-conflicts.

pytest-xfiles
last release: Feb 27, 2018, *status*: N/A, *requires*: N/A

Pytest fixtures providing data read from function, module or package related (x)files.

pytest-xflaky
last release: Oct 14, 2024, *status*: 4 - Beta, *requires*: pytest>=8.2.1

A simple plugin to use with pytest

pytest-xhtml
last release: Feb 26, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=7

pytest plugin for generating HTML reports

pytest-xiuyu
last release: Jul 25, 2023, *status*: 5 - Production/Stable, *requires*: N/A

This is a pytest plugin

pytest-xlog
last release: May 31, 2020, *status*: 4 - Beta, *requires*: N/A

Extended logging for test and decorators

pytest-xlsx
last release: Aug 07, 2024, *status*: N/A, *requires*: pytest~=8.2.2

pytest plugin for generating test cases by xlsx(excel)

pytest-xml
last release: Nov 14, 2024, *status*: 4 - Beta, *requires*: pytest>=8.0.0

Create simple XML results for parsing

pytest-xpara
last release: Aug 07, 2024, *status*: 3 - Alpha, *requires*: pytest

An extended parametrizing plugin of pytest.

pytest-xprocess
last release: May 19, 2024, *status*: 4 - Beta, *requires*: pytest>=2.8

A pytest plugin for managing processes across test runs.

pytest-xray
last release: May 30, 2019, *status*: 3 - Alpha, *requires*: N/A

pytest-xrayjira
last release: Mar 17, 2020, *status*: 3 - Alpha, *requires*: pytest (==4.3.1)

pytest-xray-reporter
last release: May 21, 2025, *status*: 4 - Beta, *requires*: pytest>=7.0.0

Pytest plugin for generating Xray JSON reports

pytest-xray-server
last release: May 03, 2022, *status*: 3 - Alpha, *requires*: pytest (>=5.3.1)

pytest-xstress
last release: Jun 01, 2024, *status*: N/A, *requires*: pytest<9.0.0,>=8.0.0

pytest-xtime
last release: Jun 05, 2025, *status*: 4 - Beta, *requires*: pytest

pytest plugin for recording execution time

pytest-xvfb
last release: Mar 12, 2025, *status*: 4 - Beta, *requires*: pytest>=2.8.1

A pytest plugin to run Xvfb (or Xephyr/Xvnc) for tests.

pytest-xvirt
last release: Dec 15, 2024, *status*: 4 - Beta, *requires*: pytest>=7.2.2

A pytest plugin to virtualize test. For example to transparently running them on a remote box.

pytest-yaml
last release: Oct 05, 2018, *status*: N/A, *requires*: pytest

This plugin is used to load yaml output to your test using pytest framework.

pytest-yaml-fei
last release: Jul 27, 2025, *status*: N/A, *requires*: pytest

a pytest yaml allure package

pytest-yaml-sanmu
last release: Sep 16, 2025, *status*: N/A, *requires*: pytest>=8.2.2

Pytest plugin for generating test cases with YAML. In test cases, you can use markers, fixtures, variables, and even call Python functions.

pytest-yamltree
last release: Mar 02, 2020, *status*: 4 - Beta, *requires*: pytest (>=3.1.1)

Create or check file/directory trees described by YAML

pytest-yamlwsgi
last release: May 11, 2010, *status*: N/A, *requires*: N/A

Run tests against wsgi apps defined in yaml

pytest-yaml-yoyo
last release: Jun 19, 2023, *status*: N/A, *requires*: pytest (>=7.2.0)

http/https API run by yaml

pytest-yapf
last release: Jul 06, 2017, *status*: 4 - Beta, *requires*: pytest (>=3.1.1)

Run yapf

pytest-yapf3
last release: Mar 29, 2023, *status*: 5 - Production/Stable, *requires*: pytest (>=7)

Validate your Python file format with yapf

pytest-yield
last release: Jan 23, 2019, *status*: N/A, *requires*: N/A

PyTest plugin to run tests concurrently, each `yield` switch context to other one

pytest-yls
last release: Apr 09, 2025, *status*: N/A, *requires*: pytest<9.0.0,>=8.3.3

Pytest plugin to test the YLS as a whole.

pytest-youqu-playwright
last release: Jun 12, 2024, *status*: N/A, *requires*: pytest

pytest-youqu-playwright

pytest-yuk
last release: Mar 26, 2021, *status*: N/A, *requires*: pytest>=5.0.0

Display tests you are uneasy with, using 😲/😖 for pass/fail of tests marked with yuk.

pytest-zafira
last release: Sep 18, 2019, *status*: 5 - Production/Stable, *requires*: pytest (==4.1.1)

A Zafira plugin for pytest

pytest-zap
last release: May 12, 2014, *status*: 4 - Beta, *requires*: N/A

OWASP ZAP plugin for py.test.

pytest-zcc
last release: Jun 02, 2024, *status*: N/A, *requires*: N/A

eee

pytest-zebrunner
last release: Jul 04, 2024, *status*: 5 - Production/Stable, *requires*: pytest>=4.5.0

Pytest connector for Zebrunner reporting

pytest-zeebe
last release: Feb 01, 2024, *status*: N/A, *requires*: pytest (>=7.4.2,<8.0.0)

Pytest fixtures for testing Camunda 8 processes using a Zeebe test engine.

pytest-zephyr-scale-integration
last release: Jun 26, 2025, *status*: N/A, *requires*: pytest

A library for integrating Jira Zephyr Scale (AdaptavistTM4J) with pytest

pytest-zephyr-telegram
last release: Sep 30, 2024, *status*: N/A, *requires*: pytest==8.3.2

Плагин для отправки данных автотестов в Телеграм и Зефир

pytest-zest
last release: Nov 17, 2022, *status*: N/A, *requires*: N/A

Zesty additions to pytest.

pytest-zigzag
last release: Feb 27, 2019, *status*: 4 - Beta, *requires*: pytest (~=3.6)

Extend py.test for RPC OpenStack testing.

pytest-zulip
last release: May 07, 2022, *status*: 5 - Production/Stable, *requires*: pytest

Pytest report plugin for Zulip

pytest-zy
last release: Mar 24, 2024, *status*: N/A, *requires*: pytest~=7.2.0

接口自动化测试框架

tursu
last release: Mar 02, 2026, *status*: 5 - Production/Stable, *requires*: pytest>=8.3.5

🎬 A pytest plugin that transpiles Gherkin feature files to Python using AST, enforcing typing for ease of use and debugging.

3.3 Configuration

3.3.1 Command line options and configuration file settings

You can get help on command line and configuration options 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 in the root directory of your repository.

A quick example of the configuration files supported by pytest:

pytest.toml

Added in version 9.0.

`pytest.toml` files take precedence over other files, even when empty.

Alternatively, the hidden version `.pytest.toml` can be used.

```
# pytest.toml or .pytest.toml
[pytest]
minversion = "9.0"
addopts = ["-ra", "-q"]
testpaths = [
    "tests",
    "integration",
]
```

pytest.ini

`pytest.ini` files take precedence over other files (except `pytest.toml` and `.pytest.toml`), even when empty.

Alternatively, the hidden version `.pytest.ini` can be used.

```
# pytest.ini or .pytest.ini
[pytest]
minversion = 6.0
addopts = -ra -q
testpaths =
    tests
    integration
```

pyproject.toml

Added in version 6.0.

Changed in version 9.0.

`pyproject.toml` files are supported for configuration.

Use `[tool.pytest]` to leverage native TOML types (supported since pytest 9.0):

```toml
# pyproject.toml
[tool.pytest]
minversion = "9.0"
addopts = ["-ra", "-q"]
testpaths = [
    "tests",
    "integration",
]
```

Use `[tool.pytest.ini_options]` for INI-style configuration (supported since pytest 6.0):

```toml
# pyproject.toml
[tool.pytest.ini_options]
minversion = "6.0"
addopts = "-ra -q"
testpaths = [
    "tests",
    "integration",
]
```

For projects that still run pytest versions older than 6.0, keep `minversion` in `pytest.ini` or `setup.cfg` too. Those versions do not read `pyproject.toml`.

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.

```ini
# 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` (now deprecated) and setuptools, and can also be used to hold pytest configuration if they have a `[tool:pytest]` section.

```ini
# 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 of 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 a configuration file because the `rootdir` is used to *find* the configuration file 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.toml`, `.pytest.toml`, `pytest.ini`, `.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.toml`, `.pytest.toml`, `pytest.ini`, `.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 tests below the current working directory and also starts determining the `rootdir` from there.

Files will only be matched for configuration if:

- `pytest.toml`: will always match and take highest precedence, even if empty.
- `pytest.ini`: will always match and take precedence (after `pytest.toml` and `.pytest.toml`), even if empty.
- `pyproject.toml`: contains a `[tool.pytest]` or `[tool.pytest.ini_options]` table.
- `tox.ini`: contains a `[pytest]` section.
- `setup.cfg`: contains a `[tool:pytest]` section.

Finally, a `pyproject.toml` file will be considered the `configfile` if no other match was found, in this case even if it does not contain a `[tool.pytest]` table (since version `9.0`) or a `[tool.pytest.ini_options]` table (since version `8.1`).

The files are considered in the order above. Options from multiple `configfiles` candidates are never merged - the first match wins.

The configuration file also determines the value of the `rootpath`.

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. It is used as a reference directory for constructing test addresses ("nodeids") and can be used also by plugins for storing per-testrun information.
- *`config.inipath`*: the determined `configfile`, may be `None` (it is named `inipath` for historical reasons).

Added 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.

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 path/pytest.toml
path/pytest.toml
path/pytest.ini
path/pyproject.toml  # must contain a [tool.pytest] table to match
path/tox.ini         # must contain [pytest] section to match
path/setup.cfg       # must contain [tool:pytest] section to match
pytest.toml
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 directory of test.log for rootdir determination (see also #1435). A dot `.` for referencing 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.

3.4.1 Constants

pytest.__version__

The current pytest version, as a string:

```
>>> import pytest
>>> pytest.__version__
'9.0.2'
```

pytest.HIDDEN_PARAM

Added in version 8.4.

Can be passed to `ids` of *Metafunc.parametrize* or to `id` of *pytest.param()* to hide a parameter set from the test name. Can only be used at most 1 time, as test names need to be unique.

pytest.version_tuple

Added 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')
```

3.4.2 Functions

pytest.approx

approx(*expected*, *rel=None*, *abs=None*, *nan_ok=False*)

Assert that two numbers (or two ordered sequences 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 ordered sequences of numbers:

```
>>> (0.1 + 0.2, 0.2 + 0.4) == approx((0.3, 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
```

Only ordered sequences are supported, because `approx` needs to infer the relative position of the sequences without ambiguity. This means `sets` and other unordered sequences are not supported.

Finally, dictionary *values* can also be compared:

```
>>> {'a': 0.1 + 0.2, 'b': 0.2 + 0.4} == approx({'a': 0.3, 'b': 0.6})
True
```

The comparison will be true if both mappings have the same keys and their respective values match the expected tolerances.

Tolerances

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
```

Non-numeric types

You can also use `approx` to compare non-numeric types, or dicts and sequences containing non-numeric 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 that 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 if you need support for comparisons, NaNs, or ULP-based tolerances.

To match strings using regex, you can use Matches from the re_assert package.

Note

Unlike built-in equality, this function considers booleans unequal to numeric zero or one. For example:

```
>>> 1 == approx(True)
False
```

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 non-numeric type.

Changed in version 6.1.0: `approx` falls back to strict equality for non-numeric types instead of raising `TypeError`.

pytest.fail

Tutorial: *How to use skip and xfail to deal with tests that cannot succeed*

fail(*reason*[, *pytrace=True*])

Explicitly fail an executing test with the given message.

Parameters

- **reason** – The message to show the user as reason for the failure.
- **pytrace** – If False, msg represents the full failure information and no python traceback will be reported.

Raises

pytest.fail.Exception – The exception that is raised.

class pytest.fail.**Exception**

The exception raised by *pytest.fail()*.

pytest.skip

skip(*reason*[, *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

- **reason** – The message to show the user as reason for the skip.
- **allow_module_level** – Allows this function to be called at module level. Raising the skip exception at module level will stop the execution of the module and prevent the collection of all tests in the module, even those defined before the `skip` call.

  Defaults to False.

Raises

pytest.skip.Exception – The exception that is raised.

> **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.

class pytest.skip.**Exception**

The exception raised by *pytest.skip()*.

pytest.importorskip

importorskip(*modname*, *minversion=None*, *reason=None*, ** (Keyword-only parameters separator (PEP 3102))*, *exc_type=None*)

Import and return the requested module `modname`, or skip the current test if the module cannot be imported.

Parameters

- **modname** – The name of the module to import.

- **minversion** – If given, the imported module's `__version__` attribute must be at least this minimal version, otherwise the test is still skipped.
- **reason** – If given, this reason is shown as the message when the module cannot be imported.
- **exc_type** – The exception that should be captured in order to skip modules. Must be `ImportError` or a subclass.

  Defaults to `ModuleNotFoundError` when not given, which means the module must be missing for the test to be skipped. Pass `exc_type=ImportError` to also skip modules that raise `ImportError` during import.

  See *pytest.importorskip default behavior regarding ImportError* for details.

Returns
The imported module. This should be assigned to its canonical name.

Raises
pytest.skip.Exception – If the module cannot be imported.

Example:

```
docutils = pytest.importorskip("docutils")
```

Added in version 8.2: The `exc_type` parameter.

Changed in version 9.1: The default for `exc_type` is now `ModuleNotFoundError`.

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).

No other code is executed after using `xfail()` (it is implemented internally by raising an exception).

Parameters
reason – The message to show the user as reason for the xfail.

> **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.

Raises
pytest.xfail.Exception – The exception that is raised.

class pytest.xfail.**Exception**

The exception raised by *pytest.xfail()*.

pytest.exit

exit(*reason*[, *returncode=None*])

Exit testing process.

Parameters

- **reason** – The message to show as the reason for exiting pytest. reason has a default value only because `msg` is deprecated.
- **returncode** – Return code to be used when exiting pytest. None means the same as `0` (no error), same as `sys.exit()`.

Raises

pytest.exit.Exception – The exception that is raised.

class pytest.exit.**Exception**

The exception raised by *pytest.exit()*.

pytest.main

Tutorial: *Calling pytest from Python code*

main(*args=None*, *plugins=None*)

Perform an in-process test run.

Parameters

- **args** – List of command line arguments. If `None` or not given, defaults to reading arguments directly from the process command line (`sys.argv`).
- **plugins** – List of plugin objects to be auto-registered during initialization.

Returns

An exit code.

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** – Variable args of the values of the parameter set, in order.
- **marks** – A single mark or a list of marks to be applied to this parameter set.

  pytest.mark.usefixtures cannot be added via this parameter.
- **id** (*str | Literal[pytest.HIDDEN_PARAM] | None*) – The id to attribute to this parameter set.

  Added in version 8.4: *pytest.HIDDEN_PARAM* means to hide the parameter set from the test name. Can only be used at most 1 time, as test names need to be unique.

pytest.raises

Tutorial: *Assertions about approximate equality*

with raises(*expected_exception: type[E] | tuple[type[E], ...]*, *, *match: str | Pattern[str] | None = ...*, *check: Callable[[E], bool] = ...*) → *RaisesExc*[E] **as excinfo**

with raises(*, *match: str | Pattern[str]*, *check: Callable[[BaseException], bool] = ...*) → *RaisesExc*[BaseException] **as excinfo**

with raises(*, *check: Callable[[BaseException], bool]*) → *RaisesExc*[BaseException] **as excinfo**

with raises(*expected_exception: type[E] | tuple[type[E], ...]*, *func: Callable[P, object]*, **args: P.args*, ***kwargs: P.kwargs*) → *ExceptionInfo*[E] **as excinfo**

Assert that a code block/function call raises an exception type, or one of its subclasses.

Parameters

- **expected_exception** – The expected exception type, or a tuple if one of multiple possible exception types are expected. Note that subclasses of the passed exceptions will also match.

  This is not a required parameter, you may opt to only use `match` and/or `check` for verifying the raised exception.

- **match** (*str | re.Pattern[str] | None*) – If specified, a string containing a regular expression, or a regular expression object, that is tested against the string representation of the exception and its **PEP 678** `__notes__` 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")`.)

- **check** (*Callable[[BaseException], bool]*) – Added in version 8.4.

  If specified, a callable that will be called with the exception as a parameter after checking the type and the match regex if specified. If it returns `True` it will be considered a match, if not it will be considered a failed match.

Use `pytest.raises` as a context manager, which will capture the exception of the given type, or any of its subclasses:

```
>>> 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 `match` argument searches the formatted exception string, which includes any PEP-678 `__notes__`:

```
>>> with pytest.raises(ValueError, match=r"had a note added"):
...     e = ValueError("value must be 42")
...     e.add_note("had a note added")
...     raise e
```

The `check` argument, if provided, must return True when passed the raised exception for the match to be successful, otherwise an `AssertionError` is raised.

```
>>> import errno
>>> with pytest.raises(OSError, check=lambda e: e.errno == errno.EACCES):
...     raise OSError(errno.EACCES, "no permission to view")
```

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"
```

Warning

Given that `pytest.raises` matches subclasses, be wary of using it to match `Exception` like this:

```
# Careful, this will catch ANY exception raised.
with pytest.raises(Exception):
    some_function()
```

Because `Exception` is the base class of almost all exceptions, it is easy for this to hide real bugs, where the user wrote this expecting a specific exception, but some other exception is being raised due to a bug introduced during a refactoring.

Avoid using `pytest.raises` to catch `Exception` unless certain that you really want to catch **any** exception raised.

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
```

Expecting exception groups

When expecting exceptions wrapped in `BaseExceptionGroup` or `ExceptionGroup`, you should instead use *`pytest.RaisesGroup`*.

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.

See also

Assertions about approximate equality for more examples and detailed discussion.

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: str | Pattern[str] | None = ...*) → *WarningsRecorder*

with deprecated_call(*func: Callable[P, T]*, **args: P.args*, ***kwargs: P.kwargs*) → T

Assert that code produces a `DeprecationWarning` or `PendingDeprecationWarning` or `FutureWarning`.

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
>>> with pytest.deprecated_call(match="^use v3 of this api$") as warning_messages:
...    assert api_call_v2() == 200
```

You may use the keyword argument `match` to assert that the warning matches a text or regex.

The return value is a list of `warnings.WarningMessage` objects, one for each warning emitted (regardless of whether it is an `expected_warning` or not).

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.

Parameters

names – The module names to register.

pytest.warns

Tutorial: *Asserting warnings with the warns function*

with warns(*expected_warning: type[Warning] | tuple[type[Warning], ...] = <class 'Warning'>, *, match: str | ~re.Pattern[str] | None = ...*) → WarningsChecker

with warns(*expected_warning:* type*[Warning] |* *tuple[*type*[Warning], ...], func: Callable[P, T], *args: P.args, **kwargs: P.kwargs*) → T

Assert that code raises a particular class of warning.

Specifically, the parameter `expected_warning` can be a warning class or tuple of warning classes, and the code inside the `with` block must issue at least one warning of that class or classes.

This helper produces a list of `warnings.WarningMessage` objects, one for each warning emitted (regardless of whether it is an `expected_warning` or not). Since pytest 8.0, unmatched warnings are also re-emitted when the context closes.

This function should be used as a context manager:

```
>>> import pytest
>>> with pytest.warns(RuntimeWarning):
...     warnings.warn("my warning", RuntimeWarning)
```

The `match` keyword argument can be used 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):  # catch re-emitted warning
...     with pytest.warns(UserWarning, match=r'must be \d+$'):
...         warnings.warn("this is not here", UserWarning)
Traceback (most recent call last):
  ...
Failed: Regex pattern did not match any of the 1 warnings emitted.
 Regex: ...
 Emitted warnings: ...UserWarning...
```

Using with `pytest.mark.parametrize`

When using *pytest.mark.parametrize* it is possible to parametrize tests such that some runs raise a warning and others do not.

This could be achieved in the same way as with exceptions, see *Parametrizing conditional raising* for an example.

pytest.freeze_includes

Tutorial: *Freezing pytest*

freeze_includes()

Return a list of module names used by pytest that should be included by cx_freeze.

3.4.3 Marks

Marks can be used to apply metadata 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(r"ignore:.*usage will be deprecated.
↪*:DeprecationWarning")
def test_foo(): ...
```

pytest.mark.parametrize

Tutorial: *How to parametrize fixtures and test functions*

This mark has the same signature as *pytest.Metafunc.parametrize()*; see there.

pytest.mark.skip

Tutorial: *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

Tutorial: *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=False*, *, *reason=None*, *raises=None*, *run=True*, *strict=strict_xfail*)

Parameters

- **condition** (*Union[bool, 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 class (or tuple of classes) expected to be raised by the test function; other exceptions will fail the test. Note that subclasses of the classes passed will also result in a match (similar to how the `except` statement works).
- **run** (*bool*) – Whether 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 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).

  Defaults to *strict_xfail*, which is `False` by default.

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_with_node* 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(...)`.

3.4.4 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: Callable[[...], object]*, *, *scope: Literal['session', 'package', 'module', 'class', 'function'] | Callable[[str, Config], Literal['session', 'package', 'module', 'class', 'function']] = 'function'*, *params: Iterable[object] | None = None*, *autouse: bool = False*, *ids: Sequence[object | None] | Callable[[Any], object | None] | None = None*, *name: str | None = None*) → FixtureFunctionDefinition

@fixture(*fixture_function: None = None*, *, *scope: Literal['session', 'package', 'module', 'class', 'function'] | Callable[[str, Config], Literal['session', 'package', 'module', 'class', 'function']] = 'function'*, *params: Iterable[object] | None = None*, *autouse: bool = False*, *ids: Sequence[object | None] | Callable[[Any], object | None] | None = None*, *name: str | None = None*) → 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** – Sequence of 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>')`.

capfd

Tutorial: *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"
```

capfdbinary

Tutorial: *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"
```

caplog

Tutorial: *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.

final class LogCaptureFixture

Provides access and control of log capturing.

property handler: LogCaptureHandler

Get the logging handler used by the fixture.

get_records(*when*)

Get the logging records for one of the possible test phases.

Parameters
when (*Literal['setup', 'call', 'teardown']*) – 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[*LogRecord*]

Added in version 3.4.

property text: str

The formatted log text.

property records: list[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.

Added in version 3.7.

clear()

Reset the list of log records and the captured log text.

set_level(*level*, *logger=None*)

Set the threshold level of a logger for the duration of a test.

Logging messages which are less severe than this level will not be captured.

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.

Will enable the requested logging level if it was disabled via `logging.disable()`.

Parameters

- **level** (*int | str*) – The level.
- **logger** (*str | None*) – The logger to update. If not given, the root logger.

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.

Will enable the requested logging level if it was disabled via `logging.disable()`.

Parameters

- **level** (*int | str*) – The level.
- **logger** (*str | None*) – The logger to update. If not given, the root logger.

filtering(*filter_*)

Context manager that temporarily adds the given filter to the caplog's *handler()* for the 'with' statement block, and removes that filter at the end of the block.

Parameters
filter – A custom `logging.Filter` object.

Added in version 7.5.

capsys

Tutorial: *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"
```

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
CaptureResult

disabled()

Temporarily disable capturing while inside the `with` block.

capteesys

Tutorial: *How to capture stdout/stderr output*

capteesys()

Enable simultaneous text capturing and pass-through of writes to `sys.stdout` and `sys.stderr` as defined by `--capture=`.

The captured output is made available via `capteesys.readouterr()` method calls, which return a `(out, err)` namedtuple. `out` and `err` will be `text` objects.

The output is also passed-through, allowing it to be "live-printed", reported, or both as defined by `--capture=`.

Returns an instance of *CaptureFixture[str]*.

Example:

```
def test_output(capteesys):
    print("hello")
    captured = capteesys.readouterr()
    assert captured.out == "hello\n"
```

capsysbinary

Tutorial: *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"
```

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

Instance of the `cache` fixture.

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.

Added in version 7.0.

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.

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.

doctest_namespace

Tutorial: *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*.

monkeypatch

Tutorial: *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)*
- *monkeypatch.context()*

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 does not have the specified target.

To undo modifications done by the fixture in a contained scope, use *context()*.

Returns a *MonkeyPatch* instance.

final class MonkeyPatch

Helper to conveniently monkeypatch attributes/items/environment variables/syspath.

Returned by the *monkeypatch* fixture.

Changed in version 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.

classmethod 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).

setattr(*target: str*, *name: object*, *value: NotSetType = NotSetType.token*, *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 example:

```
import os

monkeypatch.setattr(os, "getcwd", lambda: "/")
```

The code above replaces the `os.getcwd()` function by a `lambda` which always returns `"/"`.

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:

```
monkeypatch.setattr("os.getcwd", lambda: "/")
```

Raises `AttributeError` if the attribute does not exist, unless `raising` is set to False.

Where to patch

`monkeypatch.setattr` works by (temporarily) changing the object that a name points to with another one. There can be many names pointing to any individual object, so for patching to work you must ensure that you patch the name used by the system under test.

See the section Where to patch in the `unittest.mock` docs for a complete explanation, which is meant for `unittest.mock.patch()` but applies to `monkeypatch.setattr` as well.

delattr(*target*, *name=NotSetType.token*, *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 it the attribute does not exist, unless `raising` is set to False.

setitem(*dic*, *name*, *value*)

Set dictionary entry `name` to value.

delitem(*dic*, *name*, *raising=True*)

Delete `name` from dict.

Raises `KeyError` if it doesn't exist, unless `raising` is set to False.

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.

delenv(*name*, *raising=True*)

Delete `name` from the environment.

Raises `KeyError` if it does not exist, unless `raising` is set to False.

syspath_prepend(*path*)

Prepend `path` to `sys.path` list of import locations.

chdir(*path*)

Change the current working directory to the specified path.

Parameters

path (*str | PathLike[str]*) – The path to change into.

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

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.

Prefer to use *context()* instead.

pytestconfig

pytestconfig()

Session-scoped fixture that returns the session's *pytest.Config* object.

Example:

```
def test_foo(pytestconfig):
    if pytestconfig.get_verbosity() > 0:
        ...
```

pytester

Added 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.

exception TimeoutExpired

plugins: *list*[str | object]

A list of plugins to use with *parseconfig()* and *runpytest()*. Initially this is an empty list but plugins can be added to the list.

When running in subprocess mode, specify plugins by name (str) - adding plugin objects directly is not supported.

property path: Path

Temporary directory path used to create files/run tests from, etc.

make_hook_recorder(*pluginmanager*)

Create a new *HookRecorder* for a *PytestPluginManager*.

chdir()

Cd into the temporary directory.

This is done automatically upon instantiation.

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.

Returns
The first created file.

Return type
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.

Parameters
source (*str*) – The contents.

Returns
The conftest.py file.

Return type
Path

makeini(*source*)

Write a tox.ini file.

Parameters
source (*str*) – The contents.

Returns
The tox.ini file.

Return type
Path

maketoml(*source*)

Write a pytest.toml file.

Parameters
source (*str*) – The contents.

Returns
The pytest.toml file.

Return type
Path

Added in version 9.0.

getinicfg(*source*)

Return the pytest section from the tox.ini config file.

makepyprojecttoml(*source*)

Write a pyproject.toml file.

Parameters
source (*str*) – The contents.

Returns
The pyproject.ini file.

Return type
Path

Added in version 6.0.

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.
```

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.
```

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 (*str | PathLike[str] | None*) – The path.

mkdir(*name*)

Create a new (sub)directory.

Parameters
name (*str | PathLike[str]*) – The name of the directory, relative to the pytester path.

Returns
The created directory.

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.

copy_example(*name=None*)

Copy file from project's directory into the testdir.

Parameters
name (*str | None*) – The name of the file to copy.

Returns
Path to the copied directory (inside `self.path`).

Return type
pathlib.Path

getnode(*config*, *arg*)

Get the collection node of a file.

Parameters

- **config** (*Config*) – A pytest config. See *parseconfig()* and *parseconfigure()* for creating it.
- **arg** (*str | PathLike[str]*) – Path to the file.

Returns
The node.

Return type
Collector | Item

getpathnode(*path*)

Return the collection node of a file.

This is like *getnode()* but uses *parseconfigure()* to create the (configured) pytest Config instance.

Parameters
path (*str | PathLike[str]*) – Path to the file.

Returns
The node.

Return type
Collector | Item

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[Item | Collector]*) – The collection nodes.

Returns
The collected items.

Return type
list[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`.

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.

inline_genitems(**args*)

Run `pytest.main(['--collect-only'])` 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.

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** (*str | 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.

runpytest_inprocess(**args*, ***kwargs*)

Return result of running pytest in-process, providing a similar interface to what self.runpytest() provides.

runpytest(**args*, ***kwargs*)

Run pytest inline or in a subprocess, depending on the command line option "–runpytest" and return a *RunResult*.

parseconfig(**args*)

Return a new pytest *pytest.Config* instance from given commandline args.

This invokes the pytest bootstrapping code in _pytest.config to create a new *pytest.PytestPluginManager* 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 plugin manager.

parseconfigure(**args*)

Return a new pytest configured Config instance.

Returns a new *pytest.Config* instance like *parseconfig()*, but also calls the *pytest_configure* hook.

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** (*str | PathLike[str]*) – The module source.
- **funcname** (*str*) – The name of the test function for which to return a test item.

Returns
The test item.

Return type
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.

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** (*str | 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.

Searches a module collection node for a collection node matching the given name.

Parameters

- **modcol** (`Collector`) – A module collection node; see *getmodulecol()*.
- **name** (*str*) – The name of the node to return.

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.

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** (*str | PathLike[str]*) – The sequence of arguments to pass to `subprocess.Popen`, with path-like objects being converted to `str` automatically.
- **timeout** (*float | None*) – The period in seconds after which to timeout and raise *Pytester.TimeoutExpired*.
- **stdin** (*_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`.

Returns
The result.

Return type
RunResult

runpython(*script*)

Run a python script using sys.executable as interpreter.

runpython_c(*command*)

Run `python -c "command"`.

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** (*str | PathLike[str]*) – The sequence of arguments to pass to the pytest subprocess.
- **timeout** (*float | None*) – The period in seconds after which to timeout and raise *Pytester.TimeoutExpired*.

Returns
The result.

Return type
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.

spawn(*cmd*, *expect_timeout=10.0*)

Run a command using pexpect.

The pexpect child is returned.

final class RunResult

The result of running a command from *Pytester*.

ret: int | *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}`.

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}`.

assert_outcomes(*passed=0*, *skipped=0*, *failed=0*, *errors=0*, *xpassed=0*, *xfailed=0*, *warnings=None*, *deselected=None*)

Assert that the specified outcomes appear with the respective numbers (0 means it didn't occur) in the text output from a test run.

`warnings` and `deselected` are only checked if not 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()`.

__str__()

Return the entire original text.

Added in version 6.2: You can use *`str()`* in older versions.

fnmatch_lines_random(*lines2*)

Check lines exist in the output in any order (using `fnmatch.fnmatch()`).

re_match_lines_random(*lines2*)

Check lines exist in the output in any order (using `re.match()`).

get_lines_after(*fnline*)

Return all lines following the given line in the text.

The given line can contain glob wildcards.

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?

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?

no_fnmatch_line(*pat*)

Ensure captured lines do not match the given pattern, using `fnmatch.fnmatch`.

Parameters
pat (*str*) – The pattern to match lines.

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.

str()

Return the entire original text.

final class HookRecorder

Record all hooks called in a plugin manager.

Hook recorders are created by *Pytester*.

This wraps all the hook calls in the plugin manager, recording each call before propagating the normal calls.

getcalls(*names*)

Get all recorded calls to hooks with the given names (or name).

matchreport(*inamepart=''*, *names=('pytest_runtest_logreport', 'pytest_collectreport')*, *when=None*)

Return a testreport whose dotted import path matches.

final class RecordedHookCall

A recorded call to a hook.

The arguments to the hook call are set as attributes. For example:

```
calls = hook_recorder.getcalls("pytest_runtest_setup")
# Suppose pytest_runtest_setup was called once with `item=an_item`.
assert calls[0].item is an_item
```

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)
```

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")
```

Parameters

- **name** – The property name.
- **value** – The property value. Will be converted to a string.

Warning

Currently this fixture **does not work** with the pytest-xdist plugin. See #7767 for details.

recwarn

Tutorial: *Recording warnings*

recwarn()

Return a `WarningsRecorder` instance that records all warnings emitted by test functions.

See *How to capture warnings* for information on warning categories.

class WarningsRecorder

A context manager to record raised warnings.

Each recorded warning is an instance of `warnings.WarningMessage`.

Adapted from `warnings.catch_warnings`.

Note

`DeprecationWarning` and `PendingDeprecationWarning` are treated differently; see *Ensuring code triggers a deprecation warning*.

property list: *list*[WarningMessage]

The list of recorded warnings.

__getitem__(*i*)

Get a recorded warning by index.

__iter__()

Iterate through the recorded warnings.

__len__()

The number of recorded warnings.

pop(*cls=<class 'Warning'>*)

Pop the first recorded warning which is an instance of `cls`, but not an instance of a child class of any other match. Raises `AssertionError` if there is no match.

clear()

Clear the list of recorded warnings.

request

Example: *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

The type of the `request` fixture.

A request object gives access to the requesting test context and has a `param` attribute in case the fixture is parametrized.

fixturename: Final

Fixture for which this request is being performed.

property scope: Literal['session', 'package', 'module', 'class', 'function']

Scope string, one of "function", "class", "module", "package", "session".

property fixturenames: *list*[str]

Names of all active fixtures in this request.

abstract property node

Underlying collection node (depends on current request scope).

property 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 path: Path

Path where the test function was collected.

property keywords: MutableMapping[str, Any]

Keywords/markers dictionary for the underlying node.

property session: *Session*

Pytest session object.

abstractmethod addfinalizer(*finalizer*)

Add finalizer/teardown function to be called without arguments after the last test within the requesting test context finished execution.

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 (*str | MarkDecorator*) – An object created by a call to `pytest.mark.NAME(...)`.

raiseerror(*msg*)

Raise a FixtureLookupError exception.

Parameters
msg (*str | None*) – An optional custom error message.

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.

This method can be used during the test setup phase or the test run phase. Avoid using it during the teardown phase.

Changed in version 9.1: Calling `request.getfixturevalue()` during teardown to request a fixture that was not already requested *is deprecated*.

Parameters
argname (*str*) – The fixture name.

Raises
pytest.FixtureLookupError – If the given fixture could not be found.

subtests

The `subtests` fixture enables declaring subtests inside test functions.

Tutorial: subtests

class Subtests

Subtests fixture, enables declaring subtests inside test functions via the *test()* method.

test(*msg=None*, ***kwargs*)

Context manager for subtests, capturing exceptions raised inside the subtest scope and reporting assertion failures and errors individually.

Usage

```
def test(subtests):
    for i in range(5):
        with subtests.test("custom message", i=i):
            assert i % 2 == 0
```

param msg
If given, the message will be shown in the test report in case of subtest failure.

param kwargs
Arbitrary values that are also added to the subtest report.

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.

exception TimeoutExpired

property tmpdir: LocalPath

Temporary directory where tests are executed.

make_hook_recorder(*pluginmanager*)

See *Pytester.make_hook_recorder()*.

chdir()

See *Pytester.chdir()*.

makefile(*ext*, **args*, ***kwargs*)

See *Pytester.makefile()*.

makeconftest(*source*)

See *Pytester.makeconftest()*.

makeini(*source*)

See *Pytester.makeini()*.

getinicfg(*source*)

See *Pytester.getinicfg()*.

makepyprojecttoml(*source*)

See *Pytester.makepyprojecttoml()*.

makepyfile(**args*, ***kwargs*)

See *Pytester.makepyfile()*.

maketxtfile(**args*, ***kwargs*)

See *Pytester.maketxtfile()*.

syspathinsert(*path=None*)

See *Pytester.syspathinsert()*.

mkdir(*name*)

See *Pytester.mkdir()*.

mkpydir(*name*)

See *Pytester.mkpydir()*.

copy_example(*name=None*)

See *Pytester.copy_example()*.

getnode(*config*, *arg*)

See *Pytester.getnode()*.

getpathnode(*path*)

See *Pytester.getpathnode()*.

genitems(*colitems*)

See *Pytester.genitems()*.

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()*.

runpytest_inprocess(**args*, ***kwargs*)

See *Pytester.runpytest_inprocess()*.

runpytest(**args*, ***kwargs*)

See *Pytester.runpytest()*.

parseconfig(**args*)

See *Pytester.parseconfig()*.

parseconfigure(**args*)

See *Pytester.parseconfigure()*.

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()*.

popen(*cmdargs*, *stdout=-1*, *stderr=-1*, *stdin=NotSetType.token*, ***kw*)

See *Pytester.popen()*.

run(**cmdargs*, *timeout=None*, *stdin=NotSetType.token*)

See *Pytester.run()*.

runpython(*script*)

See *Pytester.runpython()*.

runpython_c(*command*)

See *Pytester.runpython_c()*.

runpytest_subprocess(**args*, *timeout=None*)

See *Pytester.runpytest_subprocess()*.

spawn_pytest(*string*, *expect_timeout=10.0*)

See *Pytester.spawn_pytest()*.

spawn(*cmd*, *expect_timeout=10.0*)

See *Pytester.spawn()*.

tmp_path

Tutorial: *How to use temporary directories and files in tests*

tmp_path()

Return a temporary directory (as `pathlib.Path` object) which is unique to each test function invocation. The temporary directory is created as a subdirectory of the base temporary directory, with configurable retention, as discussed in *Temporary directory location and retention*.

tmp_path_factory

Tutorial: *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, as discussed at *Temporary directory location and retention*.

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
Path

getbasetemp()

Return the base temporary directory, creating it if needed.

Returns
The base temporary directory.

Return type
Path

tmpdir

Tutorial: *The tmpdir and tmpdir_factory fixtures*

tmpdir()

Return a temporary directory (as legacy_path object) which is unique to each test function invocation. The temporary directory is created as a subdirectory of the base temporary directory, with configurable retention, as discussed in *Temporary directory location and retention*.

> **Note**
>
> These days, it is preferred to use `tmp_path`.
>
> *About the tmpdir and tmpdir_factory fixtures*.

tmpdir_factory

Tutorial: *The tmpdir and tmpdir_factory fixtures*

`tmpdir_factory` is an instance of *TempdirFactory*:

final class TempdirFactory

Backward compatibility wrapper that implements `py.path.local` for *TempPathFactory*.

> **Note**
>
> These days, it is preferred to use `tmp_path_factory`.
>
> *About the tmpdir and tmpdir_factory fixtures*.

mktemp(*basename*, *numbered=True*)

Same as *TempPathFactory.mktemp()*, but returns a `py.path.local` object.

getbasetemp()

Same as *TempPathFactory.getbasetemp()*, but returns a `py.path.local` object.

3.4.5 Hooks

Tutorial: *Writing plugins*

Reference to all hooks which can be implemented by *conftest.py files* and *plugins*.

@pytest.hookimpl

`@pytest.`**hookimpl**

pytest's decorator for marking functions as hook implementations.

See *Writing hook functions* and `pluggy.HookimplMarker()`.

@pytest.hookspec

`@pytest.`**hookspec**

pytest's decorator for marking functions as hook specifications.

See *Declaring new hooks* and `pluggy.HookspecMarker()`.

Bootstrapping hooks

Bootstrapping hooks called for plugins registered early enough (internal and third-party plugins).

pytest_load_initial_conftests(*early_config*, *parser*, *args*)

Called to implement the loading of *initial conftest files* ahead of command line option parsing.

Parameters

- **early_config** – The pytest config object.
- **args** – Arguments passed on the command line.
- **parser** – To add command line options.

Use in conftest plugins

This hook is not called for conftest files.

pytest_cmdline_parse(*pluginmanager*, *args*)

Return an initialized *Config*, parsing the specified args.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

> **Note**
>
> This hook is only called for plugin classes passed to the `plugins` arg when using *pytest.main* to perform an in-process test run.

Parameters

- **pluginmanager** – The pytest plugin manager.
- **args** – List of arguments passed on the command line.

Returns

A pytest config object.

Use in conftest plugins

This hook is not called for conftest files.

pytest_cmdline_main(*config*)

Called for performing the main command line action.

The default implementation will invoke the configure hooks and *pytest_runtestloop*.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters

config – The pytest config object.

Returns

The exit code.

Use in conftest plugins

This hook is only called for *initial conftests*.

Initialization hooks

Initialization hooks called for plugins and `conftest.py` files.

pytest_addoption(*parser*, *pluginmanager*)

Register argparse-style options and config-style config values, called once at the beginning of a test run.

Parameters

- **parser** – To add command line options, call *parser.addoption(...)*. To add config-file values call *parser.addini(...)*.
- **pluginmanager** – 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.

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 a configuration 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 hook wrappers.

Use in conftest plugins

If a conftest plugin implements this hook, it will be called immediately when the conftest is registered.

This hook is only called for *initial conftests*.

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 – The pytest plugin manager.

> **Note**
>
> This hook is incompatible with hook wrappers.

Use in conftest plugins

If a conftest plugin implements this hook, it will be called immediately when the conftest is registered.

pytest_configure(*config*)

Allow plugins and conftest files to perform initial configuration.

> **Note**
>
> This hook is incompatible with hook wrappers.

Parameters

config – The pytest config object.

Use in conftest plugins

This hook is called for every *initial conftest* file after command line options have been parsed. After that, the hook is called for other conftest files as they are registered.

pytest_unconfigure(*config*)

Called before test process is exited.

Parameters

config – The pytest config object.

Use in conftest plugins

Any conftest file can implement this hook.

pytest_sessionstart(*session*)

Called after the `Session` object has been created and before performing collection and entering the run test loop.

Parameters

session – The pytest session object.

Use in conftest plugins

This hook is only called for *initial conftests*.

pytest_sessionfinish(*session*, *exitstatus*)

Called after whole test run finished, right before returning the exit status to the system.

Parameters

- **session** – The pytest session object.
- **exitstatus** – The status which pytest will return to the system.

Use in conftest plugins

Any conftest file can implement this hook.

pytest_plugin_registered(*plugin*, *plugin_name*, *manager*)

A new pytest plugin got registered.

Parameters

- **plugin** – The plugin module or instance.
- **plugin_name** – The name by which the plugin is registered.
- **manager** – The pytest plugin manager.

Note

This hook is incompatible with hook wrappers.

Use in conftest plugins

If a conftest plugin implements this hook, it will be called immediately when the conftest is registered, once for each plugin registered thus far (including itself!), and for all plugins thereafter when they are registered.

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. Set `session.items` to the list of collected items
4. `pytest_collection_finish(session)`
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 – The pytest session object.

Use in conftest plugins

This hook is only called for *initial conftests*.

pytest_ignore_collect(*collection_path*, *config*)

Return `True` to ignore this path for collection.

Return `None` to let other plugins ignore the path for collection.

Returning `False` will forcefully *not* ignore this path for collection, without giving a chance for other plugins to ignore this path.

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

- **collection_path** (*pathlib.Path*) – The path to analyze.
- **config** – The pytest config object.

Changed in version 7.0.0: The `collection_path` parameter was added as a `pathlib.Path` equivalent of the `path` parameter. The `path` parameter has been deprecated and removed in pytest 9.0.0.

Use in conftest plugins

Any conftest file can implement this hook. For a given collection path, only conftest files in parent directories of the collection path are consulted (if the path is a directory, its own conftest file is *not* consulted - a directory cannot ignore itself!).

pytest_collect_directory(*path*, *parent*)

Create a *Collector* for the given directory, or None if not relevant.

Added in version 8.0.

For best results, the returned collector should be a subclass of *Directory*, but this is not required.

The new node needs to have the specified `parent` as a parent.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Parameters
path (*pathlib.Path*) – The path to analyze.

See *Using a custom directory collector* for a simple example of use of this hook.

Use in conftest plugins

Any conftest file can implement this hook. For a given collection path, only conftest files in parent directories of the collection path are consulted (if the path is a directory, its own conftest file is *not* consulted - a directory cannot collect itself!).

pytest_collect_file(*file_path*, *parent*)

Create a *Collector* for the given path, or None if not relevant.

For best results, the returned collector should be a subclass of *File*, but this is not required.

The new node needs to have the specified `parent` as a parent.

Parameters
file_path (*pathlib.Path*) – The path to analyze.

Changed in version 7.0.0: The `file_path` parameter was added as a `pathlib.Path` equivalent of the `path` parameter. The `path` parameter has been deprecated and removed in pytest 9.0.0.

Use in conftest plugins

Any conftest file can implement this hook. For a given file path, only conftest files in parent directories of the file path are consulted.

pytest_pycollect_makemodule(*module_path*, *parent*)

Return a *pytest.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
module_path (*pathlib.Path*) – The path of the module to collect.

Changed in version 7.0.0: The `module_path` parameter was added as a `pathlib.Path` equivalent of the `path` parameter. The `path` parameter has been deprecated in favor of `module_path` and removed in pytest 9.0.0.

Use in conftest plugins

Any conftest file can implement this hook. For a given parent collector, only conftest files in the collector's directory and its parent directories are consulted.

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** – The module/class collector.
- **name** – The name of the object in the module/class.
- **obj** – The object.

Returns

The created items/collectors.

Use in conftest plugins

Any conftest file can implement this hook. For a given collector, only conftest files in the collector's directory and its parent directories are consulted.

pytest_generate_tests(*metafunc*)

Generate (multiple) parametrized calls to a test function.

Parameters

metafunc – The *Metafunc* helper for the test function.

Use in conftest plugins

Any conftest file can implement this hook. For a given function definition, only conftest files in the functions's directory and its parent directories are consulted.

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** – The pytest config object.
- **val** – The parametrized value.
- **argname** – The automatic parameter name produced by pytest.

Use in conftest plugins

Any conftest file can implement this hook.

Hooks for influencing test skipping:

pytest_markeval_namespace(*config*)

Called when constructing the globals dictionary used for evaluating string conditions in xfail/skipif markers.

This is useful when the condition for a marker requires objects that are expensive or impossible to obtain during collection time, which is required by normal boolean conditions.

Added in version 6.2.

Parameters
config – The pytest config object.

Returns
A dictionary of additional globals to add.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in parent directories of the item are consulted.

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.

When items are deselected (filtered out from `items`), the hook *pytest_deselected* must be called explicitly with the deselected items to properly notify other plugins, e.g. with `config.hook.pytest_deselected(items=deselected_items)`.

Parameters

- **session** – The pytest session object.
- **config** – The pytest config object.
- **items** – List of item objects.

Use in conftest plugins

Any conftest plugin can implement this hook.

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 – The pytest session object.

Use in conftest plugins

Any conftest plugin can implement this hook.

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 – The pytest session 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.

Use in conftest plugins

Any conftest file can implement this hook.

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 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 `CallInfo(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** – Test item for which the runtest protocol is performed.
- **nextitem** – The scheduled-to-be-next test item (or None if this is the end my friend).

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.

Use in conftest plugins

Any conftest file can implement this hook.

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** – Full node ID of the item.
- **location** – A tuple of `(filename, lineno, testname)` where `filename` is a file path relative to `config.rootpath` and `lineno` is 0-based.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

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** – Full node ID of the item.
- **location** – A tuple of `(filename, lineno, testname)` where `filename` is a file path relative to `config.rootpath` and `lineno` is 0-based.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

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 – The item.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

pytest_runtest_call(*item*)

Called to run the test for test item (the call phase).

The default implementation calls `item.runtest()`.

Parameters

item – The item.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

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

- **item** – The item.
- **nextitem** – 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.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

pytest_runtest_makereport(*item*, *call*)

Called to create a *TestReport* for each of the setup, call and teardown runtest phases of a test item.

See *pytest_runtest_protocol* for a description of the runtest protocol.

Parameters

- **item** – The item.
- **call** – The *CallInfo* for the phase.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

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 – The function item.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

Reporting hooks

Session related reporting hooks:

pytest_collectstart(*collector*)

Collector starts collecting.

Parameters
collector – The collector.

Use in conftest plugins

Any conftest file can implement this hook. For a given collector, only conftest files in the collector's directory and its parent directories are consulted.

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 – The collector.

Use in conftest plugins

Any conftest file can implement this hook. For a given collector, only conftest files in the collector's directory and its parent directories are consulted.

pytest_itemcollected(*item*)

We just collected a test item.

Parameters
item – The item.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

pytest_collectreport(*report*)

Collector finished collecting.

Parameters
report – The collect report.

Use in conftest plugins

Any conftest file can implement this hook. For a given collector, only conftest files in the collector's directory and its parent directories are consulted.

pytest_deselected(*items*)

Called for deselected test items, e.g. by keyword.

Note that this hook has two integration aspects for plugins:

- it can be *implemented* to be notified of deselected items
- it must be *called* from *pytest_collection_modifyitems* implementations when items are deselected (to properly notify other plugins).

May be called multiple times.

Parameters

items – The items.

Use in conftest plugins

Any conftest file can implement this hook.

pytest_report_header(*config*, *start_path*)

Return a string or list of strings to be displayed as header info for terminal reporting.

Parameters

- **config** – The pytest config object.
- **start_path** (*pathlib.Path*) – The starting dir.

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 7.0.0: The `start_path` parameter was added as a `pathlib.Path` equivalent of the `start-dir` parameter. The `startdir` parameter has been deprecated and removed in pytest 9.0.0.

Use in conftest plugins

This hook is only called for *initial conftests*.

pytest_report_collectionfinish(*config*, *start_path*, *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.

Added in version 3.2.

Parameters

- **config** – The pytest config object.
- **start_path** (*pathlib.Path*) – The starting dir.
- **items** – List of pytest items that are going to be executed; this list should not be modified.

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 7.0.0: The `start_path` parameter was added as a `pathlib.Path` equivalent of the `start-dir` parameter. The `startdir` parameter has been deprecated and removed in pytest 9.0.0.

Use in conftest plugins

Any conftest plugin can implement this hook.

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** – The report object whose status is to be returned.
- **config** – The pytest config object.

Returns

The test status.

Stops at first non-None result, see *firstresult: stop at first non-None result*.

Use in conftest plugins

Any conftest plugin can implement this hook.

pytest_report_to_serializable(*config*, *report*)

Serialize the given report object into a data structure suitable for sending over the wire, e.g. converted to JSON.

Parameters

- **config** – The pytest config object.
- **report** – The report.

Use in conftest plugins

Any conftest file can implement this hook. The exact details may depend on the plugin which calls the hook.

pytest_report_from_serializable(*config*, *data*)

Restore a report object previously serialized with *pytest_report_to_serializable*.

Parameters

config – The pytest config object.

Use in conftest plugins

Any conftest file can implement this hook. The exact details may depend on the plugin which calls the hook.

pytest_terminal_summary(*terminalreporter*, *exitstatus*, *config*)

Add a section to terminal summary reporting.

Parameters

- **terminalreporter** – The internal terminal reporter object.

- **exitstatus** – The exit status that will be reported back to the OS.
- **config** – The pytest config object.

Added in version 4.2: The `config` parameter.

Use in conftest plugins

Any conftest plugin can implement this hook.

pytest_fixture_setup(*fixturedef*, *request*)

Perform fixture setup execution.

Parameters

- **fixturedef** – The fixture definition object.
- **request** – The fixture request object.

Returns

The return value of the call to the fixture function.

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.

Use in conftest plugins

Any conftest file can implement this hook. For a given fixture, only conftest files in the fixture scope's directory and its parent directories are consulted.

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** – The fixture definition object.
- **request** – The fixture request object.

Use in conftest plugins

Any conftest file can implement this hook. For a given fixture, only conftest files in the fixture scope's directory and its parent directories are consulted.

pytest_warning_recorded(*warning_message*, *when*, *nodeid*, *location*)

Process a warning captured by the internal pytest warnings plugin.

Parameters

- **warning_message** – 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** – Indicates when the warning was captured. Possible values:
  - `"config"`: during pytest configuration/initialization stage.

- `"collect"`: during test collection.
- `"runtest"`: during test execution.

- **nodeid** – Full id of the item. Empty string for warnings that are not specific to a particular node.
- **location** – 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.

Added in version 6.0.

Use in conftest plugins

Any conftest file can implement this hook. If the warning is specific to a particular node, only conftest files in parent directories of the node are consulted.

Central hook for reporting about test execution:

pytest_runtest_logreport(*report*)

Process the *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.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

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** – The pytest config object.
- **op** – The operator, e.g. `"=="`, `"!="`, `"not in"`.
- **left** – The left operand.
- **right** – The right operand.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

pytest_assertion_pass(*item*, *lineno*, *orig*, *expl*)

Called whenever an assertion passes.

Added 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* configuration option:

```
[pytest]
enable_assertion_pass_hook = true
```

```
[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 object of current test.
- **lineno** – Line number of the assert statement.
- **orig** – String with the original assertion.
- **expl** – String with the assert explanation.

Use in conftest plugins

Any conftest file can implement this hook. For a given item, only conftest files in the item's directory and its parent directories are consulted.

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** – The exception repr object.
- **excinfo** – The exception info.

Use in conftest plugins

Any conftest plugin can implement this hook.

pytest_keyboard_interrupt(*excinfo*)

Called for keyboard interrupt.

Parameters

excinfo – The exception info.

Use in conftest plugins

Any conftest plugin can implement this hook.

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 *CollectReport*.

May be called during runtest of an item (see *pytest_runtest_protocol*), in which case `report` is a *TestReport*.

This hook is not called if the exception that was raised is an internal exception like `skip.Exception`.

Parameters

- **node** – The item or collector.
- **call** – The call information. Contains the exception.
- **report** – The collection or test report.

Use in conftest plugins

Any conftest file can implement this hook. For a given node, only conftest files in parent directories of the node are consulted.

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** – The pytest config object.
- **pdb** – The Pdb instance.

Use in conftest plugins

Any conftest plugin can implement this hook.

pytest_leave_pdb(*config*, *pdb*)

Called when leaving pdb (e.g. with continue after pdb.set_trace()).

Can be used by plugins to take special action just after the python debugger leaves interactive mode.

Parameters

- **config** – The pytest config object.
- **pdb** – The Pdb instance.

Use in conftest plugins

Any conftest plugin can implement this hook.

3.4.6 Collection tree objects

These are the collector and item classes (collectively called "nodes") which make up the collection tree.

Node

class Node

Bases: ABC

Base class of *Collector* and *Item*, the components of the test collection tree.

`Collector`'s are the internal nodes of the tree, and `Item`'s are the leaf nodes.

fspath: LEGACY_PATH

A `LEGACY_PATH` copy of the *path* attribute. Intended for usage for methods not migrated to `pathlib.Path` yet, such as *Item.reportinfo*. Will be deprecated in a future release, prefer using *path* instead.

name: str

A unique name within the scope of the parent node.

parent

The parent collector node.

config: *Config*

The pytest config object.

session: *Session*

The pytest session this node is part of.

path: pathlib.Path

Filesystem path where this node was collected from.

keywords: MutableMapping[str, Any]

Keywords/markers collected from all scopes.

own_markers: *list*[*Mark*]

The marker objects belonging to this node.

extra_keyword_matches: set[str]

Allow adding of extra keywords to use for matching.

stash: *Stash*

A place where plugins can store information on the node for their own use.

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 (`Node`) – The parent node of this Node.

property ihook: HookRelay

Path-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.

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.

iter_parents()

Iterate over all parent collectors starting from and including self up to the root of the collection tree.

Added in version 8.1.

listchain()

Return a list of all parent collectors starting from the root of the collection tree down to and including self.

add_marker(*marker*, *append=True*)

Dynamically add a marker object to the node.

Parameters

- **marker** (*str | MarkDecorator*) – The marker.
- **append** (*bool*) – Whether to append the marker, or prepend it.

iter_markers(*name=None*)

Iterate over all markers of the node.

Parameters
name (*str | None*) – If given, filter the results by the name attribute.

Returns
An iterator of the markers of the node.

Return type
Iterator[Mark]

iter_markers_with_node(*name=None*)

Iterate over all markers of the node.

Parameters
name (*str | None*) – If given, filter the results by the name attribute.

Returns
An iterator of (node, mark) tuples.

Return type
Iterator[tuple[Node, Mark]]

get_closest_marker(*name: str*) → *Mark* | None

get_closest_marker(*name: str*, *default:* Mark) → *Mark*

Return the first marker matching the name, from closest (for example function) to farther level (for example module level).

Parameters

- **default** (*Mark | None*) – Fallback return value if no marker was found.
- **name** (*str*) – Name to filter by.

listextrakeywords()

Return a set of all extra keywords in self and any parents.

addfinalizer(*fin*)

Register a function to be called without arguments 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().

getparent(*cls*)

Get the closest parent node (including self) which is an instance of the given class.

Parameters
cls (type[*_NodeType*]) – The node class to search for.

Returns
The node, if found.

Return type
_NodeType | None

repr_failure(*excinfo*, *style=None*)

Return a representation of a collection or test failure.

> **See also**
>
> *Working with non-python tests*

Parameters
excinfo (ExceptionInfo[*BaseException*]) – Exception information for the failure.

Collector

class Collector

Bases: *Node*, ABC

Base class of all collectors.

Collector create children through collect() and thus iteratively build the collection tree.

exception CollectError

Bases: Exception

An error during collection, contains a custom message.

abstractmethod collect()

Collect children (items and collectors) for this collector.

repr_failure(*excinfo*)

Return a representation of a collection failure.

Parameters
excinfo (ExceptionInfo[*BaseException*]) – Exception information for the failure.

config
: The pytest config object.

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.

session
: The pytest session this node is part of.

Item

class Item

Bases: *Node*, ABC

Base class of all test invocation items.

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.

config
: The pytest config object.

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.

session
: The pytest session this node is part of.

abstractmethod runtest()
: Run the test case for this item.

Must be implemented by subclasses.

> **See also**
>
> *Working with non-python tests*

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.

reportinfo()

Get location information for this item for test reports.

Returns a tuple with three elements:

- The path of the test (default `self.path`)
- The 0-based line number of the test (default `None`)
- A name of the test to be shown (default `""`)

See also

Working with non-python tests

property location: tuple[str, int | None, str]

Returns a tuple of `(relfspath, lineno, testname)` for this item where `relfspath` is file path relative to `config.rootpath` and lineno is a 0-based line number.

File

class File

Bases: *FSCollector*, ABC

Base class for collecting tests from a file.

Working with non-python tests.

config

The pytest config object.

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.

session

The pytest session this node is part of.

FSCollector

class FSCollector

Bases: *Collector*, ABC

Base class for filesystem collectors.

path

Filesystem path where this node was collected from.

classmethod from_parent(*parent*, *, *fspath=None*, *path=None*, ***kw*)

The public constructor.

config

The pytest config object.

name

A unique name within the scope of the parent node.

parent

The parent collector node.

session

The pytest session this node is part of.

Session

final class Session

Bases: *Collector*

The root of the collection tree.

`Session` collects the initial paths given as arguments to pytest.

exception Interrupted

Bases: `KeyboardInterrupt`

Signals that the test run was interrupted.

exception Failed

Bases: `Exception`

Signals a stop as failed test run.

property startpath: Path

The path from which pytest was invoked.

Added in version 7.0.0.

isinitpath(*path*, *, *with_parents=False*)

Is path an initial path?

An initial path is a path explicitly given to pytest on the command line.

Parameters

with_parents (*bool*) – If set, also return True if the path is a parent of an initial path.

Changed in version 8.0: Added the `with_parents` parameter.

perform_collect(*args: Sequence[str] | None = None*, *genitems: Literal[True] = True*) → Sequence[*Item*]

perform_collect(*args: Sequence[str] | None = None*, *genitems: bool = True*) → Sequence[*Item* | *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.

collect()

Collect children (items and collectors) for this collector.

config

The pytest config object.

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.

session

The pytest session this node is part of.

Package

class Package

Bases: *Directory*

Collector for files and directories in a Python packages – directories with an `__init__.py` file.

> **Note**
>
> Directories without an `__init__.py` file are instead collected by *Dir* by default. Both are *Directory* collectors.

Changed in version 8.0: Now inherits from *Directory*.

collect()
Collect children (items and collectors) for this collector.

config
The pytest config object.

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.

session
The pytest session this node is part of.

Module

class Module
Bases: *File*, PyCollector

Collector for test classes and functions in a Python module.

collect()
Collect children (items and collectors) for this collector.

config
The pytest config object.

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.

session
The pytest session this node is part of.

Class

class Class
Bases: PyCollector

Collector for test methods (and nested classes) in a Python class.

classmethod from_parent(*parent*, *, *name*, *obj=None*, ***kw*)

The public constructor.

collect()

Collect children (items and collectors) for this collector.

config

The pytest config object.

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.

session

The pytest session this node is part of.

Function

class Function

Bases: `PyobjMixin`, *Item*

Item responsible for setting up and executing a Python test function.

Parameters

- **name** – The full function name, including any decorations like those added by parametrization (`my_func[my_param]`).
- **parent** – The parent Node.
- **config** – The pytest Config object.
- **callspec** – If given, this function has been parametrized and the callspec contains meta information about the parametrization.
- **callobj** – If given, the object which will be called when the Function is invoked, otherwise the callobj will be obtained from `parent` using `originalname`.
- **keywords** – Keywords bound to the function object for "-k" matching.
- **session** – The pytest Session object.
- **fixtureinfo** – Fixture information already resolved at this fixture node..
- **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).

Added in version 3.0.

classmethod from_parent(*parent*, ***kw*)

The public constructor.

property function

Underlying python 'function' object.

property instance

Python instance object the function is bound to.

Returns None if not a test method, e.g. for a standalone test function, a class or a module.

runtest()

Execute the underlying test function.

repr_failure(*excinfo*)

Return a representation of a collection or test failure.

> **See also**
>
> *Working with non-python tests*

Parameters

excinfo (ExceptionInfo[*BaseException*]) – Exception information for the failure.

config

The pytest config object.

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.

session

The pytest session this node is part of.

FunctionDefinition

class FunctionDefinition

Bases: *Function*

This class is a stop gap solution until we evolve to have actual function definition nodes and manage to get rid of `metafunc`.

runtest()

Execute the underlying test function.

config
The pytest config object.

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.

session
The pytest session this node is part of.

setup()
Execute the underlying test function.

3.4.7 Objects

Objects accessible from *fixtures* or *hooks* or importable from `pytest`.

CallInfo

final class CallInfo
Result/Exception info of a function invocation.

excinfo: *ExceptionInfo*[BaseException] | None
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: 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[[], _pytest.runner.TResult]*) – The function to call. Called without arguments.

- **when** (*Literal['collect', 'setup', 'call', 'teardown']*) – The phase in which the function is called.
- **reraise** (*type[BaseException] | tuple[type[BaseException], ...] | None*) – Exception or exceptions that shall propagate if raised by the function, instead of being wrapped in the CallInfo.

CollectReport

final class CollectReport

Bases: `BaseReport`

Collection report object.

Reports can contain arbitrary extra attributes.

nodeid: str

Normalized collection nodeid.

outcome: Literal['passed', 'failed', 'skipped']

Test outcome, always one of "passed", "failed", "skipped".

longrepr: None | *ExceptionInfo*[BaseException] | tuple[str, int, str] | str | 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.

Added in version 3.5.

property capstderr: str

Return captured text from stderr, if capturing is enabled.

Added in version 3.0.

property capstdout: str

Return captured text from stdout, if capturing is enabled.

Added 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 failed: bool

Whether the outcome is failed.

property fspath: str

The path portion of the reported node, as a string.

property head_line: str | None

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`.

Added in version 3.0.

property passed: bool

Whether the outcome is passed.

property skipped: bool

Whether the outcome is skipped.

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.

final class InvocationParams(*, *args*, *plugins*, *dir*)

Holds parameters passed during *`pytest.main()`*.

The object attributes are read-only.

Added in version 5.1.

> **Note**
>
> Note that the environment variable `PYTEST_ADDOPTS` and the `addopts` configuration option are handled by pytest, not being included in the `args` attribute.
>
> Plugins accessing `InvocationParams` must be aware of that.

args: tuple[str, ...]

The command-line arguments as passed to *pytest.main()*.

plugins: Sequence[str | object] | None

Extra plugins, might be `None`.

dir: Path

The directory from which *pytest.main()* was invoked.

class ArgsSource(**values*)

Indicates the source of the test arguments.

Added in version 7.2.

ARGS = 1

Command line arguments.

INVOCATION_DIR = 2

Invocation directory.

TESTPATHS = 3

'testpaths' configuration value.

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 rootpath: Path

The path to the *rootdir*.

Added in version 6.1.

property inipath: Path | None

The path to the *configfile*.

Added in version 6.1.

add_cleanup(*func*)

Add a function to be called when the config object gets out of use (usually coinciding with pytest_unconfigure).

classmethod fromdictargs(*option_dict*, *args*)

Constructor usable for subprocesses.

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 hook wrappers 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.

addinivalue_line(*name*, *line*)

Add a line to a configuration 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.

getini(*name*)

Return configuration value the an *configuration file*.

If a configuration value is not defined in a *configuration file*, then the `default` value provided while registering the configuration through *parser.addini* will be returned. Please note that you can even provide `None` as a valid default value.

If `default` is not provided while registering using *parser.addini*, then a default value based on the `type` parameter passed to *parser.addini* will be returned. The default values based on `type` are: `paths`, `pathlist`, `args` and `linelist` : empty list `[]` `bool` : `False` `string` : empty string `""` `int` : `0` `float` : `0.0`

If neither the `default` nor the `type` parameter is passed while registering the configuration through *parser.addini*, then the configuration is treated as a string and a default empty string '' is returned.

If the specified name hasn't been registered through a prior *parser.addini* call (usually from a plugin), a ValueError is raised.

getoption(*name*, *default=NotSetType.token*, *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** (*Any*) – Fallback value if no option of that name is **declared** via *pytest_addoption*. Note this parameter will be ignored when the option is **declared** even if the option's value is `None`.
- **skip** (*bool*) – If `True`, raise *pytest.skip()* if option is undeclared or has a `None` value. Note that even if `True`, if a default was specified it will be returned instead of a skip.

getvalue(*name*, *path=None*)

Deprecated, use getoption() instead.

getvalueorskip(*name*, *path=None*)

Deprecated, use getoption(skip=True) instead.

VERBOSITY_ASSERTIONS: Final = 'assertions'

Verbosity type for failed assertions (see *verbosity_assertions*).

VERBOSITY_TEST_CASES: Final = 'test_cases'

Verbosity type for test case execution (see *verbosity_test_cases*).

VERBOSITY_SUBTESTS: Final = 'subtests'

Verbosity type for failed subtests (see *verbosity_subtests*).

get_verbosity(*verbosity_type=None*)

Retrieve the verbosity level for a fine-grained verbosity type.

Parameters

verbosity_type (*str | None*) – Verbosity type to get level for. If a level is configured for the given type, that value will be returned. If the given type is not a known verbosity type, the global verbosity level will be returned. If the given type is None (default), the global verbosity level will be returned.

To configure a level for a fine-grained verbosity type, the configuration file should have a setting for the configuration name and a numeric value for the verbosity level. A special value of "auto" can be used to explicitly use the global verbosity level.

Example:

```
[tool.pytest]
verbosity_assertions = 2
```

```
[pytest]
verbosity_assertions = 2
```

```
pytest -v
```

```
print(config.get_verbosity())  # 1
print(config.get_verbosity(Config.VERBOSITY_ASSERTIONS))  # 2
```

Dir

final class Dir

Collector of files in a file system directory.

Added in version 8.0.

> **Note**
>
> Python directories with an `__init__.py` file are instead collected by *Package* by default. Both are *Directory* collectors.

classmethod from_parent(*parent*, *, *path*)

The public constructor.

Parameters

- **parent** (*nodes.Collector*) – The parent collector of this Dir.
- **path** (*pathlib.Path*) – The directory's path.

collect()

Collect children (items and collectors) for this collector.

config

The pytest config object.

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.

session

The pytest session this node is part of.

Directory

class Directory

Base class for collecting files from a directory.

A basic directory collector does the following: goes over the files and sub-directories in the directory and creates collectors for them by calling the hooks *pytest_collect_directory* and *pytest_collect_file*, after checking that they are not ignored using *pytest_ignore_collect*.

The default directory collectors are *Dir* and *Package*.

Added in version 8.0.

Using a custom directory collector.

config

The pytest config object.

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.

session

The pytest session this node is part of.

ExceptionInfo

final class ExceptionInfo

Wraps sys.exc_info() objects and offers help for navigating the traceback.

classmethod from_exception(*exception*, *exprinfo=None*)

Return an ExceptionInfo for an existing exception.

The exception must have a non-`None` `__traceback__` attribute, otherwise this function fails with an assertion error. This means that the exception must have been raised, or added a traceback with the `with_traceback()` method.

Parameters

exprinfo (*str | None*) – A text string helping to determine if we should strip `AssertionError` from the output. Defaults to the exception message/`__str__()`.

Added in version 7.4.

classmethod from_exc_info(*exc_info*, *exprinfo=None*)

Like *from_exception()*, but using old-style exc_info tuple.

classmethod from_current(*exprinfo=None*)

Return an ExceptionInfo matching the current traceback.

Warning

Experimental API

Parameters

exprinfo (*str | None*) – A text string helping to determine if we should strip `AssertionError` from the output. Defaults to the exception message/`__str__()`.

classmethod for_later()

Return an unfilled ExceptionInfo.

fill_unfilled(*exc_info*)

Fill an unfilled ExceptionInfo created with `for_later()`.

property type: *type*[E]

The exception class.

property value: E

The exception value.

property tb: TracebackType

The exception raw traceback.

property typename: str

The type name of the exception.

property traceback: Traceback

The traceback.

exconly(*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).

errisinstance(*exc*)

Return True if the exception is an instance of exc.

Consider using `isinstance(excinfo.value, exc)` instead.

getrepr(*showlocals=False*, *style='long'*, *abspath=False*, *tbfilter=True*, *funcargs=False*, *truncate_locals=True*, *truncate_args=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|line|no|native|value traceback style.
- **abspath** (*bool*) – If paths should be changed to absolute or left unchanged.
- **tbfilter** (*bool | Callable[[ExceptionInfo[BaseException]], Traceback]*) – A filter for traceback entries.
  - If false, don't hide any entries.
  - If true, hide internal entries and entries that contain a local variable `__tracebackhide__ = True`.
  - If a callable, delegates the filtering to the callable.

  Ignored if `style` is `"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.
- **truncate_args** (*bool*) – With `showargs==True`, make sure args can be safely represented as strings.
- **chain** (*bool*) – If chained exceptions in Python 3 should be shown.

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.

group_contains(*expected_exception*, *, *match=None*, *depth=None*)

Check whether a captured exception group contains a matching exception.

Parameters

- **expected_exception** (*Type[BaseException] | Tuple[Type[BaseException]]*) – The expected exception type, or a tuple if one of multiple possible exception types are expected.
- **match** (*str | re.Pattern[str] | None*) – If specified, a string containing a regular expression, or a regular expression object, that is tested against the string representation of the exception and its `PEP-678 <https://peps.python.org/pep-0678/> __notes__` using `re.search()`.

  To match a literal string that may contain special characters, the pattern can first be escaped with `re.escape()`.
- **depth** (*Optional[int]*) – If `None`, will search for a matching exception at any nesting depth. If >= 1, will only match an exception if it's at the specified depth (depth = 1 being the exceptions contained within the topmost exception group).

Added in version 8.0.

> **Warning**
>
> This helper makes it easy to check for the presence of specific exceptions, but it is very bad for checking that the group does *not* contain *any other exceptions*. You should instead consider using *pytest.RaisesGroup*

ExitCode

class ExitCode(**values*)

Encodes the valid exit codes by pytest.

Currently users and plugins may supply other exit codes as well.

Added in version 5.0.

FixtureDef

class FixtureDef

Bases: `Generic[FixtureValue]`

A container for a fixture definition.

Note: At this time, only explicitly documented fields and methods are considered public stable API.

property scope: Literal['session', 'package', 'module', 'class', 'function']

Scope string, one of "function", "class", "module", "package", "session".

execute(*request*)

Return the value of this fixture, executing it if not cached.

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()`.

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.

MarkGenerator

final class MarkGenerator

Factory for *MarkDecorator* objects - exposed as a `pytest.mark` singleton instance.

Example:

```
import pytest


@pytest.mark.slowtest
def test_function():
    pass
```

applies a 'slowtest' *Mark* on `test_function`.

Mark

final class Mark

A pytest mark.

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.

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 at test setup time instead.

Can be called multiple times per test function (but only on different argument names), 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** (*str | Sequence[str]*) – A comma-separated string denoting one or more argument names, or a list/tuple of argument strings.
- **argvalues** (*Iterable[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.

  Changed in version 9.1: Passing a non-`Collection` iterable (such as a generator or iterator) is deprecated. See *Non-Collection iterables in @pytest.mark.parametrize* for details.
- **indirect** (*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** (*Iterable[object | None] | Callable[[Any], object | None] | None*) – 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.

  Added in version 8.4: *pytest.HIDDEN_PARAM* means to hide the parameter set from the test name. Can only be used at most 1 time, as test names need to be unique.

  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** (*Literal['session', 'package', 'module', 'class', 'function'] | None*) – 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.

Parser

final class Parser

Parser for command line arguments and config-file values.

Variables

extra_info – Dict of generic param -> value to display in case there's an error processing the command line arguments.

getgroup(*name*, *description=''*, *after=None*)

Get (or create) a named option Group.

Parameters

- **name** (*str*) – Name of the option group.
- **description** (*str*) – Long description for –help output.
- **after** (*str | None*) – Name of another group, used for ordering –help output.

Returns

The option group.

Return type

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.

Parameters

- **opts** (*str*) – Option names, can be short or long options.
- **attrs** (*Any*) – Same attributes as the argparse library's `add_argument()` function accepts.

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 the known arguments at this point.

Returns

An argparse namespace object.

Return type

Namespace

parse_known_and_unknown_args(*args*, *namespace=None*)

Parse the known arguments at this point, and also return the remaining unknown flag arguments.

Returns

A tuple containing an argparse namespace object for the known arguments, and a list of unknown flag arguments.

Return type

tuple[*Namespace*, *list*[str]]

addini(*name*, *help*, *type=None*, *default=NotSetType.token*, ***, *aliases=()*)

Register a configuration file option.

Parameters

- **name** (*str*) – Name of the configuration.
- **type** (*Literal['string', 'paths', 'pathlist', 'args', 'linelist', 'bool', 'int', 'float'] | None*) – Type of the configuration. 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
  - `int`: an integer
  - `float`: a floating-point number

  Added in version 8.4: The `float` and `int` types.

  For `paths` and `pathlist` types, they are considered relative to the config-file. In case the execution is happening without a config-file defined, they will be considered relative to the current working directory (for example with `--override-ini`).

  Added in version 7.0: The `paths` variable type.

  Added in version 8.1: Use the current working directory to resolve `paths` and `pathlist` in the absence of a config-file.

  Defaults to `string` if `None` or not passed.
- **default** (*Any*) – Default value if no config-file option exists but is queried.
- **aliases** (*Sequence[str]*) – Additional names by which this option can be referenced. Aliases resolve to the canonical name.

  Added in version 9.0: The `aliases` parameter.

The value of configuration keys can be retrieved via a call to *config.getini(name)*.

OptionGroup

class OptionGroup

A group of options shown in its own section.

addoption(**opts*, ***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

- **opts** (*str*) – Option names, can be short or long options. Note that lower-case short options (e.g. `-x`) are reserved.

- **attrs** (*Any*) – Same attributes as the argparse library's `add_argument()` function accepts.

PytestPluginManager

final class PytestPluginManager

Bases: `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.

skipped_plugins: *list*[tuple[str, str]]

rewrite_hook: RewriteHook

register(*plugin*, *name=None*)

Register a plugin and return its name.

Parameters
name (*str | None*) – The name under which to register the plugin. If not specified, a name is generated using *get_canonical_name()*.

Returns
The plugin name. If the name is blocked from registering, returns `None`.

Return type
str | None

If the plugin is already registered, raises a `ValueError`.

getplugin(*name*)

hasplugin(*name*)

Return whether a plugin with the given name is registered.

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.

add_hookcall_monitoring(*before*, *after*)

Add before/after tracing functions for all hooks.

Returns an undo function which, when called, removes the added tracers.

`before(hook_name, hook_impls, kwargs)` will be called ahead of all hook calls and receive a hook-caller 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 `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 as hook specifications if they have been decorated with a matching `HookspecMarker`.

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.

Returns an undo function which, when called, removes the added tracing.

get_canonical_name(*plugin*)

Return a canonical name for a plugin object.

Note that a plugin may be registered under a different name specified by the caller of *register(plugin, name)*. To obtain the name of a registered plugin use *get_name(plugin)* instead.

get_hookcallers(*plugin*)

Get all hook callers for the specified plugin.

Returns
The hook callers, or `None` if `plugin` is not registered in this plugin manager.

Return type
list[*HookCaller*] | None

get_name(*plugin*)

Return the name the plugin is registered under, or `None` if is isn't.

get_plugin(*name*)

Return the plugin registered under the given name, if any.

get_plugins()

Return a set of all registered plugin objects.

has_plugin(*name*)

Return whether a plugin with the given name is registered.

is_blocked(*name*)

Return whether the given plugin name is blocked.

is_registered(*plugin*)

Return whether the plugin is already registered.

list_name_plugin()

Return a list of (name, plugin) pairs for all registered plugins.

list_plugin_distinfo()

Return a list of (plugin, distinfo) pairs 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 | None*) – If given, loads only plugins with the given `name`.

Returns

The number of plugins loaded by this call.

Return type

int

set_blocked(*name*)

Block registrations of the given name, unregister if already registered.

subset_hook_caller(*name*, *remove_plugins*)

Return a proxy `HookCaller` instance for the named method which manages calls to all registered plugins except the ones from remove_plugins.

unblock(*name*)

Unblocks a name.

Returns whether the name was actually blocked.

unregister(*plugin=None*, *name=None*)

Unregister a plugin and all of its hook implementations.

The plugin can be specified either by the plugin object or the plugin name. If both are specified, they must agree.

Returns the unregistered plugin, or `None` if not found.

project_name

The project name.

hook

The "hook relay", used to call a hook on all registered plugins. See Calling hooks.

trace

The tracing entry point. See Built-in tracing.

RaisesExc

final class RaisesExc

Added in version 8.4.

This is the class constructed when calling *pytest.raises()*, but may be used directly as a helper class with *RaisesGroup* when you want to specify requirements on sub-exceptions.

You don't need this if you only want to specify the type, since *RaisesGroup* accepts `type[BaseException]`.

Parameters

- **expected_exception** (*type[BaseException] | tuple[type[BaseException]] | None*) – The expected type, or one of several possible types. May be `None` in order to only make use of `match` and/or `check`

  The type is checked with `isinstance()`, and does not need to be an exact match. If that is wanted you can use the `check` parameter.

- **match** (*str | Pattern[str]*) – A regex to match.

- **check** (*Callable[[BaseException], bool]*) – If specified, a callable that will be called with the exception as a parameter after checking the type and the match regex if specified. If it returns `True` it will be considered a match, if not it will be considered a failed match.

RaisesExc.matches() can also be used standalone to check individual exceptions.

Examples:

```
with RaisesGroup(RaisesExc(ValueError, match="string"))
    ...
with RaisesGroup(RaisesExc(check=lambda x: x.args == (3, "hello"))):
    ...
with RaisesGroup(RaisesExc(check=lambda x: type(x) is ValueError)):
    ...
```

fail_reason

Set after a call to *matches()* to give a human-readable reason for why the match failed. When used as a context manager the string will be printed as the reason for the test failing.

matches(*exception*)

Check if an exception matches the requirements of this *RaisesExc*. If it fails, *RaisesExc.fail_reason* will be set.

Examples:

```
assert RaisesExc(ValueError).matches(my_exception):
# is equivalent to
assert isinstance(my_exception, ValueError)

# this can be useful when checking e.g. the ``__cause__`` of an exception.
with pytest.raises(ValueError) as excinfo:
    ...
assert RaisesExc(SyntaxError, match="foo").matches(excinfo.value.__cause__)
```

(continues on next page)

(continued from previous page)

```
# above line is equivalent to
assert isinstance(excinfo.value.__cause__, SyntaxError)
assert re.search("foo", str(excinfo.value.__cause__)
```

RaisesGroup

Tutorial: *Assertions about expected exception groups*

final class RaisesGroup

Added in version 8.4.

Contextmanager for checking for an expected `ExceptionGroup`. This works similar to *`pytest.raises()`*, but allows for specifying the structure of an `ExceptionGroup`. *`ExceptionInfo.group_contains()`* also tries to handle exception groups, but it is very bad at checking that you *didn't* get unexpected exceptions.

The catching behaviour differs from except*, being much stricter about the structure by default. By using `allow_unwrapped=True` and `flatten_subgroups=True` you can match except* fully when expecting a single exception.

Parameters

- **args** – Any number of exception types, *`RaisesGroup`* or *`RaisesExc`* to specify the exceptions contained in this exception. All specified exceptions must be present in the raised group, *and no others*.

  If you expect a variable number of exceptions you need to use *`pytest.raises(ExceptionGroup)`* and manually check the contained exceptions. Consider making use of *`RaisesExc.matches()`*.

  It does not care about the order of the exceptions, so `RaisesGroup(ValueError, TypeError)` is equivalent to `RaisesGroup(TypeError, ValueError)`.

- **match** (*`str | re.Pattern[str] | None`*) – If specified, a string containing a regular expression, or a regular expression object, that is tested against the string representation of the exception group and its **PEP 678** `__notes__` using `re.search()`.

  To match a literal string that may contain special characters, the pattern can first be escaped with `re.escape()`.

  Note that " (5 subgroups)" will be stripped from the `repr` before matching.

- **check** (*`Callable[[E], bool]`*) – If specified, a callable that will be called with the group as a parameter after successfully matching the expected exceptions. If it returns `True` it will be considered a match, if not it will be considered a failed match.

- **allow_unwrapped** (*`bool`*) – If expecting a single exception or *`RaisesExc`* it will match even if the exception is not inside an exceptiongroup.

  Using this together with `match`, `check` or expecting multiple exceptions will raise an error.

- **flatten_subgroups** (*`bool`*) – "flatten" any groups inside the raised exception group, extracting all exceptions inside any nested groups, before matching. Without this it expects you to fully specify the nesting structure by passing *`RaisesGroup`* as expected parameter.

Examples:

```
with RaisesGroup(ValueError):
    raise ExceptionGroup("", (ValueError(),))
# match
with RaisesGroup(
    ValueError,
    ValueError,
    RaisesExc(TypeError, match="^expected int$"),
    match="^my group$",
):
    raise ExceptionGroup(
        "my group",
        [
            ValueError(),
            TypeError("expected int"),
            ValueError(),
        ],
    )
# check
with RaisesGroup(
    KeyboardInterrupt,
    match="^hello$",
    check=lambda x: isinstance(x.__cause__, ValueError),
):
    raise BaseExceptionGroup("hello", [KeyboardInterrupt()]) from ValueError
# nested groups
with RaisesGroup(RaisesGroup(ValueError)):
    raise ExceptionGroup("", (ExceptionGroup("", (ValueError(),)),))

# flatten_subgroups
with RaisesGroup(ValueError, flatten_subgroups=True):
    raise ExceptionGroup("", (ExceptionGroup("", (ValueError(),)),))

# allow_unwrapped
with RaisesGroup(ValueError, allow_unwrapped=True):
    raise ValueError
```

RaisesGroup.matches() can also be used directly to check a standalone exception group.

The matching algorithm is greedy, which means cases such as this may fail:

```
with RaisesGroup(ValueError, RaisesExc(ValueError, match="hello")):
    raise ExceptionGroup("", (ValueError("hello"), ValueError("goodbye")))
```

even though it generally does not care about the order of the exceptions in the group. To avoid the above you should specify the first `ValueError` with a *RaisesExc* as well.

Note

When raised exceptions don't match the expected ones, you'll get a detailed error message explaining why. This includes `repr(check)` if set, which in Python can be overly verbose, showing memory locations etc etc.

If installed and imported (in e.g. `conftest.py`), the `hypothesis` library will monkeypatch this output to provide shorter & more readable repr's.

fail_reason

Set after a call to *matches()* to give a human-readable reason for why the match failed. When used as a context manager the string will be printed as the reason for the test failing.

matches(*exception: BaseException | None*) → TypeGuard[ExceptionGroup[ExcT_1]]

matches(*exception: BaseException | None*) → TypeGuard[BaseExceptionGroup[BaseExcT_1]]

Check if an exception matches the requirements of this RaisesGroup. If it fails, `RaisesGroup.fail_reason` will be set.

Example:

```
with pytest.raises(TypeError) as excinfo:
    ...
assert RaisesGroup(ValueError).matches(excinfo.value.__cause__)
# the above line is equivalent to
myexc = excinfo.value.__cause
assert isinstance(myexc, BaseExceptionGroup)
assert len(myexc.exceptions) == 1
assert isinstance(myexc.exceptions[0], ValueError)
```

TerminalReporter

final class TerminalReporter(*config*, *file=None*)

wrap_write(*content*, *, *flush=False*, *margin=8*, *line_sep='\n'*, ***markup*)

Wrap message with margin for progress info.

rewrite(*line*, ***markup*)

Rewinds the terminal cursor to the beginning and writes the given line.

Parameters

erase – If True, will also add spaces until the full terminal width to ensure previous lines are properly erased.

The rest of the keyword arguments are markup instructions.

build_summary_stats_line()

Build the parts used in the last summary stats line.

The summary stats line is the line shown at the end, "=== 12 passed, 2 errors in Xs===".

This function builds a list of the "parts" that make up for the text in that line, in the example above it would be:

```
[
    ("12 passed", {"green": True}),
    ("2 errors", {"red": True}
]
```

That last dict for each line is a "markup dictionary", used by TerminalWriter to color output.

The final color of the line is also determined by this function, and is the second element of the returned tuple.

TestReport

class TestReport

Bases: BaseReport

Basic test report object (also used for setup and teardown calls if they fail).

Reports can contain arbitrary extra attributes.

nodeid: str

Normalized collection nodeid.

location: tuple[str, int | None, 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. The filesystempath may be relative to config.rootdir. The line number is 0-based.

keywords: Mapping[str, Any]

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: None | *ExceptionInfo*[BaseException] | tuple[str, int, str] | str | TerminalRepr

None or a failure representation.

when: Literal['setup', 'call', 'teardown']

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: float

Time it took to run just the test.

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.

classmethod from_item_and_call(*item*, *call*)

Create and fill a TestReport with standard item and call info.

Parameters

- **item** (Item) – The item.
- **call** (CallInfo[None]) – The call info.

property caplog: str

Return captured log lines, if log capturing is enabled.

Added in version 3.5.

property capstderr: str

Return captured text from stderr, if capturing is enabled.

Added in version 3.0.

property capstdout: str

Return captured text from stdout, if capturing is enabled.

Added 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 failed: bool

Whether the outcome is failed.

property fspath: str

The path portion of the reported node, as a string.

property head_line: str | None

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`.

Added in version 3.0.

property passed: bool

Whether the outcome is passed.

property skipped: bool

Whether the outcome is skipped.

TestShortLogReport

class TestShortLogReport

Used to store the test status result category, shortletter and verbose word. For example `"rerun", "R", ("RERUN", {"yellow": True})`.

Variables

- **category** – The class of result, for example “`passed`”, “`skipped`”, “`error`”, or the empty string.
- **letter** – The short letter shown as testing progresses, for example `"."`, `"s"`, `"E"`, or the empty string.
- **word** – Verbose word is shown as testing progresses in verbose mode, for example `"PASSED"`, `"SKIPPED"`, `"ERROR"`, or the empty string.

category: str

Alias for field number 0

letter: str

Alias for field number 1

word: str | tuple[str, Mapping[str, bool]]

Alias for field number 2

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 *`StashKey`*s 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]
```

Added in version 7.0.

__setitem__(*key*, *value*)
Set a value for key.

__getitem__(*key*)
Get the value for key.

Raises `KeyError` if the key wasn't set before.

get(*key*, *default*)
Get the value for key, or return default if the key wasn't set before.

setdefault(*key*, *default*)
Return the value of key if already set, otherwise set the value of key to default and return default.

__delitem__(*key*)
Delete the value for key.

Raises `KeyError` if the key wasn't set before.

__contains__(*key*)
Return whether key was set.

__len__()
Return how many items exist in the stash.

class StashKey
Bases: `Generic[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.

Added in version 7.0.

3.4.8 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]
```

3.4.9 Environment Variables

Environment variables that can be used to change pytest's behavior.

CI

When set to a non-empty value, pytest acknowledges that it is running in a CI process. See also ci-pipelines.

BUILD_NUMBER

When set to a non-empty value, pytest acknowledges that it is running in a CI process. Alternative to *CI*. See also ci-pipelines.

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_VERSION

This environment variable is defined at the start of the pytest session and is undefined afterwards. It contains the value of `pytest.__version__`, and among other things can be used to easily check if a code is running from within a pytest run.

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_DEBUG_TEMPROOT

Root for temporary directories produced by fixtures like `tmp_path` as discussed in *Temporary directory location and retention*.

PYTEST_DISABLE_PLUGIN_AUTOLOAD

When set, disables plugin auto-loading through entry point packaging metadata. Only plugins explicitly specified in `PYTEST_PLUGINS` or with `-p` will be loaded. See also *–disable-plugin-autoload*.

PYTEST_PLUGINS

Contains comma-separated list of modules that should be loaded as plugins:

```
export PYTEST_PLUGINS=mymodule.plugin,xdist
```

See also `-p`.

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 to a non-empty string (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 to a non-empty string (regardless of value), pytest will use color in terminal output. `PY_COLORS` and `NO_COLOR` take precedence over `FORCE_COLOR`.

3.4.10 Exceptions

exception UsageError

Bases: `Exception`

Error in pytest usage or invocation.

final exception FixtureLookupError

Bases: `LookupError`

Could not return a requested fixture (missing or invalid).

3.4.11 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: *PytestWarning*

Warning emitted by the pytest assert rewrite module.

class PytestCacheWarning

Bases: *PytestWarning*

Warning emitted by the cache plugin in various situations.

class PytestCollectionWarning

Bases: *PytestWarning*

Warning emitted when pytest is not able to collect a file or symbol in a module.

class PytestConfigWarning

Bases: *PytestWarning*

Warning emitted for configuration issues.

class PytestDeprecationWarning

Bases: *PytestWarning*, DeprecationWarning

Warning class for features that will be removed in a future version.

class PytestExperimentalApiWarning

Bases: *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 PytestReturnNotNoneWarning

Bases: *PytestWarning*

Warning emitted when a test function returns a value other than None.

See *Returning non-None value in test functions* for details.

class PytestRemovedIn10Warning

Bases: *PytestDeprecationWarning*

Warning class for features that will be removed in pytest 10.

class PytestUnknownMarkWarning

Bases: *PytestWarning*

Warning emitted on use of unknown markers.

See *How to mark test functions with attributes* for details.

class PytestUnraisableExceptionWarning

Bases: *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: *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.

3.4.12 Configuration Options

Here is a list of builtin configuration options that may be written in a `pytest.ini` (or `.pytest.ini`), `pyproject.toml`, `tox.ini`, or `setup.cfg` file, usually located at the root of your repository.

To see each file format in detail, 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 `pytest.toml` 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

Type
`list[str]`

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 configuration file content:

```
# content of pytest.toml
[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
```

cache_dir

Type
`str`

Default
`".pytest_cache"`

Sets the directory where the cache plugin's content is stored. Directory may be relative or absolute path. If setting relative path, then directory is created relative to *rootdir*. Additionally, a 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*.

collect_imported_tests

Type
bool

Default
true

Added in version 8.4.

Setting this to `false` will make pytest collect classes/functions from test files **only** if they are defined in that file (as opposed to imported there).

```
[pytest]
collect_imported_tests = false
```

```
[pytest]
collect_imported_tests = false
```

pytest traditionally collects classes/functions in the test module namespace even if they are imported from another file.

For example:

```
# contents of src/domain.py
class Testament: ...


# contents of tests/test_testament.py
from domain import Testament


def test_testament(): ...
```

In this scenario, with the default options, pytest will collect the class `Testament` from `tests/test_testament.py` because it starts with `Test`, even though in this case it is a production class being imported in the test module namespace.

Set `collected_imported_tests` to `false` in the configuration file prevents that.

consider_namespace_packages

Type
bool

Default
false

Controls if pytest should attempt to identify namespace packages when collecting Python modules.

Set to `True` if the package you are testing is part of a namespace package. Namespace packages are also supported as *`--pyargs`* target.

Only native namespace packages are supported, with no plans to support legacy namespace packages.

For best results when using `consider_namespace_packages`, pytest needs to be able to import your namespace packages. This is best achieved by installing the packages in your environment, most commonly in "editable" mode. If you can't install the packages, consider adding the namespace root paths to *pythonpath*.

Added in version 8.1.

console_output_style

Type
`str`

Default
`"progress"`

Sets the console output style while running tests:

- `classic`: classic pytest output.
- `progress`: like classic pytest output, but with a progress indicator.
- `progress-even-when-capture-no`: allows the use of the progress indicator even when `capture=no`.
- `count`: like progress, but shows progress as the number of tests completed instead of a percent.
- `times`: show tests duration.

You can fallback to `classic` if you prefer or the new mode is causing unexpected problems:

```
[pytest]
console_output_style = "classic"
```

```
[pytest]
console_output_style = classic
```

disable_test_id_escaping_and_forfeit_all_rights_to_community_support

Type
`bool`

Default
`false`

Added in version 4.4.

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 configuration file:

```
[pytest]
disable_test_id_escaping_and_forfeit_all_rights_to_community_support = true
```

```
[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.

See *@pytest.mark.parametrize: parametrizing test functions*.

doctest_encoding

Type
str

Default
"utf-8"

Default encoding to use to decode text files with docstrings. *See how pytest handles doctests*.

doctest_optionflags

Type
list[str]

One or more doctest flag names from the standard `doctest` module. *See how pytest handles doctests*.

empty_parameter_set_mark

Type
str

Default
"skip"

Allows to pick the action for empty parametersets in parameterization

- `skip` skips tests with an empty parameterset
- `xfail` marks tests with an empty parameterset as xfail(run=False)
- `fail_at_collect` raises an exception if parametrize collects an empty parameter set

```
[pytest]
empty_parameter_set_mark = "xfail"
```

```
[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.

enable_assertion_pass_hook

Type
bool

Default
false

Enables the *pytest_assertion_pass* hook. Make sure to delete any previously generated `.pyc` cache files.

```
[pytest]
enable_assertion_pass_hook = true
```

```
[pytest]
enable_assertion_pass_hook = true
```

faulthandler_exit_on_timeout

Type
bool

Default
false

Exit the pytest process after the per-test timeout is reached by passing `exit=True` to the `faulthandler.dump_traceback_later()` function. This is particularly useful to avoid wasting CI resources for test suites that are prone to putting the main Python interpreter into a deadlock state.

```
[pytest]
faulthandler_timeout = 5
faulthandler_exit_on_timeout = true
```

```
[pytest]
faulthandler_timeout = 5
faulthandler_exit_on_timeout = true
```

faulthandler_timeout

Type
float

Default
`0` (disabled)

Dumps the tracebacks of all threads if a test takes longer than `X` seconds to run (including fixture setup and teardown). Implemented using the `faulthandler.dump_traceback_later()` function, so all caveats there apply.

```
[pytest]
faulthandler_timeout = 5
```

```
[pytest]
faulthandler_timeout = 5
```

For more information please refer to faulthandler.

filterwarnings

Type
list[str]

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.

```
[pytest]
filterwarnings = [
    'error',
    'ignore::DeprecationWarning',
    # Note the use of single quote below to denote "raw" strings in TOML.
    'ignore:function ham\(\) should not be used:UserWarning',
]
```

```
[pytest]
filterwarnings =
    error
    ignore::DeprecationWarning
    ignore:function ham\(\) should not be used:UserWarning
```

This tells pytest to ignore deprecation warnings and turn all other warnings into errors. For more information please refer to *How to capture warnings*.

max_warnings

Type
int

Added in version 9.1.

Maximum number of warnings allowed before the test run is considered a failure. When all tests pass, but the total number of warnings exceeds this value, pytest exits with *pytest.ExitCode* MAX_WARNINGS_ERROR (code 6).

```
[pytest]
max_warnings = 10
```

```
[pytest]
max_warnings = 10
```

Note that *filtered warnings* do not count toward this maximum total.

Can also be set via the *--max-warnings* command-line option.

junit_duration_report

Type
str

Default
"total"

Added in version 4.1.

Configures how durations are recorded into the JUnit XML report:

- total: 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"
```

```
[pytest]
junit_duration_report = call
```

junit_family

Type
str

Default
"xunit2"

Added 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.

```
[pytest]
junit_family = "xunit2"
```

```
[pytest]
junit_family = xunit2
```

junit_log_passing_tests

Type
`bool`

Default
`true`

Added in version 4.6.

If `junit_logging != "no"`, configures if the captured output should be written to the JUnit XML file for **passing** tests.

```
[pytest]
junit_log_passing_tests = false
```

```
[pytest]
junit_log_passing_tests = False
```

junit_logging

Type
`str`

Default
`"no"`

Added 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`: no captured output is written.

```
[pytest]
junit_logging = "system-out"
```

```
[pytest]
junit_logging = system-out
```

junit_suite_name

Type
str

Default
"pytest"

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"
```

```
[pytest]
junit_suite_name = my_suite
```

log_auto_indent

Type
str

Default
"false"

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
- "[positive integer]" - auto-indent multiline log messages by [value] spaces

```
[pytest]
log_auto_indent = "false"
```

```
[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

Type
bool

Default
false

Enable log display during test run (also known as *"live logging"*).

```
[pytest]
log_cli = true
```

```
[pytest]
log_cli = true
```

log_cli_date_format

Type
str

Default
Fallback to `log_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"
```

```
[pytest]
log_cli_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *Live Logs*.

log_cli_format

Type
str

Default
Fallback to `log_format`

Sets a `logging`-compatible string used to format live logging messages.

```
[pytest]
log_cli_format = "%(asctime)s %(levelname)s %(message)s"
```

```
[pytest]
log_cli_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *Live Logs*.

log_cli_level

Type
str

Default
Fallback to `log_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. Note in TOML the integer must be quoted, as there is no support for config parameters of mixed type.

```
[pytest]
log_cli_level = "INFO"
log_cli_level = "10"
```

```
[pytest]
log_cli_level = INFO
log_cli_level = 10
```

For more information, see *Live Logs*.

log_date_format

Type
str

Default
"%H:%M:%S"

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"
```

```
[pytest]
log_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *How to manage logging*.

log_file

Type
str

Sets a file name relative to the current working directory where log messages should be written to, in addition to the other logging facilities that are active.

```
[pytest]
log_file = "logs/pytest-logs.txt"
```

```
[pytest]
log_file = logs/pytest-logs.txt
```

For more information, see *How to manage logging*.

log_file_date_format

Type
str

Default
Fallback to `log_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"
```

```
[pytest]
log_file_date_format = %Y-%m-%d %H:%M:%S
```

For more information, see *How to manage logging*.

log_file_format

Type
str

Default
Fallback to `log_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"
```

```
[pytest]
log_file_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *How to manage logging*.

log_file_level

Type
str

Default
Fallback to `log_level`

Sets the minimum log message level that should be captured for the logging file. The integer value (in TOML, as a string) or the names of the levels can be used.

```
[pytest]
log_file_level = "INFO"
log_cli_level = "10"
```

```
[pytest]
log_file_level = INFO
log_cli_level = 10
```

For more information, see *How to manage logging*.

log_file_mode

Type
str

Default
`"w"`

Sets the mode that the logging file is opened with. The options are `"w"` to recreate the file or `"a"` to append to the file.

```
[pytest]
log_file_mode = "a"
```

```
[pytest]
log_file_mode = a
```

For more information, see *How to manage logging*.

log_format

Type
str

Default
%(levelname)-8s %(name)s:%(filename)s:%(lineno)d %(message)s

Sets a logging-compatible string used to format captured logging messages.

```
[pytest]
log_format = "%(asctime)s %(levelname)s %(message)s"
```

```
[pytest]
log_format = %(asctime)s %(levelname)s %(message)s
```

For more information, see *How to manage logging*.

log_level

Type
str

Sets the minimum log message level that should be captured for logging capture. Not set by default, so it depends on the root/parent log handler's effective level, where it is "WARNING" by default. The integer value (in TOML, as a string) or the names of the levels can be used.

```
[pytest]
log_level = "INFO"
log_cli_level = "10"
```

```
[pytest]
log_level = INFO
log_cli_level = 10
```

For more information, see *How to manage logging*.

markers

Type
list[str]

When the *strict_markers* configuration option is set, 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 set *strict_markers* to true to avoid future regressions:

```
[pytest]
addopts = ["--strict-markers"]
markers = ["slow", "serial"]
```

```
[pytest]
strict_markers = true
markers =
    slow
    serial
```

minversion

Type
str

Specifies a minimal pytest version required for running tests.

```
[pytest]
minversion = 3.0  # will fail if we run with pytest-2.8
```


```
[pytest]
minversion = 3.0  # will fail if we run with pytest-2.8
```


norecursedirs

Type
list[str]

Default
["*.egg", ".*", "_darcs", "build", "CVS", "dist", "node_modules", "venv", "{arch}"]

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
```

Setting a norecursedirs replaces the default. Here is an example of how to avoid certain directories:

```
[pytest]
norecursedirs = [".svn", "_build", "tmp*"]
```

```
[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. 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

Type
list[str]

Default
["Test"]

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"]
```

```
[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

Type
`list[str]`

Default
`["test_*.py", "*_test.py"]`

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"]
```

```
[pytest]
python_files = test_*.py check_*.py example_*.py
```

Or one per line:

```
[pytest]
python_files =
    test_*.py
    check_*.py
    example_*.py
```

python_functions

Type
`list[str]`

Default
`["test"]`

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"]
```

```
[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

Type
list[str]

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"]
```

```
[pytest]
pythonpath = src1 src2
```

required_plugins

Type
list[str]

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"]
```

```
[pytest]
required_plugins = pytest-django>=3.0.0,<4.0.0 pytest-html pytest-xdist>=1.0.0
```

strict

Type
bool

Default
false

If set to `true`, enable "strict mode", which enables the following options:

- *strict_config*
- *strict_markers*
- *strict_parametrization_ids*
- *strict_xfail*

Plugins may also enable their own strictness options.

If you explicitly set an individual strictness option, it takes precedence over `strict`.

> **Note**
>
> If pytest adds new strictness options in the future, they will also be enabled in strict mode. Therefore, you should only enable strict mode if you use a pinned/locked version of pytest, or if you want to proactively adopt new strictness options as they are added.

```
[pytest]
strict = true
```

```
[pytest]
strict = true
```

Added in version 9.0.

strict_config

Type
bool

Default
false

If set to `true`, any warnings encountered while parsing the `pytest` section of the configuration file will raise errors.

```
[pytest]
strict_config = true
```

```
[pytest]
strict_config = true
```

You can also enable this option via the *strict* option.

strict_markers

Type
bool

Default
false

If set to `true`, markers not registered in the `markers` section of the configuration file will raise errors.

```
[pytest]
strict_markers = true
```

```
[pytest]
strict_markers = true
```

You can also enable this option via the *strict* option.

strict_parametrization_ids

Type
bool

Default

`false`

If set to `true`, pytest emits an error if it detects non-unique parameter set IDs.

If not set, pytest automatically handles this by adding `0`, `1`, … to duplicate IDs, making them unique.

```
[pytest]
strict_parametrization_ids = true
```

```
[pytest]
strict_parametrization_ids = true
```

You can also enable this option via the *`strict`* option.

For example,

```
import pytest


@pytest.mark.parametrize("letter", ["a", "a"])
def test_letter_is_ascii(letter):
    assert letter.isascii()
```

will emit an error because both cases (parameter sets) have the same auto-generated ID “a”.

To fix the error, if you decide to keep the duplicates, explicitly assign unique IDs:

```
import pytest


@pytest.mark.parametrize("letter", ["a", "a"], ids=["a0", "a1"])
def test_letter_is_ascii(letter):
    assert letter.isascii()
```

See *`parametrize`* and *`pytest.param()`* for other ways to set IDs.

`strict_xfail`

Type

`bool`

Default

`false`

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]
strict_xfail = true
```

```
[pytest]
strict_xfail = true
```

You can also enable this option via the *`strict`* option.

Changed in version 9.0: Renamed from `xfail_strict` to `strict_xfail`. `xfail_strict` is accepted as an alias for `strict_xfail`.

testpaths

Type
list[str]

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. File system paths may use shell-style wildcards, including the recursive `**` pattern.

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"]
```

```
[pytest]
testpaths = testing doc
```

This configuration means that executing:

```
pytest
```

has the same practical effects as executing:

```
pytest testing doc
```

tmp_path_retention_count

Type
str

Default
"3"

How many sessions should pytest keep the `tmp_path` directories, according to *tmp_path_retention_policy*.

```
[pytest]
tmp_path_retention_count = "3"
```

```
[pytest]
tmp_path_retention_count = 3
```

tmp_path_retention_policy

Type
str

Default
"all"

Controls which directories created by the `tmp_path` fixture are kept around, based on test outcome.

- `all`: retains directories for all tests, regardless of the outcome.
- `failed`: retains directories only for tests with outcome `error` or `failed`.
- `none`: directories are always removed after each test ends, regardless of the outcome.

```
[pytest]
tmp_path_retention_policy = "all"
```

```
[pytest]
tmp_path_retention_policy = all
```

truncation_limit_chars

Type
int

Default
640

Controls maximum number of characters to truncate assertion message contents.

Setting value to `0` disables the character limit for truncation.

```
[pytest]
truncation_limit_chars = 640
```

```
[pytest]
truncation_limit_chars = 640
```

pytest truncates the assert messages to a certain limit by default to prevent comparison with large data to overload the console output.

Note

If pytest detects it is running on CI, truncation is disabled automatically.

truncation_limit_lines

Type
int

Default
8

Controls maximum number of lines to truncate assertion message contents.

Setting value to `0` disables the lines limit for truncation.

```
[pytest]
truncation_limit_lines = 8
```

```
[pytest]
truncation_limit_lines = 8
```

pytest truncates the assert messages to a certain limit by default to prevent comparison with large data to overload the console output.

> **Note**
>
> If pytest detects it is running on CI, truncation is disabled automatically.

usefixtures

Type
`list[str]`

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"]
```

```
[pytest]
usefixtures =
    clean_db
```

verbosity_assertions

Type
`str`

Default
`"auto"`

Set a verbosity level specifically for assertion related output, overriding the application wide level.

```
[pytest]
verbosity_assertions = "2"
```

```
[pytest]
verbosity_assertions = 2
```

A special value of `"auto"` can be used to explicitly use the global verbosity level.

verbosity_subtests

Type
`str`

Default
`"auto"`

Set the verbosity level specifically for **passed** subtests.

```
[pytest]
verbosity_subtests = "1"
```

```
[pytest]
verbosity_subtests = 1
```

A value of `1` or higher will show output for **passed** subtests (**failed** subtests are always reported). Passed subtests output can be suppressed with the value `0`, which overwrites the `-v` command-line option.

A special value of `"auto"` can be used to explicitly use the global verbosity level.

See also: subtests.

verbosity_test_cases

Type
`str`

Default
`"auto"`

Set a verbosity level specifically for test case execution related output, overriding the application wide level.

```
[pytest]
verbosity_test_cases = "2"
```

```
[pytest]
verbosity_test_cases = 2
```

A special value of `"auto"` can be used to explicitly use the global verbosity level.

3.4.13 Command-line Flags

This section documents all command-line options provided by pytest's core plugins.

> **Note**
>
> External plugins can add their own command-line options. This reference documents only the options from pytest's core plugins. To see all available options including those from installed plugins, run `pytest --help`.

Test Selection

-k EXPRESSION

Only run tests which match the given substring expression. An expression is a Python evaluable expression where all names are substring-matched against test names and their parent classes.

Examples:

```
pytest -k "test_method or test_other"  # matches names containing 'test_method'␣
↪OR 'test_other'
pytest -k "not test_method"            # matches names NOT containing 'test_method
↪'
pytest -k "not test_method and not test_other"  # excludes both
```

The matching is case-insensitive. Keywords are also matched to classes and functions containing extra names in their `extra_keyword_matches` set.

See *Specifying which tests to run* for more information and examples.

-m MARKEXPR

Only run tests matching given mark expression. Supports `and`, `or`, and `not` operators.

Examples:

```
pytest -m slow                  # run tests marked with @pytest.mark.slow
pytest -m "not slow"            # run tests NOT marked slow
pytest -m "mark1 and not mark2" # run tests marked mark1 but not mark2
```

See *How to mark test functions with attributes* for more information on markers.

--markers

Show all available markers (builtin, plugin, and per-project markers defined in configuration).

Test Execution Control

-x, --exitfirst

Exit instantly on first error or failed test.

--maxfail=NUM

Exit after first `num` failures or errors. Useful for CI environments where you want to fail fast but see a few failures.

--last-failed, --lf

Rerun only the tests that failed at the last run. If no tests failed (or no cached data exists), all tests are run. See also `cache_dir` and *How to re-run failed tests and maintain state between test runs*.

--failed-first, --ff

Run all tests, but run the last failures first. This may re-order tests and thus lead to repeated fixture setup/teardown.

--new-first, --nf

Run tests from new files first, then the rest of the tests sorted by file modification time.

--stepwise, --sw

Exit on test failure and continue from last failing test next time. Useful for fixing multiple test failures one at a time.

See *Stepwise* for more information.

--stepwise-skip, --sw-skip

Ignore the first failing test but stop on the next failing test. Implicitly enables `--stepwise`.

--stepwise-reset, --sw-reset

Resets stepwise state, restarting the stepwise workflow. Implicitly enables `--stepwise`.

--last-failed-no-failures, --lfnf

With `--last-failed`, determines whether to execute tests when there are no previously known failures or when no cached `lastfailed` data was found.

- `all` (default): runs the full test suite again
- `none`: just emits a message about no known failures and exits successfully

--runxfail

Report the results of xfail tests as if they were not marked. Useful for debugging xfailed tests. See *XFail: mark test functions as expected to fail*.

Collection

--collect-only, --co

Only collect tests, don't execute them. Shows which tests would be collected and run.

--pyargs

Try to interpret all arguments as Python packages. Useful for running tests of installed packages:

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

--ignore=PATH

Ignore path during collection (multi-allowed). Can be specified multiple times.

--ignore-glob=PATTERN

Ignore path pattern during collection (multi-allowed). Supports glob patterns.

--deselect=NODEID_PREFIX

Deselect item (via node id prefix) during collection (multi-allowed).

--confcutdir=DIR

Only load `conftest.py` files relative to specified directory.

--noconftest

Don't load any `conftest.py` files.

--keep-duplicates

Keep duplicate tests. By default, pytest removes duplicate test items.

--collect-in-virtualenv

Don't ignore tests in a local virtualenv directory. By default, pytest skips tests in virtualenv directories.

--continue-on-collection-errors

Force test execution even if collection errors occur.

--import-mode

Prepend/append to sys.path when importing test modules and conftest files.

- `prepend` (default): prepend to sys.path
- `append`: append to sys.path
- `importlib`: use importlib to import test modules

See *pytest import mechanisms and sys.path/PYTHONPATH* for more information.

Fixtures

--fixtures, --funcargs

Show available fixtures, sorted by plugin appearance. Fixtures with leading _ are only shown with `--verbose`.

--fixtures-per-test

Show fixtures per test.

--setup-only

Only setup fixtures, do not execute tests. See *How to use fixtures*.

--setup-show

Show setup of fixtures while executing tests.

--setup-plan

Show what fixtures and tests would be executed but don't execute anything.

Debugging

--pdb

Start the interactive Python debugger on errors or KeyboardInterrupt. See pdb-option.

--pdbcls=MODULENAME:CLASSNAME

Specify a custom interactive Python debugger for use with *--pdb*.

Example:

```
pytest --pdbcls=IPython.terminal.debugger:TerminalPdb
```

--trace

Immediately break when running each test.

See trace-option for more information.

--full-trace

Don't cut any tracebacks (default is to cut).

See how-to-modifying-python-tb-printing for more information.

--debug, --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. Default file name if not specified: `pytestdebug.log`.

--trace-config

Trace considerations of conftest.py files.

Output and Reporting

-v, --verbose

Increase verbosity. Can be specified multiple times (e.g., `-vv`) for even more verbose output.

See pytest.fine_grained_verbosity for fine-grained control over verbosity.

-q, --quiet

Decrease verbosity.

--verbosity=NUM

Set verbosity level explicitly. Default: 0.

-r CHARS, **--report-chars**=CHARS

Show extra test summary info as specified by chars:

- `f`: failed
- `E`: error
- `s`: skipped
- `x`: xfailed
- `X`: xpassed
- `p`: passed
- `P`: passed with output
- `a`: all except passed (p/P)
- `A`: all
- `w`: warnings (enabled by default)
- `N`: resets the list

Default: `'fE'`

Examples:

```
pytest -rA          # show all outcomes
pytest -rfE         # show only failed and errors (default)
pytest -rfs         # show failed and skipped
```

See pytest.detailed_failed_tests_usage for more information.

--no-header

Disable header.

--no-summary

Disable summary.

--no-fold-skipped

Do not fold skipped tests in short summary.

--force-short-summary

Force condensed summary output regardless of verbosity level.

-l, --showlocals

Show locals in tracebacks (disabled by default).

--no-showlocals

Hide locals in tracebacks (negate *--showlocals* passed through addopts).

--tb=STYLE

Traceback print mode:

- `auto`: intelligent traceback formatting (default)
- `long`: exhaustive, informative traceback formatting
- `short`: shorter traceback format
- `line`: only the failing line
- `native`: Python's standard traceback
- `no`: no traceback

See how-to-modifying-python-tb-printing for examples.

--xfail-tb

Show tracebacks for xfail (as long as *--tb* != `no`).

--show-capture

Controls how captured stdout/stderr/log is shown on failed tests.

- `no`: don't show captured output
- `stdout`: show captured stdout
- `stderr`: show captured stderr
- `log`: show captured logging
- `all` (default): show all captured output

--color=WHEN

Color terminal output:

- `yes`: always use color
- `no`: never use color
- `auto` (default): use color if terminal supports it

--code-highlight={yes,no}

Whether code should be highlighted (only if *--color* is also enabled). Default: `yes`.

--pastebin=MODE

Send failed|all info to bpaste.net pastebin service.

--durations=NUM

Show N slowest setup/test durations (N=0 for all). See *Profiling test execution duration*.

--durations-min=NUM

Minimal duration in seconds for inclusion in slowest list. Default: 0.005 (or 0.0 if `-vv` is given).

Output Capture

--capture=METHOD

Per-test capturing method:

- `fd`: capture at file descriptor level (default)
- `sys`: capture at sys level
- `no`: don't capture output
- `tee-sys`: capture but also show output on terminal

See *How to capture stdout/stderr output*.

-s

Shortcut for *--capture=no*.

JUnit XML

--junit-xml=PATH, **--junitxml**=PATH

Create junit-xml style report file at given path.

--junit-prefix=STR, **--junitprefix**=STR

Prepend prefix to classnames in junit-xml output.

Cache

--cache-show[=PATTERN]

Show cache contents, don't perform collection or tests. Default glob pattern: `'*'`.

--cache-clear

Remove all cache contents at start of test run. See *How to re-run failed tests and maintain state between test runs*.

Warnings

--disable-pytest-warnings, --disable-warnings

Disable warnings summary.

-W WARNING, **--pythonwarnings**=WARNING

Set which warnings to report, see -W option of Python itself. Can be specified multiple times.

--max-warnings=NUM

Exit with *pytest.ExitCode* MAX_WARNINGS_ERROR (code 6) if all the tests pass, but the number of warnings exceeds the given threshold. By default there is no limit. Can also be set via the *max_warnings* configuration option.

Doctest

--doctest-modules

Run doctests in all .py modules.

See *How to run doctests* for more information on using doctests with pytest.

--doctest-report

Choose another output format for diffs on doctest failure:

- none
- cdiff
- ndiff
- udiff
- only_first_failure

--doctest-glob=PATTERN

Doctests file matching pattern. Default: test*.txt.

--doctest-ignore-import-errors

Ignore doctest collection errors.

--doctest-continue-on-failure

For a given doctest, continue to run after the first failure.

Configuration

-c FILE, **--config-file**=FILE

Load configuration from FILE instead of trying to locate one of the implicit configuration files.

--rootdir=ROOTDIR

Define root directory for tests. Can be relative path: 'root_dir', './root_dir', 'root_dir/another_dir/'; absolute path: '/home/user/root_dir'; path with variables: '$HOME/root_dir'.

--basetemp=DIR

Base temporary directory for this test run. Warning: this directory is removed if it exists.

See *Temporary directory location and retention* for more information.

-o OPTION=VALUE, **--override-ini**=OPTION=VALUE

Override configuration option with option=value style. Can be specified multiple times.

Example:

```
pytest -o strict_xfail=true -o cache_dir=cache
```

--strict-config

Enables the *strict_config* option.

--strict-markers

Enables the *strict_markers* option.

--strict

Enables the *strict* option (which enables all strictness options).

--assert=MODE

Control assertion debugging tools:

- plain: performs no assertion debugging
- rewrite (default): rewrites assert statements in test modules on import to provide assert expression information

Logging

See *How to manage logging* for a guide on using these flags.

--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=FORMAT

Log format used by the logging module.

--log-date-format=FORMAT

Log date format used by the logging module.

--log-cli-level=LEVEL

CLI logging level. See *Live Logs*.

--log-cli-format=FORMAT

Log format used by the logging module for CLI output.

--log-cli-date-format=FORMAT

Log date format used by the logging module for CLI output.

--log-file=PATH

Path to a file logging will be written to.

--log-file-mode

Log file open mode:

- w (default): recreate the file
- a: append to the file

--log-file-level=LEVEL

Log file logging level.

--log-file-format=FORMAT

Log format used by the logging module for the log file.

--log-file-date-format=FORMAT

Log date format used by the logging module for the log file.

--log-auto-indent=VALUE

Auto-indent multiline messages passed to the logging module. Accepts `true|on`, `false|off` or an integer.

--log-disable=LOGGER

Disable a logger by name. Can be passed multiple times.

Plugin and Extension Management

-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`. See also *--disable-plugin-autoload*.

--disable-plugin-autoload

Disable plugin auto-loading through entry point packaging metadata. Only plugins explicitly specified in *-p* or env var *PYTEST_PLUGINS* will be loaded.

Version and Help

-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.

Complete Help Output

All the command-line flags can also 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 evaluable
                        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
```

(continues on next page)

(continued from previous page)

```
                        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
  --maxfail=num         Exit after first num failures or errors
  --strict-config       Enables the strict_config option
  --strict-markers      Enables the strict_markers option
  --strict              Enables the strict option
  --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=modulename:classname
                        Specify a custom interactive Python debugger for use
                        with --pdb.For example:
                        --pdbcls=IPython.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, --last-failed-no-failures={all,none}
                        With ``--lf``, determines whether to execute tests
                        when there are no previously (known) failures or
                        when no cached ``lastfailed`` data was found.
                        ``all`` (the default) runs the full test suite
                        again. ``none`` just emits a message about no known
                        failures and exits successfully.
  --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.
  --sw-reset, --stepwise-reset
                        Resets stepwise state, restarting the stepwise
                        workflow. 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
```

(continues on next page)

(continued from previous page)

```
                        list. Default: 0.005 (or 0.0 if -vv is given).
  -v, --verbose         Increase verbosity
  --no-header           Disable header
  --no-summary          Disable summary
  --no-fold-skipped     Do not fold skipped tests in short summary.
  --force-short-summary
                        Force condensed summary output regardless of
                        verbosity level.
  -q, --quiet           Decrease verbosity
  --verbosity=VERBOSE   Set verbosity. Default: 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)
  --no-showlocals       Hide locals in tracebacks (negate --showlocals
                        passed through addopts)
  --tb=style            Traceback print mode
                        (auto/long/short/line/native/no)
  --xfail-tb            Show tracebacks for xfail (as long as --tb != no)
  --show-capture={no,stdout,stderr,log,all}
                        Controls how captured stdout/stderr/log is shown on
                        failed tests. Default: 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). Default: yes.
  --pastebin=mode       Send failed|all info to bpaste.net pastebin service
  --junitxml, --junit-xml=path
                        Create junit-xml style report file at given path
  --junitprefix, --junit-prefix=str
                        Prepend prefix to classnames in junit-xml output

pytest-warnings:
  -W, --pythonwarnings PYTHONWARNINGS
                        Set which warnings to report, see -W option of
                        Python itself
  --max-warnings=num    Exit with error if the number of warnings exceeds
                        this threshold

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
```

(continues on next page)

(continued from previous page)

```
                        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
  --continue-on-collection-errors
                        Force test execution even if collection errors occur
  --import-mode={prepend,append,importlib}
                        Prepend/append to sys.path when importing test
                        modules and conftest files. Default: 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 collection errors
  --doctest-continue-on-failure
                        For a given doctest, continue to run after the first
                        failure

test session debugging and configuration:
  -c, --config-file FILE
                        Load configuration from `FILE` instead of trying to
                        locate one of the implicit configuration files.
  --rootdir=ROOTDIR     Define root directory for tests. Can be relative
                        path: 'root_dir', './root_dir',
                        'root_dir/another_dir/'; absolute path:
                        '/home/user/root_dir'; path with variables:
                        '$HOME/root_dir'.
  --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`. See also
                        --disable-plugin-autoload.
  --disable-plugin-autoload
                        Disable plugin auto-loading through entry point
                        packaging metadata. Only plugins explicitly
                        specified in -p or env var PYTEST_PLUGINS will be
                        loaded.
  --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. Default: pytestdebug.log.
```

(continues on next page)

(continued from previous page)

```
  -o, --override-ini OVERRIDE_INI
                        Override configuration option with "option=value"
                        style, e.g. `-o strict_xfail=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
                        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 used by the logging module
  --log-date-format=LOG_DATE_FORMAT
                        Log date format used by the logging module
  --log-cli-level=LOG_CLI_LEVEL
                        CLI logging level
  --log-cli-format=LOG_CLI_FORMAT
                        Log format used by the logging module
  --log-cli-date-format=LOG_CLI_DATE_FORMAT
                        Log date format used by the logging module
  --log-file=LOG_FILE   Path to a file when logging will be written to
  --log-file-mode={w,a}
                        Log file open mode
  --log-file-level=LOG_FILE_LEVEL
                        Log file logging level
  --log-file-format=LOG_FILE_FORMAT
                        Log format used by the logging module
  --log-file-date-format=LOG_FILE_DATE_FORMAT
                        Log date format 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.
  --log-disable=LOGGER_DISABLE
                        Disable a logger by name. Can be passed multiple
                        times.

[pytest] configuration options in the first pytest.toml|pytest.ini|tox.ini|setup.
↪cfg|pyproject.toml file found:

  markers (linelist):   Register new markers for test functions
  empty_parameter_set_mark (string):
                        Default marker for empty parametersets
  strict_config (bool): Any warnings encountered while parsing the `pytest`
```

(continues on next page)

(continued from previous page)

```
                      section of the configuration file raise errors
strict_markers (bool):
                      Markers not registered in the `markers` section of
                      the configuration file raise errors
strict (bool):        Enables all strictness options, currently:
                      strict_config, strict_markers, strict_xfail,
                      strict_parametrization_ids
filterwarnings (linelist):
                      Each line specifies a pattern for
                      warnings.filterwarnings. Processed after
                      -W/--pythonwarnings.
max_warnings (string):
                      Maximum number of warnings allowed before failing
                      the test run
norecursedirs (args): Directory patterns to avoid for recursion
testpaths (args):     Directories to search for tests when no files or
                      directories are given on the command line
collect_imported_tests (bool):
                      Whether to collect tests in imported modules outside
                      `testpaths`
consider_namespace_packages (bool):
                      Consider namespace packages when resolving module
                      names during import
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)
strict_parametrization_ids (bool):
                      Emit an error if non-unique parameter set IDs are
                      detected
console_output_style (string):
                      Console output: "classic", or with additional
                      progress information ("progress" (percentage) |
                      "count" | "progress-even-when-capture-no" (forces
                      progress even when capture=no)
verbosity_test_cases (string):
                      Specify a verbosity level for test case execution,
                      overriding the main level. Higher levels will
                      provide more detailed information about each test
                      case executed.
strict_xfail (bool):  Default for the strict parameter of xfail markers
                      when not given explicitly (default: False) (alias:
                      xfail_strict)
```

(continues on next page)

(continued from previous page)

```
tmp_path_retention_count (string):
                      How many sessions should we keep the `tmp_path`
                      directories, according to
                      `tmp_path_retention_policy`.
tmp_path_retention_policy (string):
                      Controls which directories created by the `tmp_path`
                      fixture are kept around, based on test outcome.
                      (all/failed/none)
enable_assertion_pass_hook (bool):
                      Enables the pytest_assertion_pass hook. Make sure to
                      delete any previously generated pyc cache files.
truncation_limit_lines (string):
                      Set threshold of LINES after which truncation will
                      take effect
truncation_limit_chars (string):
                      Set threshold of CHARS after which truncation will
                      take effect
verbosity_assertions (string):
                      Specify a verbosity level for assertions, overriding
                      the main level. Higher levels will provide more
                      detailed explanation when an assertion fails.
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_mode (string):
```

(continues on next page)

(continued from previous page)

```
                        Default value for --log-file-mode
  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
  faulthandler_exit_on_timeout (bool):
                        Exit the test process if a test takes more than
                        faulthandler_timeout seconds to finish
  verbosity_subtests (string):
                        Specify verbosity level for subtests. Higher levels
                        will generate output for passed subtests. Failed
                        subtests are always reported.
  addopts (args):       Extra command line options
  minversion (string):  Minimally required pytest version
  pythonpath (paths):   Add paths to sys.path
  required_plugins (args):
                        Plugins that must be present for pytest to run

Environment variables:
  CI                        When set to a non-empty value, pytest knows it is running␣
↪in a CI process and does not truncate summary info
  BUILD_NUMBER              Equivalent to CI
  PYTEST_ADDOPTS            Extra command line options
  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
  PYTEST_DEBUG_TEMPROOT     Override the system temporary directory
  PYTEST_THEME              The Pygments style to use for code output
  PYTEST_THEME_MODE         Set the PYTEST_THEME to be either 'dark' or 'light'


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
```

CHAPTER

FOUR

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 align 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 reuse 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*.

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` directory. 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.

Create a `pyproject.toml` file in the root of your repository as described in Packaging Python Projects. The first few lines should look like this:

```
[build-system]
requires = ["hatchling"]
build-backend = "hatchling.build"

[project]
name = "PACKAGENAME"
version = "PACKAGEVERSION"
```

where `PACKAGENAME` and `PACKAGEVERSION` are the name and version of your package respectively.

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). Methods decorated with `@staticmethod` and `@classmethods` are also considered.

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

`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
src/
    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 .`.

For new projects, we recommend to use `importlib` *import mode* (see *which-import-mode* for a detailed explanation). To this end, add the following to your configuration file:

```
# content of pytest.toml
[pytest]
addopts = ["--import-mode=importlib"]
```

Generally, but especially if you use the default import mode `prepend`, it is **strongly** suggested to use a `src` layout. Here, your application root package resides in a sub-directory of your root, i.e. `src/mypkg/` instead of `mypkg`.

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

If you do not use an editable install and use the `src` layout as above you need to extend the Python's search path for module files to execute the tests against the local copy directly. You can do it in an ad-hoc manner by setting the `PYTHONPATH` environment variable:

```
PYTHONPATH=src pytest
```

or in a permanent manner by using the *pythonpath* configuration variable and adding the following to your configuration file:

```
[pytest]
pythonpath = ["src"]
```

```
[pytest]
pythonpath = src
```

Note

If you do not use an editable install and do not use the `src` layout (`mypkg` directly in the root directory) you can rely on the fact that Python by default puts the current directory in `sys.path` to import your package and run `python -m pytest` to execute the tests against the local copy directly.

See *Invoking pytest versus python -m pytest* for more information about the difference between calling `pytest` and `python -m pytest`.

See also

src layout vs flat layout
The Python Packaging User Guide discusses the trade-offs between the `src` layout and `flat` layout.

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
[src/]mypkg/
    __init__.py
    app.py
    view.py
    tests/
        __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`, making things much less surprising.

Choosing an import mode

For historical reasons, pytest defaults to the `prepend` *import mode* instead of the `importlib` import mode we recommend for new projects. The reason lies in the way the `prepend` mode works:

Since there are no packages to derive a full package name from, `pytest` will import your test files as *top-level* modules. The test files in the first example (*src layout*) would be imported as `test_app` and `test_view` top-level modules by adding `tests/` to `sys.path`.

This results in a drawback compared to the import mode `importlib`: your test files must have **unique names**.

If you need to have test modules with the same name, as a workaround you might add `__init__.py` files to your `tests` directory and subdirectories, changing them to packages:

```
pyproject.toml
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.

The `importlib` import mode does not have any of the drawbacks above, because `sys.path` is not changed when importing test modules.

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. `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.

4.3.6 Checking with flake8-pytest-style

In order to ensure that pytest is being used correctly in your project, it can be helpful to use the flake8-pytest-style flake8 plugin.

flake8-pytest-style checks for common mistakes and coding style violations in pytest code, such as incorrect use of fixtures, test function names, and markers. By using this plugin, you can catch these errors early in the development process and ensure that your pytest code is consistent and easy to maintain.

A list of the lints detected by flake8-pytest-style can be found on its PyPI page.

> **Note**
>
> flake8-pytest-style is not an official pytest project. Some of the rules enforce certain style choices, such as using `@pytest.fixture()` over `@pytest.fixture`, but you can configure the plugin to fit your preferred style.

4.3.7 Using pytest's strict mode

Added in version 9.0.

Pytest contains a set of configuration options that make it more strict. The options are off by default for compatibility or other reasons, but you should enable them if you can.

You can enable all of the strictness options at once by setting the *`strict`* configuration option:

```
[pytest]
strict = true
```

```
[pytest]
strict = true
```

See the *`strict`* documentation for the options it enables and their effect.

If pytest adds new strictness options in the future, they will also be enabled in strict mode. Therefore, you should only enable strict mode if you use a pinned/locked version of pytest, or if you want to proactively adopt new strictness options as they are added. If you don't want to automatically pick up new options, you can enable options individually:

```
[pytest]
strict_config = true
strict_markers = true
strict_parametrization_ids = true
strict_xfail = true
```

```
[pytest]
strict_config = true
strict_markers = true
strict_parametrization_ids = true
strict_xfail = true
```

If you want to use strict mode but are having trouble with a specific option, you can turn it off individually:

```
[pytest]
strict = true
strict_parametrization_ids = false
```

```
[pytest]
strict = true
strict_parametrization_ids = false
```

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.

Thread safety

pytest is single-threaded, executing its tests always in the same thread, sequentially, never spawning any threads itself.

Even in case of plugins which run tests in parallel, for example pytest-xdist, usually work by spawning multiple *processes* and running tests in batches, without using multiple threads.

It is of course possible (and common) for tests and fixtures to spawn threads themselves as part of their testing workflow (for example, a fixture that starts a server thread in the background, or a test which executes production code that spawns threads), but some care must be taken:

- Make sure to eventually wait on any spawned threads – for example at the end of a test, or during the teardown of a fixture.
- Avoid using primitives provided by pytest (`pytest.warns()`, `pytest.raises()`, etc) from multiple threads, as they are not thread-safe.

If your test suite uses threads and you are seeing flaky test results, do not discount the possibility that the test is implicitly using global state in pytest itself.

4.4.3 Related 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:

- pytest-rerunfailures
- pytest-replay: This plugin helps to reproduce locally crashes or flaky tests observed during CI runs.
- pytest-flakefinder - blog post

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, Tifany Yung, and Darko Marinov. "DeFlaker: Automatically detecting flaky tests." In *Proceedings of the 2018 International Conference on Software Engineering*. 2018. PDF
- Dutta, Saikat and Shi, August and Choudhary, Rutvik and Zhang, Zhekun and Jain, Aryaman and Misailovic, Sasa. "Detecting flaky tests in probabilistic and machine learning applications." In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA)*, pp. 211-224. ACM, 2020. PDF
- Habchi, Sarra and Haben, Guillaume and Sohn, Jeongju and Franci, Adriano and Papadakis, Mike and Cordy, Maxime and Le Traon, Yves. "What Made This Test Flake? Pinpointing Classes Responsible for Test Flakiness." In Proceedings of the 38th IEEE International Conference on Software Maintenance and Evolution (ICSME), IEEE, 2022. PDF
- Lamprou, Sokrates. "Non-deterministic tests and where to find them: Empirically investigating the relationship between flaky tests and test smells by examining test order dependency." Bachelor thesis, Department of Computer and Information Science, Linköping University, 2022. LIU-IDA/LITH-EX-G–19/056–SE. PDF
- Leinen, Fabian and Elsner, Daniel and Pretschner, Alexander and Stahlbauer, Andreas and Sailer, Michael and Jürgens, Elmar. "Cost of Flaky Tests in Continuous Integration: An Industrial Case Study." Technical University of Munich and CQSE GmbH, Munich, Germany, 2023. 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
- Dropbox: * Athena: Our automated build health management system by Utsav Shah, 2019 * How To Manage Flaky Tests in your CI Workflows by Li Haoyi, 2025
- Uber: * Handling Flaky Unit Tests in Java by Uber Engineering, 2021 * Flaky Tests Overhaul at Uber by Uber Engineering, 2024

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 is a non-trivial process, so 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 `importlib.import_module` function.

  It is highly recommended to arrange your test modules as packages by adding `__init__.py` files to your directories containing tests. This will make the tests part of a proper Python package, allowing pytest to resolve their full name (for example `tests.core.test_core` for `test_core.py` inside the `tests.core` package).

  If the test directory tree is not arranged as packages, then each test file needs to have a unique name compared to the other test files, otherwise pytest will raise an error if it finds two tests with the same name.

  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 `importlib.import_module`.

  This better allows users 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 be put in `sys.modules` after importing.

- `importlib`: this mode uses more fine control mechanisms provided by `importlib` to import test modules, without changing `sys.path`.

  Advantages of this mode:

- pytest will not change `sys.path` at all.
- Test module names do not need to be unique – pytest will generate a unique name automatically based on the `rootdir`.

Disadvantages:

- Test modules can't import each other.
- Testing utility modules in the tests directories (for example a `tests.helpers` module containing test-related functions/classes) are not importable. The recommendation in this case is to place testing utility modules together with the application/library code, for example `app.testing.helpers`.

  Important: by "test utility modules", we mean functions/classes which are imported by other tests directly; this does not include fixtures, which should be placed in `conftest.py` files, along with the test modules, and are discovered automatically by pytest.

It works like this:

1. Given a certain module path, for example `tests/core/test_models.py`, derives a canonical name like `tests.core.test_models` and tries to import it.

   For non-test modules, this will work if they are accessible via `sys.path`. So for example, `.env/lib/site-packages/app/core.py` will be importable as `app.core`. This happens when plugins import non-test modules (for example doctesting).

   If this step succeeds, the module is returned.

   For test modules, unless they are reachable from `sys.path`, this step will fail.

2. If the previous step fails, we import the module directly using `importlib` facilities, which lets us import it without changing `sys.path`.

   Because Python requires the module to also be available in `sys.modules`, pytest derives a unique name for it based on its relative location from the `rootdir`, and adds the module to `sys.modules`.

   For example, `tests/core/test_models.py` will end up being imported as the module `tests.core.test_models`.

Added in version 6.0.

Note

Initially we intended to make `importlib` the default in future releases, however it is clear now that it has its own set of drawbacks so the default will remain `prepend` for the foreseeable future.

Note

By default, pytest will not attempt to resolve namespace packages automatically, but that can be changed via the *`consider_namespace_packages`* configuration variable.

See also

The *`pythonpath`* configuration variable.

The *`consider_namespace_packages`* configuration variable.

Choosing a test layout.

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 directory. It will then search upwards until it can find the last directory 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 detail 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 directory. 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 detail 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*.

CHAPTER

FIVE

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

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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project/assertion
collected 44 items

failure_demo.py FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF         [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

failure_demo.py:21: AssertionError
_____________________________ TestFailing.test_simple _____________________________
```

(continues on next page)

(continued from previous page)

```
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:32: 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:35:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = 42, b = 54

    def otherfunc_multi(a, b):
>       assert a == b
E       assert 42 == 54

failure_demo.py:16: 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:41: AssertionError
___________________ TestSpecialisedExplanations.test_eq_text ___________________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0007>
```

(continues on next page)

(continued from previous page)

```
    def test_eq_text(self):
>       assert "spam" == "eggs"
E       AssertionError: assert 'spam' == 'eggs'
E
E         - eggs
E         + spam

failure_demo.py:46: 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
E         - foo 2 bar
E         ?     ^
E         + foo 1 bar
E         ?     ^

failure_demo.py:49: 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
E           foo
E         - eggs
E         + spam
E           bar

failure_demo.py:52: 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...2222222222222' == '111111111111...
↪2222222222222'
E
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         - 1111111111b222222222
E         ?           ^
E         + 1111111111a222222222
```

(continues on next page)

(continued from previous page)

```
E         ?           ^

failure_demo.py:57: 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
E       AssertionError: assert '1\n1\n1\n1\n...n2\n2\n2\n2\n' == '1\n1\n1\n1\n...n2\
↪n2\n2\n2\n'
E
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
E         ...Full output truncated (7 lines hidden), use '-vv' to show

failure_demo.py:62: 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
E         At index 2 diff: 2 != 3
E         Use -v to get more diff

failure_demo.py:65: 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
E         At index 100 diff: 1 != 2
E         Use -v to get more diff

failure_demo.py:70: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_dict _____________________
```

(continues on next page)

(continued from previous page)

```
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
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         Use -v to get more diff

failure_demo.py:73: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_set _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef000f>

    def test_eq_set(self):
>       assert {0, 10, 11, 12} == {0, 20, 21}
E       assert {0, 10, 11, 12} == {0, 20, 21}
E
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         Use -v to get more diff

failure_demo.py:76: 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
E         Right contains one more item: 3
E         Use -v to get more diff

failure_demo.py:79: 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]
```

(continues on next page)

(continued from previous page)

```
E       assert 1 in [0, 2, 3, 4, 5]

failure_demo.py:82: 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...nand a\ntail'
E
E         'foo' is contained here:
E           some multiline
E           text
E           which
E           includes foo
E         ?          +++
E           and a
E           tail

failure_demo.py:86: AssertionError
_____________ TestSpecialisedExplanations.test_not_in_text_single _____________

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
E         'foo' is contained here:
E           single foo line
E         ?        +++

failure_demo.py:90: 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
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 tail
E         ?           +++

failure_demo.py:94: AssertionError
______ TestSpecialisedExplanations.test_not_in_text_single_long_term _______
```

(continues on next page)

(continued from previous page)

```
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 'fffffffffff...fffffffffff' not in 'head head h...l␣
↪tail tail '
E
E         'fffffffffffffffff...fffffffffffffffff' is contained here:
E           head head␣
↪fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffftail tail␣
↪tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail␣
↪tail tail
E         ?          ␣
↪++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

failure_demo.py:98: 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")
>       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           - c
E           + b

failure_demo.py:110: AssertionError
_____________ TestSpecialisedExplanations.test_eq_attrs _____________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef0017>

    def test_eq_attrs(self):
        import attr
```

(continues on next page)

(continued from previous page)

```
        @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           - c
E           + b

failure_demo.py:122: 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

failure_demo.py:130: AssertionError
_____________________________ test_attribute_instance _____________________________

    def test_attribute_instance():
        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:137: AssertionError
_____________________________ test_attribute_failure _____________________________

    def test_attribute_failure():
        class Foo:
```

(continues on next page)

(continued from previous page)

```
            def _get_b(self):
                raise Exception("Failed to get attrib")

            b = property(_get_b)

        i = Foo()
>       assert i.b == 2
               ^^^

failure_demo.py:148:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

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:143: 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:158: AssertionError
_____________________________ TestRaises.test_raises _____________________________

self = <failure_demo.TestRaises object at 0xdeadbeef001d>

    def test_raises(self):
        s = "qwe"
>       raises(TypeError, int, s)
E       ValueError: invalid literal for int() with base 10: 'qwe'

failure_demo.py:168: ValueError
_________________________ TestRaises.test_raises_doesnt __________________________
```

(continues on next page)

(continued from previous page)

```
self = <failure_demo.TestRaises object at 0xdeadbeef001e>

    def test_raises_doesnt(self):
>       raises(OSError, int, "3")
E       Failed: DID NOT RAISE <class 'OSError'>

failure_demo.py:171: 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:174: ValueError
____________________________ TestRaises.test_tupleerror ____________________________

self = <failure_demo.TestRaises object at 0xdeadbeef0020>

    def test_tupleerror(self):
>       a, b = [1]  # noqa: F841
        ^^^^
E       ValueError: not enough values to unpack (expected 2, got 1)

failure_demo.py:177: 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:182: 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: F821
           ^^^^^^^^^^
E       NameError: name 'namenotexi' is not defined

failure_demo.py:185: NameError
_____________________ test_dynamic_compile_shows_nicely ______________________
```

(continues on next 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:204:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

>   ???
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:215:
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _
failure_demo.py:12: in somefunc
    otherfunc(x, y)
_ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _ _

a = 44, b = 43

    def otherfunc(a, b):
>       assert a == b
E       assert 44 == 43

failure_demo.py:8: AssertionError
____________________ TestMoreErrors.test_z1_unpack_error ____________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef0024>

    def test_z1_unpack_error(self):
        items = []
>       a, b = items
```

(continues on next page)

(continued from previous page)

```
        ^^^^
E       ValueError: not enough values to unpack (expected 2, got 0)

failure_demo.py:219: ValueError
_____________________ 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:223: 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>
↪('456')
E        +    where <built-in method startswith of str object at 0xdeadbeef0027> =
↪'123'.startswith

failure_demo.py:228: 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>
↪('456')
E        +    where <built-in method startswith of str object at 0xdeadbeef0027> =
↪'123'.startswith
E        +      where '123' = <function TestMoreErrors.test_startswith_nested.<locals>
↪.f at 0xdeadbeef0029>()
E        +    and   '456' = <function TestMoreErrors.test_startswith_nested.<locals>.
↪g at 0xdeadbeef002a>()
```

(continues on next page)

(continued from previous page)

```
failure_demo.py:237: 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)

failure_demo.py:240: 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:244: 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:247: 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:252: AssertionError
_____________________ TestCustomAssertMsg.test_single_line ____________________

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef002f>

    def test_single_line(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"
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:263: AssertionError
_____________________ TestCustomAssertMsg.test_multiline ____________________

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef0030>

    def test_multiline(self):
        class A:
            a = 1

        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:270: 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:283: AssertionError
========================= short test summary info ==========================
FAILED failure_demo.py::test_generative[3-6] - assert (3 * 2) < 6
```

(continues on next page)

(continued from previous page)

```
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...
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.toml
[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:

```
<configuration file 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 conftest.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 conftest.py
import pytest
```

(continues on next page)

(continued from previous page)

```
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)
  inifile: None
  rootdir: /home/sweet/project
```

If you need to provide more detailed error messages, you can use the `type` parameter and raise *`pytest.UsageError`*:

```
# content of conftest.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:])
    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:

```
# installable external 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-9.x.y, pluggy-1.x.y
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 `conftest.py` file adding a `--runslow` command line option to control skipping of `pytest.mark.slow` marked tests:

```
# content of conftest.py

import pytest


def pytest_addoption(parser):
    parser.addoption(
        "--runslow", action="store_true", default=False, help="run slow tests"
    )


def pytest_configure(config):
    config.addinivalue_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:
```

(continues on next page)

(continued from previous page)

```
            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-9.x.y, pluggy-1.x.y
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-9.x.y, pluggy-1.x.y
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):
```

(continues on next page)

(continued from previous page)

```
    __tracebackhide__ = True
    if not hasattr(x, "config"):
        pytest.fail(f"not configured: {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


class ConfigException(Exception):
    pass


def checkconfig(x):
    __tracebackhide__ = operator.methodcaller("errisinstance", ConfigException)
    if not hasattr(x, "config"):
        raise ConfigException(f"not configured: {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 this:

```
import os


if os.environ.get("PYTEST_VERSION") is not None:
    # Things you want to do if your code is called by pytest.
    ...
else:
    # Things you want to do if your code is not called by pytest.
    ...
```

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-9.x.y, pluggy-1.x.y
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.get_verbosity() > 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-9.x.y, pluggy-1.x.y -- $PYTHON_PREFIX/bin/
↪python
cachedir: .pytest_cache
info1: did you know that ...
```

(continues on next page)

(continued from previous page)

```
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-9.x.y, pluggy-1.x.y
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():
    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-9.x.y, pluggy-1.x.y
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

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")
                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)
```

(continues on next page)

(continued from previous page)

```
            # if name found, test has failed for the combination of class name & test␣
↪name
            if test_name is not None:
                pytest.xfail(f"previous test failed ({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-9.x.y, pluggy-1.x.y
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 - 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="package")
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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 7 items

a/test_db.py F                                                       [ 14%]
a/test_db2.py F                                                      [ 28%]
b/test_error.py E                                                    [ 42%]
test_step.py .Fx.                                                    [100%]
```

(continues on next page)

(continued from previous page)

```
================================== 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,␣
↪capteesys, doctest_namespace, monkeypatch, pytestconfig, record_property, record_
↪testsuite_property, record_xml_attribute, recwarn, subtests, 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 =================================
_____________________________________ test_a1 ______________________________________

db = <conftest.DB object at 0xdeadbeef0002>

    def test_a1(db):
>       assert 0, db  # to show value
        ^^^^^^^^^^^
E       AssertionError: <conftest.DB object at 0xdeadbeef0002>
E       assert 0

a/test_db.py:2: AssertionError
_____________________________________ test_a2 ______________________________________

db = <conftest.DB object at 0xdeadbeef0002>

    def test_a2(db):
>       assert 0, db  # to show value
        ^^^^^^^^^^^
E       AssertionError: <conftest.DB object at 0xdeadbeef0002>
E       assert 0

a/test_db2.py:2: AssertionError
_____________________ TestUserHandling.test_modification ______________________

self = <test_step.TestUserHandling object at 0xdeadbeef0003>

    def test_modification(self):
>       assert 0
E       assert 0

test_step.py:11: AssertionError
========================== short test summary info ==========================
FAILED a/test_db.py::test_a1 - AssertionError: <conftest.DB object at 0x7...
FAILED a/test_db2.py::test_a2 - AssertionError: <conftest.DB object at 0x...
FAILED test_step.py::TestUserHandling::test_modification - assert 0
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 `conftest.py` file. Note that each fixture is

only instantiated if there is a test actually needing it (unless you use "autouse" fixtures 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 conftest.py

import os.path

import pytest


@pytest.hookimpl(wrapper=True, tryfirst=True)
def pytest_runtest_makereport(item, call):
    # execute all other hooks to obtain the report object
    rep = yield

    # 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, encoding="utf-8") 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")

    return rep
```

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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 2 items
```

(continues on next page)

(continued from previous page)

```
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:

```python
# content of conftest.py
import pytest
from pytest import StashKey, CollectReport

phase_report_key = StashKey[dict[str, CollectReport]]()


@pytest.hookimpl(wrapper=True, tryfirst=True)
def pytest_runtest_makereport(item, call):
    # execute all other hooks to obtain the report object
    rep = yield

    # store test results for each phase of a call, which can
    # be "setup", "call", "teardown"
    item.stash.setdefault(phase_report_key, {})[rep.when] = rep

    return rep
```

(continues on next page)

(continued from previous page)

```
@pytest.fixture
def something(request):
    yield
    # request.node is an "item" because we use the default
    # "function" scope
    report = request.node.stash[phase_report_key]
    if report["setup"].failed:
        print("setting up a test failed", request.node.nodeid)
    elif report["setup"].skipped:
        print("setting up a test skipped", request.node.nodeid)
    elif ("call" not in report) or report["call"].failed:
        print("executing test failed or skipped", 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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 3 items

test_module.py Esetting up a test failed test_module.py::test_setup_fails
Fexecuting test failed or skipped test_module.py::test_call_fails
F

================================== ERRORS ==================================
_____________________ ERROR at setup of test_setup_fails ______________________
```

(continues on next page)

(continued from previous page)

```
    @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

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 (*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 --junit=xml=results.xml test-suite/
```

5.1.3 Parametrizing tests

`pytest` allows you 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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 8 items

<Dir parametrize.rst-213>
  <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])
```

(continues on next page)

(continued from previous page)

```
    metafunc.parametrize(argnames, argvalues, ids=idlist, scope="class")


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-9.x.y, pluggy-1.x.y
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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 4 items

<Dir parametrize.rst-213>
  <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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 2 items

<Dir parametrize.rst-213>
```

(continues on next page)

(continued from previous page)

```
  <Module test_backends.py>
    <Function test_db_initialized[d1]>
    <Function test_db_initialized[d2]>

======================== 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 one 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-9.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__]
```

(continues on next page)

(continued from previous page)

```
    argnames = sorted(funcarglist[0])
    metafunc.parametrize(
        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
```

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."""
```

(continues on next page)

(continued from previous page)

```
from __future__ import annotations

import shutil
import subprocess
import textwrap

import pytest


pythonlist = ["python3.11", "python3.12", "python3.13"]


@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(
                rf"""
                import pickle
                f = open({str(self.picklefile)!r}, 'wb')
                s = pickle.dump({obj!r}, f, protocol=2)
                f.close()
                """
            )
        )
        subprocess.run((self.pythonpath, str(dumpfile)), check=True)

    def load_and_is_true(self, expression):
        loadfile = self.picklefile.with_name("load.py")
        loadfile.write_text(
            textwrap.dedent(
                rf"""
                import pickle
                f = open({str(self.picklefile)!r}, 'rb')
                obj = pickle.load(f)
```

(continues on next page)

(continued from previous page)

```
                f.close()
                res = eval({expression!r})
                if not res:
                    raise SystemExit(1)
                """
            )
        )
        print(loadfile)
        subprocess.run((self.pythonpath, str(loadfile)), check=True)


@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
sssssssssss......sss......                                              [100%]
========================= short test summary info ==========================
SKIPPED [15] multipython.py:67: 'python3.11' not found
12 passed, 15 skipped in 0.12s
```

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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 2 items

test_module.py .s                                                        [100%]

========================== short test summary info ===========================
SKIPPED [1] test_module.py:3: 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"
        ),
    ],
)
```

(continues on next page)

(continued from previous page)

```
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-9.x.y, pluggy-1.x.y -- $PYTHON_PREFIX/bin/
↪python
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 were 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.

`contextlib.nullcontext` can be used to test cases that are not expected to raise exceptions but that should result in some value. The value is given as the `enter_result` parameter, which will be available as the `with` statement's target (`e` in the example below).

For example:

```
from contextlib import nullcontext

import pytest


@pytest.mark.parametrize(
    "example_input,expectation",
    [
        (3, nullcontext(2)),
```

(continues on next page)

(continued from previous page)

```
        (2, nullcontext(3)),
        (1, nullcontext(6)),
        (0, pytest.raises(ZeroDivisionError)),
    ],
)
def test_division(example_input, expectation):
    """Test how much I know division."""
    with expectation as e:
        assert (6 / example_input) == e
```

In the example above, the first three test cases should run without any exceptions, while the fourth should raise a `ZeroDivisionError` exception, which is expected by pytest.

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


@pytest.mark.device(serial="123")
def test_something_quick():
    pass


@pytest.mark.device(serial="abc")
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-9.x.y, pluggy-1.x.y -- $PYTHON_PREFIX/bin/
↪python
cachedir: .pytest_cache
rootdir: /home/sweet/project
```

(continues on next page)

(continued from previous page)

```
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-9.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 =======================
```

Additionally, you can restrict a test run to only run tests matching one or multiple marker keyword arguments, e.g. to run only tests marked with `device` and the specific `serial="123"`:

```
$ pytest -v -m "device(serial='123')"
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-9.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_something_quick PASSED                          [100%]

====================== 1 passed, 3 deselected in 0.12s =======================
```

Note

Only keyword argument matching is supported in marker expressions.

Note

Only `int`, (unescaped) `str`, `bool` & `None` values are supported in marker expressions.

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-9.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-9.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-9.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%]
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 --collect-only`.

Using `-k expr` to select tests based on their name

Added 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-9.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-9.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-9.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 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.toml
[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=strict_xfail): 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␣
```

(continues on next page)

(continued from previous page)

```
↪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. DEPRECATED, use @pytest.
↪hookimpl(tryfirst=True) instead.

@pytest.mark.trylast: mark a hook implementation function such that the plugin␣
↪machinery will try to call it last/as late as possible. DEPRECATED, use @pytest.
↪hookimpl(trylast=True) instead.
```

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`* configuration 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 specified 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.addinivalue_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(f"test requires env in {envnames!r}")
```

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 invocation specifying a different environment than what the test needs:

```
$ pytest -E stage2
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 1 item

test_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-9.x.y, pluggy-1.x.y
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
```

(continues on next page)

(continued from previous page)

```
@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=strict_xfail): 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. DEPRECATED, use @pytest.
↪hookimpl(tryfirst=True) instead.

@pytest.mark.trylast: mark a hook implementation function such that the plugin␣
↪machinery will try to call it last/as late as possible. DEPRECATED, use @pytest.
↪hookimpl(trylast=True) instead.
```

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


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(f"glob args={mark.args} kwargs={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(f"cannot run on platform {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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
collected 4 items

test_plat.py s.s.                                                        [100%]

========================== short test summary info ==========================
SKIPPED [2] conftest.py:13: 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-9.x.y, pluggy-1.x.y
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:

```python
# 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-9.x.y, pluggy-1.x.y
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-9.x.y, pluggy-1.x.y
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
```

(continues on next page)

(continued from previous page)

```
    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_method2 called")


class TestOther:
```

(continues on next page)

(continued from previous page)

```
    @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):
        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_method2 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
```

(continues on next page)

(continued from previous page)

```
        |-- 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
...
```

Changing directory recursion

You can set the *norecursedirs* option in a configuration file:

```
# content of pytest.toml
[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.toml
# 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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
configfile: pytest.toml
collected 2 items

<Dir pythoncollection.rst-214>
  <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:

```
# content of pytest.toml
# Example 2: have pytest look for files with "test" and "example"
```

(continues on next page)

(continued from previous page)

```
[pytest]
python_files = ["test_*.py", "example_*.py"]
```

Note

the `python_functions` and `python_classes` options have 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 a configuration file and the *`addopts`* option you can make this change more permanently:

```
# content of pytest.toml
[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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
configfile: pytest.toml
collected 3 items

<Dir pythoncollection.rst-214>
  <Dir CWD>
    <Module 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.toml
[pytest]
python_files = ["*.py"]
```

However, many projects will have a `setup.py` which they don't want to be imported. Moreover, there may be 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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project
configfile: pytest.toml
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 `conftest.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 conftest.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
```

Note

If you are working with abstract test classes and want to avoid manually setting the `__test__` attribute for subclasses, you can use a mixin class to handle this automatically. For example:

```
# Mixin to handle abstract test classes
class NotATest:
    def __init_subclass__(cls):
        cls.__test__ = NotATest not in cls.__bases__


# Abstract test class
class AbstractTest(NotATest):
    pass


# Subclass that will be collected as a test
class RealTest(AbstractTest):
    def test_example(self):
        assert 1 + 1 == 2
```

This approach ensures that subclasses of abstract test classes are automatically collected without needing to explicitly set the `__test__` attribute.

5.1.7 Working with non-python tests

A basic example for specifying tests in Yaml files

Here is an example `conftest.py` (extracted from Ali Afshar's special purpose pytest-yamlwsgi plugin). This `conftest.py` will collect `test*.yaml` files and will execute the yaml-formatted content as custom tests:

```
# content of conftest.py
from __future__ import annotations

import pytest
```

(continues on next page)

(continued from previous page)

```
def pytest_collect_file(parent, file_path):
    if file_path.suffix == ".yaml" and file_path.name.startswith("test"):
        return YamlFile.from_parent(parent, path=file_path)


class YamlFile(pytest.File):
    def collect(self):
        # We need a yaml parser, e.g. PyYAML.
        import yaml

        raw = yaml.safe_load(self.path.open(encoding="utf-8"))
        for name, spec in sorted(raw.items()):
            yield YamlItem.from_parent(self, name=name, spec=spec)


class YamlItem(pytest.Item):
    def __init__(self, *, spec, **kwargs):
        super().__init__(**kwargs)
        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.",
                ]
            )
        return super().repr_failure(excinfo)

    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
```

(continues on next page)

(continued from previous page)

```
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-9.x.y, pluggy-1.x.y
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 - usecase execution failed
======================== 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. It should return a tuple `(path, lineno, description)`, where:

- `path` is the path shown in reports (usually `self.path` or `self.fspath`).
- `lineno` is a zero-based line number, or `0` when no specific line applies.
- `description` is a short label shown for the collected item:

```
nonpython $ pytest -v
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-9.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 =================================
```

(continues on next page)

(continued from previous page)

```
_____________________________ 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 - usecase execution failed
======================== 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-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project/nonpython
collected 2 items

<Package nonpython>
  <YamlFile test_simple.yaml>
    <YamlItem hello>
    <YamlItem ok>

======================== 2 tests collected in 0.12s =========================
```

5.1.8 Using a custom directory collector

By default, pytest collects directories using *pytest.Package*, for directories with `__init__.py` files, and *pytest.Dir* for other directories. If you want to customize how a directory is collected, you can write your own *pytest.Directory* collector, and use *pytest_collect_directory* to hook it up.

A basic example for a directory manifest file

Suppose you want to customize how collection is done on a per-directory basis. Here is an example `conftest.py` plugin that allows directories to contain a `manifest.json` file, which defines how the collection should be done for the directory. In this example, only a simple list of files is supported, however you can imagine adding other keys, such as exclusions and globs.

```python
# content of conftest.py
from __future__ import annotations

import json

import pytest


class ManifestDirectory(pytest.Directory):
    def collect(self):
        # The standard pytest behavior is to loop over all `test_*.py` files and
        # call `pytest_collect_file` on each file. This collector instead reads
        # the `manifest.json` file and only calls `pytest_collect_file` for the
        # files defined there.
        manifest_path = self.path / "manifest.json"
        manifest = json.loads(manifest_path.read_text(encoding="utf-8"))
```

(continues on next page)

(continued from previous page)

```
        ihook = self.ihook
        for file in manifest["files"]:
            yield from ihook.pytest_collect_file(
                file_path=self.path / file, parent=self
            )


@pytest.hookimpl
def pytest_collect_directory(path, parent):
    # Use our custom collector for directories containing a `manifest.json` file.
    if path.joinpath("manifest.json").is_file():
        return ManifestDirectory.from_parent(parent=parent, path=path)
    # Otherwise fallback to the standard behavior.
    return None
```

You can create a `manifest.json` file and some test files:

```
{
    "files": [
        "test_first.py",
        "test_second.py"
    ]
}
```

```
# content of test_first.py
from __future__ import annotations


def test_1():
    pass
```

```
# content of test_second.py
from __future__ import annotations


def test_2():
    pass
```

```
# content of test_third.py
from __future__ import annotations


def test_3():
    pass
```

And you can now execute the test specification:

```
customdirectory $ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project/customdirectory
configfile: pytest.ini
```

(continues on next page)

(continued from previous page)

```
collected 2 items

tests/test_first.py .                                                    [ 50%]
tests/test_second.py .                                                   [100%]

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

Notice how `test_three.py` was not executed, because it is not listed in the manifest.

You can verify that your custom collector appears in the collection tree:

```
customdirectory $ pytest --collect-only
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-9.x.y, pluggy-1.x.y
rootdir: /home/sweet/project/customdirectory
configfile: pytest.ini
collected 2 items

<Dir customdirectory>
  <ManifestDirectory tests>
    <Module test_first.py>
      <Function test_1>
    <Module test_second.py>
      <Function test_2>

======================== 2 tests collected in 0.12s =========================
```

5.2 Backwards Compatibility Policy

Pytest is an actively evolving 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 that 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 or better mechanisms through documentation.

b) transitional: the old and new APIs don't conflict, and we can help users transition by using warnings while supporting both for a prolonged period of 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).

   A deprecated feature scheduled to be removed in major version X will use the warning class `PytestRemovedInXWarning` (a subclass of *`PytestDeprecationWarning`*).

   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 `PytestRemovedInXWarning` (e.g., `PytestRemovedIn4Warning`) 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 a normal transition is unreasonably unsustainable and would offset important developments or 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_runtest_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 *-1* 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 primarily 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` configuration option (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 Python version support

Released pytest versions support all Python versions that are actively maintained at the time of the release:

pytest version	min. Python version
9.0+	3.10+
8.4	3.9+
8.0 - 8.3	3.8+
7.1 - 7.4	3.7+
6.2 - 7.0	3.6+
5.0 - 6.1	3.5+
3.3 - 4.6	2.7, 3.4+

Status of Python Versions.

5.5 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.

5.5.1 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*.

`request.getfixturevalue()` during fixture teardown

Deprecated since version 9.1.

Calling *`request.getfixturevalue()`* during teardown to request a fixture that was not already requested is deprecated.

This pattern is brittle because teardown runs after pytest has started unwinding active scopes. Depending on the requested fixture's scope and the current teardown order, the lookup may appear to work, or it may fail.

In pytest 10, first-time fixture requests made during teardown will become an error. If teardown logic needs another fixture, request it before teardown begins, either by declaring it in the fixture signature or by calling `request.getfixturevalue()` before the fixture yields.

Fixtures that were already requested before teardown started are unaffected and may still be retrieved while they remain active, though this is discouraged.

`config.inicfg`

Deprecated since version 9.0.

The private `config.inicfg` attribute is deprecated. Use *`config.getini()`* to access configuration values instead.

`config.inicfg` was never documented and it should have had a _ prefix from the start. Pytest performs caching, transformation and aliasing on configuration options which make direct access to the raw `config.inicfg` untenable.

Reading configuration values:

Instead of accessing `config.inicfg` directly, use *`config.getini()`*:

```
# Deprecated
value = config.inicfg["some_option"]

# Use this instead
value = config.getini("some_option")
```

Setting configuration values:

Setting or deleting configuration values after initialization is not supported. If you need to override configuration values, use the `-o` command line option:

```
pytest -o some_option=value
```

or set them in your configuration file instead.

Non-Collection iterables in `@pytest.mark.parametrize`

Deprecated since version 9.1.

Using non-`Collection` iterables (such as generators, iterators, or custom iterable objects) for the `argvalues` parameter in *@pytest.mark.parametrize* and *`metafunc.parametrize`* is deprecated.

These iterables get exhausted after the first iteration, leading to tests getting unexpectedly skipped in cases such as:

- Running *`pytest.main()`* multiple times in the same process
- Using class-level parametrize decorators where the same mark is applied to multiple test methods
- Collecting tests multiple times

Example of problematic code:

```
import pytest


def data_generator():
    yield 1
    yield 2


@pytest.mark.parametrize("n", data_generator())
class Test:
    def test_1(self, n):
        pass

    # test_2 will be skipped because data_generator() is exhausted.
    def test_2(self, n):
        pass
```

You can fix it by converting generators and iterators to lists or tuples:

```
import pytest
```

(continues on next page)

(continued from previous page)

```
def data_generator():
    yield 1
    yield 2


@pytest.mark.parametrize("n", list(data_generator()))
class Test:
    def test_1(self, n):
        pass

    def test_2(self, n):
        pass
```

Note that `range` objects are `Collection` and are not affected by this deprecation.

`monkeypatch.syspath_prepend` with legacy namespace packages

Deprecated since version 9.0.

When using *monkeypatch.syspath_prepend()*, pytest automatically calls `pkg_resources.fixup_namespace_packages()` if `pkg_resources` is imported. This is only needed for legacy namespace packages that use `pkg_resources.declare_namespace()`.

Legacy namespace packages are deprecated in favor of native namespace packages (**PEP 420**). If you are using `pkg_resources.declare_namespace()` in your `__init__.py` files, you should migrate to native namespace packages by removing the `__init__.py` files from your namespace packages.

This deprecation warning will only be issued when:

1. `pkg_resources` is imported, and
2. The specific path being prepended contains a declared namespace package (via `pkg_resources.declare_namespace()`)

To fix this warning, convert your legacy namespace packages to native namespace packages:

Legacy namespace package (deprecated):

```
# mypkg/__init__.py
__import__("pkg_resources").declare_namespace(__name__)
```

Native namespace package (recommended):

Simply remove the `__init__.py` file entirely. Python 3.3+ natively supports namespace packages without `__init__.py`.

Configuring hook specs/impls using markers

Deprecated since version 7.2.

Before pluggy, pytest's plugin library, was its own package and had a clear API, pytest just used `pytest.mark` to configure hooks.

The *pytest.hookimpl()* and *pytest.hookspec()* decorators have been available since years and should be used instead.

```
@pytest.mark.tryfirst
def pytest_runtest_call(): ...


# or
def pytest_runtest_call(): ...


pytest_runtest_call.tryfirst = True
```

should be changed to:

```
@pytest.hookimpl(tryfirst=True)
def pytest_runtest_call(): ...
```

Changed `hookimpl` attributes:

- `tryfirst`
- `trylast`
- `optionalhook`
- `hookwrapper`

Changed `hookwrapper` attributes:

- `firstresult`
- `historic`

Directly constructing internal classes

Deprecated since version 7.0.

Directly constructing the following classes is now deprecated:

- `_pytest.mark.structures.Mark`
- `_pytest.mark.structures.MarkDecorator`
- `_pytest.mark.structures.MarkGenerator`
- `_pytest.python.Metafunc`
- `_pytest.runner.CallInfo`
- `_pytest._code.ExceptionInfo`
- `_pytest.config.argparsing.Parser`
- `_pytest.config.argparsing.OptionGroup`
- `_pytest.pytester.HookRecorder`

These constructors have always been considered private, but now issue a deprecation warning, which may become a hard error in pytest 8.

Diamond inheritance between *pytest.Collector* and *pytest.Item*

Deprecated since version 7.0.

Defining a custom pytest node type which is both an *Item* and a *Collector* (e.g. *File*) now issues a warning. It was never sanely supported and triggers hard to debug errors.

Some plugins providing linting/code analysis have been using this as a hack. Instead, a separate collector node should be used, which collects the item. See *Working with non-python tests* for an example, as well as an example pr fixing inheritance.

Constructors of custom *Node* subclasses should take **kwargs

Deprecated since version 7.0.

If custom subclasses of nodes like *pytest.Item* override the `__init__` method, they should take `**kwargs`. Thus,

```
class CustomItem(pytest.Item):
    def __init__(self, name, parent, additional_arg):
        super().__init__(name, parent)
        self.additional_arg = additional_arg
```

should be turned into:

```
class CustomItem(pytest.Item):
    def __init__(self, *, additional_arg, **kwargs):
        super().__init__(**kwargs)
        self.additional_arg = additional_arg
```

to avoid hard-coding the arguments pytest can pass to the superclass. See *Working with non-python tests* for a full example.

For cases without conflicts, no deprecation warning is emitted. For cases with conflicts (such as *pytest.File* now taking `path` instead of `fspath`, as *outlined above*), a deprecation warning is now raised.

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 it can be searched/replaced safely.

5.5.2 Removed Features and Breaking Changes

As stated in our *Backwards Compatibility Policy* policy, deprecated features are removed only in major releases after an appropriate period of deprecation has passed.

Some breaking changes which could not be deprecated are also listed.

`pytest.importorskip` default behavior regarding `ImportError`

Deprecated since version 8.2.

Removed in version 9.1.

Traditionally *pytest.importorskip()* captured `ImportError`, with the original intent being to skip tests where a dependent module is not installed, for example testing with different dependencies.

However, some packages might be installed in the system but not importable due to some other issue, for example a compilation error or a broken installation. In those cases, *pytest.importorskip()* would still silently skip the test, but more often than not users would rather see the unexpected error so the underlying issue can be fixed.

In 8.2, the exc_type parameter was added, giving users the ability to pass ModuleNotFoundError to skip tests only if the module cannot really be found, and not because of some other error.

As of 9.1, *pytest.importorskip()* only captures ModuleNotFoundError by default. If you want to preserve the previous behavior and skip on other ImportError exceptions during import, pass exc_type=ImportError explicitly.

fspath argument for Node constructors replaced with pathlib.Path

Deprecated since version 7.0.

Removed in version 9.1.

In order to support the transition from py.path.local to pathlib, the fspath argument to *Node* constructors like *pytest.Function.from_parent()* and *pytest.Class.from_parent()* is now deprecated.

Plugins which construct nodes should pass the path argument, of type pathlib.Path, instead of the fspath argument.

Plugins which implement custom items and collectors are encouraged to replace fspath parameters (py.path.local) with path parameters (pathlib.Path), and drop any other usage of the py library if possible.

If possible, plugins with custom items should use *cooperative constructors* to avoid hardcoding arguments they only pass on to the superclass.

Note

The name of the *Node* arguments and attributes (the new attribute being path) is **the opposite** of the situation for hooks, *outlined below* (the old argument being path).

This is an unfortunate artifact due to historical reasons, which should be resolved in future versions as we slowly get rid of the py dependency (see #9283 for a longer discussion).

Due to the ongoing migration of methods like *reportinfo()* which still is expected to return a py.path.local object, nodes still have both fspath (py.path.local) and path (pathlib.Path) attributes, no matter what argument was used in the constructor. We expect to deprecate the fspath attribute in a future release.

sync test depending on async fixture

Deprecated since version 8.4.

Removed in version 9.0.

Pytest has for a long time given an error when encountering an asynchronous test function, prompting the user to install a plugin that can handle it. It has not given any errors if you have an asynchronous fixture that's depended on by a synchronous test. If the fixture was an async function you did get an "unawaited coroutine" warning, but for async yield fixtures you didn't even get that. This is a problem even if you do have a plugin installed for handling async tests, as they may require special decorators for async fixtures to be handled, and some may not robustly handle if a user accidentally requests an async fixture from their sync tests. Fixture values being cached can make this even more unintuitive, where everything will "work" if the fixture is first requested by an async test, and then requested by a synchronous test.

Unfortunately there is no 100% reliable method of identifying when a user has made a mistake, versus when they expect an unawaited object from their fixture that they will handle on their own. To suppress this warning when you in fact did intend to handle this you can wrap your async fixture in a synchronous fixture:

```
import asyncio
import pytest


@pytest.fixture
```

(continues on next page)

(continued from previous page)

```
async def unawaited_fixture():
    return 1


def test_foo(unawaited_fixture):
    assert 1 == asyncio.run(unawaited_fixture)
```

should be changed to

```
import asyncio
import pytest


@pytest.fixture
def unawaited_fixture():
    async def inner_fixture():
        return 1

    return inner_fixture()


def test_foo(unawaited_fixture):
    assert 1 == asyncio.run(unawaited_fixture)
```

You can also make use of `pytest_fixture_setup` to handle the coroutine/asyncgen before pytest sees it - this is the way current async pytest plugins handle it.

If a user has an async fixture with `autouse=True` in their `conftest.py`, or in a file containing both synchronous tests and the fixture, they will receive this warning. Unless you're using a plugin that specifically handles async fixtures with synchronous tests, we strongly recommend against this practice. It can lead to unpredictable behavior (with larger scopes, it may appear to "work" if an async test is the first to request the fixture, due to value caching) and will generate unawaited-coroutine runtime warnings (but only for non-yield fixtures). Additionally, it creates ambiguity for other developers about whether the fixture is intended to perform setup for synchronous tests.

The anyio pytest plugin supports synchronous tests with async fixtures, though certain limitations apply.

Applying a mark to a fixture function

Deprecated since version 7.4.

Removed in version 9.0.

Applying a mark to a fixture function never had any effect, but it is a common user error.

```
@pytest.mark.usefixtures("clean_database")
@pytest.fixture
def user() -> User: ...
```

Users expected in this case that the `usefixtures` mark would have its intended effect of using the `clean_database` fixture when `user` was invoked, when in fact it has no effect at all.

Now pytest will issue a warning when it encounters this problem, and will raise an error in the future versions.

`py.path.local` arguments for hooks replaced with `pathlib.Path`

Deprecated since version 7.0.

Removed in version 9.0.

In order to support the transition from `py.path.local` to `pathlib`, the following hooks now receive additional arguments:

- *`pytest_ignore_collect(collection_path: pathlib.Path)`* as equivalent to `path`
- *`pytest_collect_file(file_path: pathlib.Path)`* as equivalent to `path`
- *`pytest_pycollect_makemodule(module_path: pathlib.Path)`* as equivalent to `path`
- *`pytest_report_header(start_path: pathlib.Path)`* as equivalent to `startdir`
- *`pytest_report_collectionfinish(start_path: pathlib.Path)`* as equivalent to `startdir`

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.

Note

The name of the *`Node`* arguments and attributes, *outlined above* (the new attribute being `path`) is **the opposite** of the situation for hooks (the old argument being `path`).

This is an unfortunate artifact due to historical reasons, which should be resolved in future versions as we slowly get rid of the py dependency (see #9283 for a longer discussion).

`yield` tests

Removed in version 4.0: `yield` tests `xfail`.

Removed in version 8.4: `yield` tests raise a collection error.

pytest no longer supports `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 in 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
```

Support for tests written for nose

Deprecated since version 7.2.

Removed in version 8.0.

Support for running tests written for nose is now deprecated.

`nose` has been in maintenance mode-only for years, and maintaining the plugin is not trivial as it spills over the code base (see #9886 for more details).

setup/teardown

One thing that might catch users by surprise is that plain `setup` and `teardown` methods are not pytest native, they are in fact part of the `nose` support.

```
class Test:
    def setup(self):
        self.resource = make_resource()

    def teardown(self):
        self.resource.close()

    def test_foo(self): ...

    def test_bar(self): ...
```

Native pytest support uses `setup_method` and `teardown_method` (see *Method and function level setup/teardown*), so the above should be changed to:

```
class Test:
    def setup_method(self):
        self.resource = make_resource()

    def teardown_method(self):
        self.resource.close()

    def test_foo(self): ...

    def test_bar(self): ...
```

This is easy to do in an entire code base by doing a simple find/replace.

@with_setup

Code using @with_setup such as this:

```
from nose.tools import with_setup


def setup_some_resource(): ...


def teardown_some_resource(): ...
```

(continues on next page)

(continued from previous page)

```
@with_setup(setup_some_resource, teardown_some_resource)
def test_foo(): ...
```

Will also need to be ported to a supported pytest style. One way to do it is using a fixture:

```
import pytest


def setup_some_resource(): ...


def teardown_some_resource(): ...


@pytest.fixture
def some_resource():
    setup_some_resource()
    yield
    teardown_some_resource()


def test_foo(some_resource): ...
```

The `compat_co_firstlineno` attribute

Nose inspects this attribute on function objects to allow overriding the function's inferred line number. Pytest no longer respects this attribute.

Passing `msg=` to `pytest.skip`, `pytest.fail` or `pytest.exit`

Deprecated since version 7.0.

Removed in version 8.0.

Passing the keyword argument `msg` to *`pytest.skip()`*, *`pytest.fail()`* or *`pytest.exit()`* is now deprecated and `reason` should be used instead. This change is to bring consistency between these functions and the `@pytest.mark.skip` and `@pytest.mark.xfail` markers which already accept a `reason` argument.

```
def test_fail_example():
    # old
    pytest.fail(msg="foo")
    # new
    pytest.fail(reason="bar")


def test_skip_example():
    # old
    pytest.skip(msg="foo")
    # new
    pytest.skip(reason="bar")


def test_exit_example():
```

(continues on next page)

(continued from previous page)

```
    # old
    pytest.exit(msg="foo")
    # new
    pytest.exit(reason="bar")
```

The `pytest.Instance` collector

Removed in version 7.0.

The `pytest.Instance` collector type has been removed.

Previously, Python test methods were collected as *`Class`* -> `Instance` -> *`Function`*. Now *`Class`* collects the test methods directly.

Most plugins which reference `Instance` do so in order to ignore or skip it, using a check such as `if isinstance(node, Instance): return`. Such plugins should simply remove consideration of `Instance` on pytest>=7. However, to keep such uses working, a dummy type has been instanced in `pytest.Instance` and `_pytest.python.Instance`, and importing it emits a deprecation warning. This was removed in pytest 8.

Using `pytest.warns(None)`

Deprecated since version 7.0.

Removed in version 8.0.

`pytest.warns(None)` is now deprecated because it was frequently misused. Its correct usage was checking that the code emits at least one warning of any type - like `pytest.warns()` or `pytest.warns(Warning)`.

See *Additional use cases of warnings in tests* for examples.

Backward compatibilities in `Parser.addoption`

Deprecated since version 2.4.

Removed in version 8.0.

Several behaviors of *`Parser.addoption`* are now removed in pytest 8 (deprecated since pytest 2.4.0):

- `parser.addoption(..., help=".. %default ..")` - use `%(default)s` instead.
- `parser.addoption(..., type="int/string/float/complex")` - use `type=int` etc. instead.

The `--strict` command-line option (reintroduced)

Deprecated since version 6.2.

Changed in version 9.0.

The `--strict` command-line option had been deprecated in favor of `--strict-markers`, which better conveys what the option does.

In version 8.1, we accidentally un-deprecated `--strict`.

In version 9.0, we changed `--strict` to make it set the new *`strict`* configuration option. It now enables all strictness related options (including *`strict_markers`*).

Implementing the `pytest_cmdline_preparse` hook

Deprecated since version 7.0.

Removed in version 8.0.

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: ...
```

Collection changes in pytest 8

Added a new *pytest.Directory* base collection node, which all collector nodes for filesystem directories are expected to subclass. This is analogous to the existing *pytest.File* for file nodes.

Changed *pytest.Package* to be a subclass of *pytest.Directory*. A `Package` represents a filesystem directory which is a Python package, i.e. contains an `__init__.py` file.

pytest.Package now only collects files in its own directory; previously it collected recursively. Sub-directories are collected as sub-collector nodes, thus creating a collection tree which mirrors the filesystem hierarchy.

session.name is now `""`; previously it was the rootdir directory name. This matches *session.nodeid* which has always been `""`.

Added a new *pytest.Dir* concrete collection node, a subclass of *pytest.Directory*. This node represents a filesystem directory, which is not a *pytest.Package*, i.e. does not contain an `__init__.py` file. Similarly to `Package`, it only collects the files in its own directory, while collecting sub-directories as sub-collector nodes.

Files and directories are now collected in alphabetical order jointly, unless changed by a plugin. Previously, files were collected before directories.

The collection tree now contains directories/packages up to the *rootdir*, for initial arguments that are found within the rootdir. For files outside the rootdir, only the immediate directory/package is collected – note however that collecting from outside the rootdir is discouraged.

As an example, given the following filesystem tree:

```
myroot/
    pytest.ini
    top/
    ├── aaa
    │   └── test_aaa.py
    ├── test_a.py
    ├── test_b
    │   ├── __init__.py
    │   └── test_b.py
    ├── test_c.py
    └── zzz
        ├── __init__.py
        └── test_zzz.py
```

the collection tree, as shown by `pytest --collect-only top/` but with the otherwise-hidden *Session* node added for clarity, is now the following:

```
<Session>
  <Dir myroot>
    <Dir top>
      <Dir aaa>
        <Module test_aaa.py>
          <Function test_it>
      <Module test_a.py>
        <Function test_it>
      <Package test_b>
        <Module test_b.py>
          <Function test_it>
      <Module test_c.py>
        <Function test_it>
      <Package zzz>
        <Module test_zzz.py>
          <Function test_it>
```

Previously, it was:

```
<Session>
  <Module top/test_a.py>
    <Function test_it>
  <Module top/test_c.py>
    <Function test_it>
  <Module top/aaa/test_aaa.py>
    <Function test_it>
  <Package test_b>
    <Module test_b.py>
      <Function test_it>
  <Package zzz>
    <Module test_zzz.py>
      <Function test_it>
```

Code/plugins which rely on a specific shape of the collection tree might need to update.

pytest.Package is no longer a *pytest.Module* or *pytest.File*

Changed in version 8.0.

The `Package` collector node designates a Python package, that is, a directory with an `__init__.py` file. Previously `Package` was a subtype of `pytest.Module` (which represents a single Python module), the module being the `__init__.py` file. This has been deemed a design mistake (see #11137 and #7777 for details).

The `path` property of `Package` nodes now points to the package directory instead of the `__init__.py` file.

Note that a `Module` node for `__init__.py` (which is not a `Package`) may still exist, if it is picked up during collection (e.g. if you configured *python_files* to include `__init__.py` files).

Collecting `__init__.py` files no longer collects package

Removed in version 8.0.

Running `pytest pkg/__init__.py` now collects the `pkg/__init__.py` file (module) only. Previously, it collected the entire `pkg` package, including other test files in the directory, but excluding tests in the `__init__.py` file itself (unless *python_files* was changed to allow `__init__.py` file).

To collect the entire package, specify just the directory: `pytest pkg`.

The `pytest.collect` module

Deprecated since version 6.0.

Removed in version 7.0.

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_warning_captured` hook

Deprecated since version 6.0.

Removed in version 7.0.

This hook has an `item` parameter which cannot be serialized by `pytest-xdist`.

Use the `pytest_warning_recorded` hook instead, which replaces the `item` parameter by a `nodeid` parameter.

The `pytest._fillfuncargs` function

Deprecated since version 6.0.

Removed in version 7.0.

This function was kept for backward compatibility with an older plugin.

Its 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.

`--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` hook 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 configuration file:

```
[pytest]
junit_family = "xunit2"
```

```
[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"
```

```
[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 `--junit-xml` 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 #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",
    [
```

(continues on next page)

(continued from previous page)

```
        (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 ...
```

(continues on next page)

(continued from previous page)

```
@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 ...


@pytest.fixture(name="cell")
def cell_fixture():
    return cell()
```

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()
```

5.6 Contributing

Contributions are highly welcomed and appreciated. Every little bit of help counts, so do not hesitate!

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. See also the "good first issue" issues that are friendly to new contributors.

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:
```

(continues on next page)

(continued from previous page)

```
        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

Development of the pytest core, support code, and some plugins 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 posting a new topic in the pytest-dev GitHub Discussions 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 AI/LLM-Assisted Contributions Policy

We welcome contributions from all developers, including those who use AI/LLM tools as part of their workflow. However, we have requirements to protect the time and effort of our reviewers:

Purely agentic contributions are not accepted. Pull requests that are entirely generated by AI agents, with no meaningful human review, understanding, or oversight, will be closed. Every contribution must demonstrate that a human has reviewed, understood, and taken responsibility for the changes. If you submit it, you own it.

You are responsible for your contribution. Regardless of how the code was produced, the person submitting a pull request must understand the changes and be able to respond to review feedback. If a reviewer asks questions or requests changes, they expect to interact with someone who can engage substantively, not an automated loop replaying prompts.

Credit AI tools via attribution. If AI agents helped produce your code or commits, consider adding `Co-authored-by` trailers to your commit messages to credit them. This is not required, but helps reviewers set expectations and is appreciated.

Respect maintainer time. Submitting low-effort AI-generated pull requests that waste reviewer time may result in a ban from the project. Our maintainers are volunteers, and contributions should reflect genuine engagement with the project.

Context

With the advent of unsupervised agentic tools like OpenClaw, there has been a rise in low-quality contributions where an agent produces a large number of low-quality pull requests. Oftentimes this can look similar to a human beginner with new access to tools and trying to learn, but in practice it is usually an unsupervised agentic tool generating changes without meaningful human review.

When a human contributor is learning, we are glad to invest time to help, give feedback, and guide them in the right direction. With fully agentic, unsupervised tools, that same review effort does not support anyone's learning or growth. Instead, it diverts limited maintainer time away from improving the project and supporting engaged contributors.

There is also an asymmetry at play: someone is prioritizing what we review without making an equivalent investment of time or effort. When a contributor works on an issue themselves, they invest real time, effectively earning influence over what the project focuses on. Unsupervised agentic contributions expect to set that priority at near-zero cost to the sender, while shifting the entire burden onto maintainers.

Fully agentic contributions invert the intended benefit of these tools: rather than saving time, they create avoidable review and triage work. There is no accountable human author thoughtfully iterating on feedback, only automated output driven by prompts.

From our own experience using coding agents, we know they must be carefully prompted, supervised, and checked by humans. Even modern models can make serious mistakes when operating at framework or tooling level, and those mistakes can be subtle and time-consuming to diagnose.

Running such tools unsupervised on open-source projects imposes this cost on maintainers and other contributors without their consent. Our goal with this policy is to set clear expectations, protect reviewer time, and ensure that contributions remain collaborative, respectful, and sustainable.

5.6.8 Preparing Pull Requests

Short version

1. Fork the repository.
2. Fetch tags from upstream if necessary (if you cloned only main `git fetch --tags https://github.com/pytest-dev/pytest`).

3. Enable and install pre-commit to ensure style-guides and code checks are followed.
4. Follow PEP-8 for naming.
5. Tests are run using `tox`:

```
tox -e linting,py313
```

   The test environments above are usually enough to cover most cases locally.

6. Write a `changelog` entry: `changelog/2574.bugfix.rst`, use issue id number and one of `feature`, `improvement`, `bugfix`, `doc`, `deprecation`, `breaking`, `vendor`, `packaging`, `contrib`, or `misc` for the issue type.
7. 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
$ git fetch --tags https://github.com/pytest-dev/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.

   You will need the tags to test locally, so be sure you have the tags from the main repository. If you suspect you don't, set the main repository as upstream and fetch the tags:

```
$ git remote add upstream https://github.com/pytest-dev/pytest
$ git fetch upstream --tags
```

   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 a supported Python version available in your system. Now running tests is as simple as issuing this command:

   ```
   $ tox -e linting,py
   ```

   This command will run tests via the "tox" tool against your default Python version 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 3.13 and pass options to pytest (e.g. enter pdb on failure) you can do:

   ```
   $ tox -e py313 -- --pdb
   ```

   Or to only run tests in a particular test module on Python 3.12:

   ```
   $ tox -e py312 -- 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 `dev` extra:

   ```
   $ python3 -m venv .venv
   $ source .venv/bin/activate  # Linux
   $ .venv/Scripts/activate.bat  # Windows
   $ pip install -e ".[dev]"
   ```

   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`, `packaging`, `contrib`, or `misc`. 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
    ```

(continues on next page)

(continued from previous page)

```
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.

5.6.9 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.

5.6.10 Merge/squash guidelines

When a PR is approved and ready to be integrated to the `main` branch, one has the option to *merge* the commits unchanged, or *squash* all the commits into a single commit.

Here are some guidelines on how to proceed, based on examples of a single PR commit history:

1. Miscellaneous commits:

    - `Implement X`

- `Fix test_a`
- `Add myself to AUTHORS`
- `fixup! Fix test_a`
- `Update tests/test_integration.py`
- `Merge origin/main into PR branch`
- `Update tests/test_integration.py`

In this case, prefer to use the **Squash** merge strategy: the commit history is a bit messy (not in a derogatory way, often one just commits changes because they know the changes will eventually be squashed together), so squashing everything into a single commit is best. You must clean up the commit message, making sure it contains useful details.

2. Separate commits related to the same topic:

   - `Implement X`
   - `Add myself to AUTHORS`
   - `Update CHANGELOG for X`

   In this case, prefer to use the **Squash** merge strategy: while the commit history is not "messy" as in the example above, the individual commits do not bring much value overall, specially when looking at the changes a few months/years down the line.

3. Separate commits, each with their own topic (refactorings, renames, etc), but still have a larger topic/purpose.

   - `Refactor class X in preparation for feature Y`
   - `Remove unused method`
   - `Implement feature Y`

   In this case, prefer to use the **Merge** strategy: each commit is valuable on its own, even if they serve a common topic overall. Looking at the history later, it is useful to have the removal of the unused method separately on its own commit, along with more information (such as how it became unused in the first place).

4. Separate commits, each with their own topic, but without a larger topic/purpose other than improve the code base (using more modern techniques, improve typing, removing clutter, etc).

   - `Improve internal names in X`
   - `Add type annotations to Y`
   - `Remove unnecessary dict access`
   - `Remove unreachable code due to EOL Python`

   In this case, prefer to use the **Merge** strategy: each commit is valuable on its own, and the information on each is valuable in the long term.

As mentioned, those are overall guidelines, not rules cast in stone. This topic was discussed in #12633.

Backport PRs (as those created automatically from a `backport` label) should always be **squashed**, as they preserve the original PR author.

5.6.11 Backporting bug fixes for the next patch release

Pytest makes a 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 in 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.

Automatic method:

Add a `backport 1.2.x` label to the PR you want to backport. This will create a backport PR against the `1.2.x` branch.

Manual method:

1. `git checkout origin/1.2.x -b backport-XXXX` # use the main PR number here
2. Locate the merge commit on the PR, in the *merged* message, for example:

   nicoddemus merged commit 0f8b462 into pytest-dev:main

3. `git cherry-pick -x -m1 REVISION` # use the revision you found above (`0f8b462`).
4. 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* or *backport x.x.x* labels, 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.

Backports should be **squashed** (rather than **merged**), as doing so preserves the original PR author correctly.

5.6.12 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 to decide 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, we should acknowledge the time, effort, and interest demonstrated by the person who 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>

5.6.13 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 requester 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 in our 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 provides 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 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

5.11.1 Web

- pytest issue tracker to report bugs or suggest features.
- pytest discussions at GitHub for general questions.
- 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.

5.11.2 Chat

- pytest discord server for pytest development visibility and general assistance.
- `#pytest` on irc.libera.chat IRC channel for random questions (using an IRC client, or via webchat)
- `#pytest` on Matrix.

5.11.3 Microblogging

- Bluesky: @pytest.org
- Mastodon: @pytest@fosstodon.org
- Twitter/X: @pytestdotorg

5.11.4 Mail

- Testing In Python: a mailing list for Python testing tools and discussion.
- Mail to core@pytest.org for topics that cannot be discussed in public. Mails sent there will be distributed among the members in the pytest core team, who can also be contacted individually:
  - Bruno Oliveira (@nicoddemus, bruno@pytest.org)
  - Freya Bruhin (@The-Compiler, freya@pytest.org)
  - Pierre Sassoulas (@Pierre-Sassoulas, pierre@pytest.org)
  - Ran Benita (@bluetech, ran@pytest.org)
  - Ronny Pfannschmidt (@RonnyPfannschmidt, ronny@pytest.org)
  - Zac Hatfield-Dodds (@Zac-HD, zac@pytest.org)

5.11.5 Other

- The *contribution guide* for help on submitting pull requests to GitHub.
- Freya Bruhin (@The-Compiler) offers pytest professional teaching and consulting via Bruhin Software.

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")
```

(continues on next page)

(continued from previous page)

```
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" using `request.config` is:

```
@pytest.fixture(autouse=True)
def skip_if_no_db(request):
    if not request.config.getoption("--db", default=False):
        pytest.skip("--db was not specified")


def test_function():
    pass
```

Note

`pytest.config` was removed in pytest 5.0. Use `request.config` (via the `request` fixture) or the `pytestconfig` fixture instead. See *pytest.config global* for details.

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 has 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).
- Python Testing with pytest, Second Edition, by Brian Okken (2022).

5.14.2 Talks and blog postings

- Training: pytest - simple, rapid and fun testing with Python, Freya Bruhin, PyConDE 2022
- pytest: Simple, rapid and fun testing with Python, (@ 4:22:32), Freya Bruhin, WeAreDevelopers World Congress 2021
- Webinar: pytest: Test Driven Development für Python (German), Freya 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, Freya Bruhin, PyConDE 2019
- Abridged metaprogramming classics - this episode: pytest, Oliver Bestwalter, PyConDE 2019 (repository, recording)
- Testing PySide/PyQt code easily using the pytest framework, Freya 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, Freya 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 Laugher (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