pytest Python test runner

START HERE

`pytest`
Run all tests

`pytest --lf`
Rerun last failures

`pytest <path>::<test>`
Run one test

`pytest -v`
One line per test

`pytest -k "<expr>"`
Filter by name

`pytest -s`
Show print output

`pytest -x`
Stop at first failure

`pytest --pdb`
Debug on failure

Collect find tests → `filter -k -m` → **Setup** fixtures → `run` → **Call** test body → `teardown` → **Report** pass / fail

CONCEPTS

Files named test_*.py or *_test.py, functions named test_*, are collected; plain assert is all you need.

Fixtures in conftest.py are shared with every test below that directory, with no import.

python -m pytest also puts the current directory on sys.path, which helps with import errors.

SELECT TESTS

`pytest <dir> / <file.py>`
Run a directory or file

`pytest <file.py>::<test>`
Run one test function

`pytest <file.py>::<Class>::<method>`
Run one method of a class

`pytest "<file.py>::<test>[<id>]"`
Run one parametrized case

`pytest -k "a and not b"`
Select by name expression
Case-insensitive substring match on test, class and module names.

`pytest -m <marker>`
Select by marker

`pytest -m "not slow"`
Skip a marker

`pytest --collect-only -q`
List test ids without running

`pytest --deselect <nodeid>`
Drop one test by id

`pytest --ignore=<path>`
Skip a path in collection

RUN CONTROL

`pytest -x`
Stop at first failure

`pytest --maxfail=<n>`
Stop after n failures

`pytest --lf`
Rerun only last failures

`pytest --ff`
Failures first, then the rest

`pytest --sw`
Stop, resume at that test next run

`pytest --durations=<n>`
Show n slowest tests

`pytest -W error`
Turn warnings into errors

OUTPUT

`pytest -v / -vv`
More detail per test

`pytest -q`
Quieter output

`pytest -s`
Do not capture stdout

`pytest --tb=short`
Compact tracebacks
Styles: auto, long, short, line, native, no.

`pytest -l`
Show locals in tracebacks

`pytest -ra`
Summary of all but passed
Letters: f E s x X p P a A; default is fE.

`pytest --junitxml=<file>`
Write a JUnit XML report

`pytest --log-cli-level=INFO`
Live log output

DEBUGGING

`pytest --pdb`
Open pdb at a failure

`pytest --trace`
Break at the start of each test

`breakpoint()`
Drop into the debugger here
Capturing is paused while the prompt is active.

`pytest --setup-show`
Show fixture setup and teardown

`pytest --fixtures`
List available fixtures

`pytest --markers`
List registered markers

FIXTURES

`@pytest.fixture`
Declare a fixture

`def test_x(<fixture>):`
Request a fixture by argument name

`yield <value>`
Setup before, teardown after

`@pytest.fixture(scope="<s>")`
Share across tests
Scopes: function (default), class, module, package, session.

`@pytest.fixture(autouse=True)`
Apply to every test in scope

`@pytest.fixture(params=<list>)`
Run tests once per value

FIXTURES (cont.)

`request.param`
Current value of a param fixture

`@pytest.mark.usefixtures`
Use fixtures without arguments

`conftest.py`
Directory-wide shared fixtures

Fixture with teardown

```
@pytest.fixture
def db():
    conn = connect()
    yield conn
    conn.close()
```

PARAMETRIZE AND MARKS

`@pytest.mark.parametrize("a,b", [(1, 2), (3, 4)])`
Run a test per data row

`pytest.param(<v>, id="<name>")`
Name one case

`pytest.param(<v>, marks=pytest.mark.xfail)`
Mark a single case

`@pytest.mark.skip`
Always skip (reason="<why>")

`@pytest.mark.skipif(<cond>)`
Skip on a condition (add reason=)

`@pytest.mark.xfail`
Expected to fail (reason="<why>")

`pytest.skip("<why>")`
Skip from inside a test

`pytestmark = pytest.mark.<name>`
Mark a whole module

Stacked parametrize decorators produce every combination of their values.

ASSERTIONS AND BUILT-INS

`assert <expr>`
Plain assert with rich diffs

`with pytest.raises(<Exc>):`
Expect an exception

`pytest.raises(<Exc>, match="<re>")`
Check the message by regex

`pytest.approx(<x>)`
Compare floats with tolerance

`tmp_path`
Temporary directory per test

`monkeypatch`
Patch attrs, env, dicts

ASSERTIONS AND BUILT-INS (cont.)

`capsys`
Capture stdout and stderr

`caplog`
Inspect log records

`monkeypatch.setenv("<K>", "<v>")`
Set an env var for one test

`monkeypatch.setattr(<obj>, "<name>", <val>)`
Replace an attribute

CONFIG

`[tool.pytest]`
pyproject table (pytest 9+)
Older versions use [tool.pytest.ini_options]. pytest.toml and pytest.ini also work.

`testpaths = ["tests"]`
Where to look by default

`addopts = ["-ra", "-q"]`
Options added to every run

`markers = ["slow: slow tests"]`
Register custom markers

`--strict-markers`
Error on unregistered markers

`pythonpath = ["src"]`
Extra import paths

RECIPES

Fast failure loop

```
# stop early, then rerun the failures
pytest -x --tb=short
pytest --lf -x -l
```

Skip slow tests by default

```
# pyproject.toml
[tool.pytest]
markers = ["slow: takes long"]
addopts = ["-m", "not slow"]

# run them explicitly
pytest -m slow
```

Table-driven test

```
@pytest.mark.parametrize("s,n",
[
    ("1", 1),
    (" 2 ", 2),
    pytest.param("x", 0,
id="bad"),
])
def test_parse(s, n):
    assert parse(s) == n
```

Test with temp files and env

```
def test_cfg(tmp_path,
monkeypatch):
    f = tmp_path / "a.toml"
    f.write_text("a=1")
    monkeypatch.chdir(tmp_path)
    assert load(f).a == 1
```