pytest

Version 9.2 · verified 2026-10-29

A gradual walk through pytest 9 — plain asserts, running and selecting tests, fixtures and scope, parametrize, markers, built-in fixtures, config.

Video walkthrough — scan the code to watch.

Contents

Mental model 2

Writing tests 2

Running and selecting 3

Exceptions, warnings and floats 4

Fixtures 5

Scope, conftest.py and autouse 6

Parametrize 7

Markers: skip, xfail, custom 9

Built-in fixtures 10

Configuration and plugins 11

Further reading 12

Mental model

A pytest test is a plain function whose name starts with `test`, and a failing `assert` is the whole reporting mechanism — pytest rewrites the assert at import time so the failure shows every intermediate value. Everything a test needs (data, clients, temp dirs, patches) is a **fixture**: you ask for it by naming it as a parameter, and pytest builds it, shares it for its scope, and tears it down afterwards.

Writing tests

Collected	Rule
Files	`test_*.py` or `*_test.py`
Functions	`test` prefix, at module level
Classes	`Test` prefix, no `__init__`
Methods	`test` prefix inside a `Test` class

Collection starts from `testpaths` (if configured) or the current directory. No base class and no `self.assertEqual` — a bare `assert` is enough, and pytest prints the operands when it fails.

```
# test_cart.py
def total(items):
    return sum(p * q for p, q in items)


def test_total():
    assert total([(2, 3), (5, 1)]) == 11
```

```
class TestEmpty:
    def test_zero(self):
        assert total([]) == 0
```

> **Gotcha:** Assert rewriting applies only to test modules, `conftest.py` and plugins. An `assert` inside a helper module fails with a bare `AssertionError` and no detail — call `pytest.register_assert_rewrite("helpers")` in the root `conftest.py` before importing it.

Running and selecting

Command	Effect
`pytest test_a.py::test_x`	One test, by node ID
`pytest -k "cart and not slow"`	Match names and markers
`pytest -x` / `--maxfail=3`	Stop at first / third failure
`pytest --lf` / `--ff`	Rerun last failures / run them first
`pytest -s`	Show `print` output (no capture)
`pytest -ra --durations=5`	Why things skipped; slowest five

A node ID is `path::Class::test[param]`, exactly as printed in the report, so you can copy it from a failure line. Add `--pdb` to drop into the debugger at the failing line. `-q` and `-v` adjust detail.

```
pytest -x --lf -q
pytest "tests/test_cart.py::test_total"
pytest -m "not slow" -n auto   # -n needs xdist
```

> **Gotcha:** Exit code `5` means *no tests were collected*, and CI treats it as a failure. A typo in `testpaths`, or a file named `cart_tests.py` (the pattern is `*_test.py`), lands you there.

Exceptions, warnings and floats

Helper	Checks
`pytest.raises(E, match=r"...")`	Block raises `E`; message regex
`pytest.warns(W)`	Block emits warning `W`
`x == pytest.approx(y)`	Float equal within tolerance

`match` is applied with `re.search`, so a substring works but regex metacharacters such as `(` or `.` need escaping. The context manager returns an `ExceptionInfo`; its `.value` is the raised exception.

```
import pytest


def test_bad_qty():
    with pytest.raises(
        ValueError, match="invalid literal"
    ) as e:
        int("qty")
```

```
    assert "'qty'" in str(e.value)


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

> **Gotcha:** Any line after the raising call inside the `with` block never runs. Keep only the call that should raise inside it, and put follow-up asserts after the block.

Fixtures

Form	Meaning
`@pytest.fixture`	Declares a fixture
`def test_x(db):`	Requests the fixture `db`
`return value`	Provides the value, no teardown
`yield value`	Code after `yield` is teardown

A fixture can request other fixtures the same way tests do. Within one test each fixture runs once, however many times it is requested, so a shared dependency is built only once.

```
import pytest


@pytest.fixture
def db():
    conn = {"open": True}
```

```
    yield conn              # test runs here
    conn["open"] = False    # teardown


@pytest.fixture
def user(db):
    db["user"] = "ada"
    return "ada"


def test_user(db, user):
    assert db["user"] == user
```

> **Gotcha:** Teardown after `yield` runs even when the test fails, but not when the fixture itself raises *before* the `yield`. Put the risky setup step first so nothing is left half-created.

Scope, conftest.py and autouse

`scope=`	Built once per
`"function"` (default)	Test
`"class"` / `"module"`	Class / test file
`"package"`	Directory package
`"session"`	Whole run

Fixtures in a `conftest.py` are visible to every test in that directory and below, with no import. `autouse=True` applies a fixture to every test in its

reach without being requested — useful for resetting global state, dangerous for anything expensive.

```python
# tests/conftest.py
import pytest


@pytest.fixture(scope="session")
def config():
    return {"api": "http://test"}


@pytest.fixture(autouse=True)
def reset_cache():
    yield
    # clear module-level caches here
```

> **Gotcha:** A fixture may only request fixtures of the same or a wider scope. A `session` fixture that asks for a function-scoped one (including `tmp_path` or `monkeypatch`) fails with `ScopeMismatch` — use `tmp_path_factory` instead.

Parametrize

Form	Use
`parametrize("a,b", [(1, 2)])`	Run once per tuple
`ids=["small"]`	Readable test IDs (keyword-only)
`pytest.param(..., marks=m)`	Mark a single case

Form	Use
Stack two decorators	Cartesian product

Each case becomes its own test with its own node ID, so failures are reported — and rerun with `--lf` — individually. When the cases are only known at run time, use the `subtests` fixture (pytest 9.0+): `with subtests.test(i=i):` reports each block separately.

```
import pytest


@pytest.mark.parametrize(
    "raw, want",
    [
        ("1", 1),
        (" 2 ", 2),
        pytest.param(
            "x", None, marks=pytest.mark.xfail
        ),
    ],
    ids=["plain", "spaces", "junk"],
)
def test_parse(raw, want):
    assert int(raw) == want
```

> **Gotcha:** Since pytest 9.2, `ids`, `indirect` and `scope` are keyword-only. `parametrize("a", "b", [...])` now raises `TypeError` — wrap multiple names in one string: `"a, b"`.

Markers: skip, xfail, custom

Marker	Effect
`skip(reason=...)`	Never runs
`skipif(cond, reason=...)`	Skips when `cond` is true
`xfail(strict=True)`	Expected failure; a pass fails the run
`@pytest.mark.slow`	Custom tag, select with `-m slow`

Register custom markers in config so typos are caught. The `-m` flag takes a boolean expression over marker names, such as `-m "slow and not db"`.

```
import sys
import pytest


@pytest.mark.skipif(
    sys.platform == "win32", reason="POSIX"
)
def test_root():
    assert "/".startswith("/")


@pytest.mark.slow
```

```
def test_report():
    assert sum(range(10**6)) > 0
```

> **Warning:** Without `strict_markers`, a misspelled marker such as `@pytest.mark.slwo` only warns, and `-m "not slow"` silently runs the test. Enable `strict = true` (below) to make it an error.

Built-in fixtures

Fixture	Gives you
`tmp_path`	Fresh `pathlib.Path` directory per test
`monkeypatch`	Patch attrs, env, dict items; auto-undone
`capsys`	Captured `stdout` / `stderr`
`caplog`	Captured `logging` records

All of these clean up after the test on their own: patches are reverted and environment variables restored. Run `pytest --fixtures` to list every fixture available in a directory.

```
import os


def greet():
    print("hi", os.environ["USER_NAME"])


def test_greet(monkeypatch, capsys, tmp_path):
    monkeypatch.setenv("USER_NAME", "ada")
```

```
    greet()
    assert capsys.readouterr().out == "hi ada\n"
    (tmp_path / "a.txt").write_text("x")
    assert (tmp_path / "a.txt").exists()
```

> **Gotcha:** Patch the name where it is *looked up*, not where it is defined. After `from pay import charge`, the module `shop` holds its own `charge` — patch `shop.charge`, not `pay.charge`.

Configuration and plugins

File	Table
`pytest.toml` (9.0+)	`[pytest]` — native TOML types
`pyproject.toml`	`[tool.pytest]` (9.0+, native types)
`pyproject.toml`	`[tool.pytest.ini_options]` (strings)
`pytest.ini`	`[pytest]`

`strict = true` turns on `strict_config`, `strict_markers`, `strict_parametrization_ids` and `strict_xfail` — and any strictness option added later, so pin pytest when you use it. The docs recommend `--import-mode=importlib` for new projects.

```
# pyproject.toml
[tool.pytest]
minversion = "9.0"
testpaths = ["tests"]
```

```
addopts = ["-ra", "--import-mode=importlib"]
strict = true
markers = ["slow: long-running tests"]
```

Plugins add what core leaves out: `pytest-xdist` (`-n auto`), `pytest-cov` (`--cov`), `pytest-asyncio` (`async def` tests).

> **Gotcha:** pytest reads exactly one config file. With both a `pytest.ini` and a `[tool.pytest]` table, the `.ini` wins and the TOML is ignored — pytest 9 at least prints a warning now.

Further reading

- pytest — How to use fixtures
- pytest — How to parametrize
- pytest — Configuration
- pytest — Good integration practices
- pytest — API reference
- pytest changelog