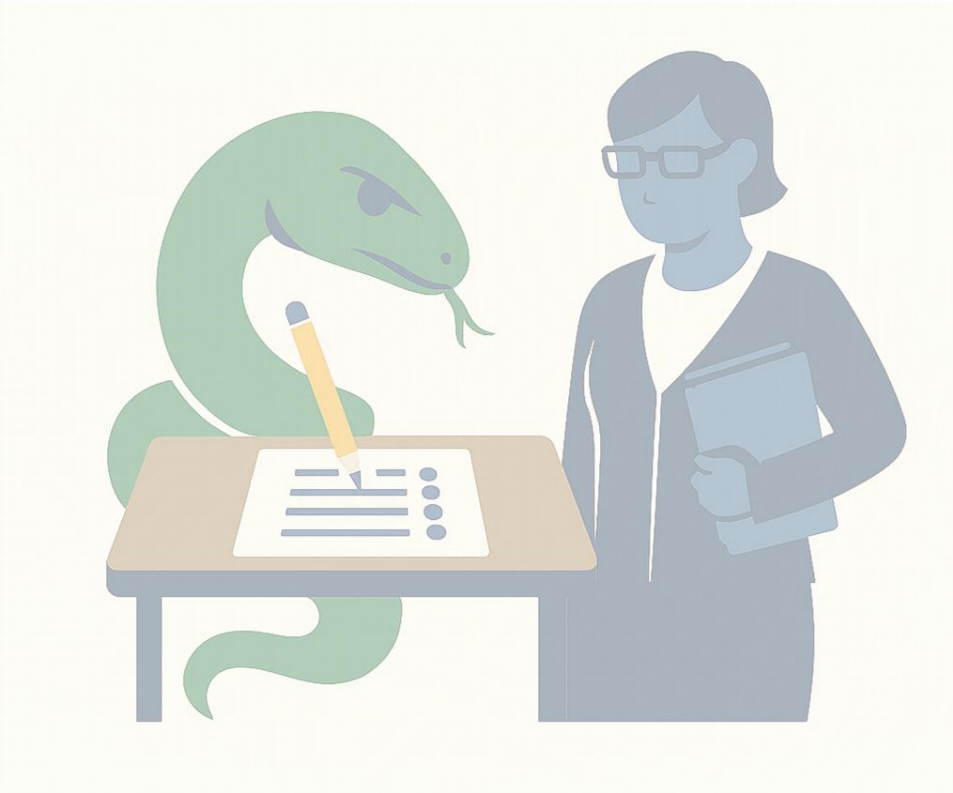


pytest

VAIRA MUTHU THANGAVEL

AGENDA

- Intro & Motivation
- pytest Basics & First Tests
- Fixtures & Parametrization
- Markers, Skips, And Plugins
- Parallel, Coverage
- Advanced Topics (Hypothesis, Hooks, Debugging)
- Walkthrough
- Q&A / Resources

WHY TEST?

- **Confidence:** Tests Catch Regressions Early.
- **Documentation:** Tests Document Intended Behavior.
- **Design:** Tests Encourage Modular, Testable Code.
- **Automation:** Run In CI To Enforce Quality.

What is pytest?

- **Simple:** Minimal ceremony to write tests.
- **Powerful:** Fixtures, markers, plugins, and hooks.
- **Community:** Large ecosystem (plugins like xdist, cov, mock, hypothesis).
- **Readable:** Uses plain assert statements with good failure introspection.

Installation & Basic Setup

- **Install:** pip install pytest
- **Optional extras:** pip install pytest-cov pytest-xdist
- **Project files:** pytest.ini or pyproject.toml to configure markers and options
- **Minimal pytest.ini:**
  - pytest.ini content:

```
[pytest]                                  → pytest component
minversion = 6.0                          → pytest version
addopts = -ra -q                          → report all test outcomes except passed(ra), less noise(q)
markers =
    slow: marks tests as slow (deselect with -m "not slow") → pytest –m “not slow”
    integration: tests hitting DB/network
```

Test Discovery & Naming

- **Files**: test_*.py or *_test.py
- **Test functions**: def test_something():
- **Test classes**: class names should start with Test (no init)
- **Test methods**: names start with test_

First Test Example

- No boilerplate classes required
- Plain assert gives helpful diffs

Example

```
def add(a, b):
    return a + b
def test_add():
    assert add(2, 3) == 5
    assert add(-1, 1) == 0
```

Run test with -q

Assertions & Failure Output

- Use plain assert expressions
- pytest rewrites expressions to show left/right values
- Use assert x == y, "optional message" sparingly

Example fail output snippet:

- show when assert result == expected prints both values.

Running Tests & Common Options

- **Commands:**
- pytest — run all tests
- pytest -q — quieter
- pytest -v — verbose
- pytest -k "substring or and not" — select tests by name expression
- pytest -m marker — run tests with marker
- pytest -x — stop after first failure
- pytest --maxfail=3 — stop after 3 failures
- pytest -s — disable capture (print to stdout)
- pytest -q --disable-warnings

-k and -m are extremely useful for iterative development

Combine with -q and -k for fast feedback

Fixtures: Introduction

- ***@pytest.fixture*** provides setup/teardown reusable components
- Inject fixtures by naming them in test function signature
- Scopes: function, module, class, package, session
- autouse=True for implicit fixtures
- Example fixture:

```
@pytest.fixture
def sample_dict():
    return {"a": 1, "b”: 2}
```

Fixtures: Lifecycle & Yield

- Use yield in fixture to run teardown after yield
- setup → run test → teardown
- **Example teardown pattern:**

```
@pytest.fixture
def resource():
    r = allocate()  → setup
    yield r
    r.close()       → teardown
```

- Scoping trade-offs: large scope speeds tests but increases coupling

Parametrization

- *@pytest.mark.parametrize("input,expected", [(1,2),(2,3)])*
- Parametrized tests produce separate test ids
- Use ids= to give readable names
- indirect=True to pass parameters to fixtures

Example:

```
@pytest.mark.parametrize("x,y", [(1,2),(3,4)])
def test_inc(x,y):
    assert inc(x) == y
```

Markers: skip and xfail

- *@pytest.mark.skip(reason="...") — always skip*
- *@pytest.mark.skipif(condition, reason="...") — conditional skip*
- *@pytest.mark.xfail(reason="...") — expected failure (won't fail CI)*
- *Register custom markers in pytest.ini to avoid warnings*

- **Example:**
- *@pytest.mark.skipif(sys.platform == "win32", reason="Unix-only")*

Monkeypatch & Mocking

- monkeypatch fixture for changing environment, attributes, or os.environ
- Use unittest.mock or pytest-mock for mocking callables
- Prefer monkeypatch for environment-level changes
- For temporary changes

Examples:

- *monkeypatch.setenv("API_KEY", "test")*
- *monkeypatch.setattr(module, "func", lambda: 42)*

tmp_path, tmpdir & Filesystem

- Use tmp_path fixture (pathlib.Path) to create temporary files
- tmp_path_factory for module/session scoped temp dirs
- Prefer tmp_path over manual tempfile in tests

Example:

```
def test_write(tmp_path):
    p = tmp_path/"data.txt“
    p.write_text("hello")
    assert p.read_text() == "hello"
```

Plugins: Ecosystem

- **pytest-cov:** coverage reporting (combine with coverage)
- **pytest-xdist:** parallel test execution (-n auto)
- **pytest-mock:** wrapper over unittest.mock
- **pytest-asyncio:** testing async code
- **hypothesis:** property-based testing

Parallel Testing with xdist

- Install: pip install pytest-xdist
- Run: pytest -n auto or pytest -n 4
- Use with pytest-cov carefully:

  pytest -n auto --dist=loadscope --cov=my_package --cov-report=term-missing

- Watch out for shared resources and flaky tests (use locks)

Coverage with pytest-cov

- Install: pip install pytest-cov
- Command:

  pytest --cov=my_package --cov-report=term-missing

- Combine with CI to fail on low coverage: --cov-fail-under=80
- HTML report: --cov-report=html

Test Selection & Troubleshooting

- -k name expressions, -m markers
- -q -k "name" for quick iteration
- --lf / --last-failed to re-run failing tests in previous executions
- --maxfail and -x to stop early

Organizing Tests & conftest.py

- Use tests/ directory root
- conftest.py contains fixtures and hooks shared across tests
- Avoid importing fixtures from conftest.py (pytest auto-discovers)
- Keep conftest.py focused to avoid tight coupling

```
project/
    src/
        myapp/
    tests/
        unit/
        integration/
        conftest.py
```

Hooks & Plugin Points

- Functions that let you customize how pytest collects, runs, and reports tests.
- pytest_addoption(parser) to add CLI flags
- pytest_configure(config) and pytest_sessionstart(session) lifecycle hooks
- pytest_runtest_setup(item) to alter test setup
- Useful for project-wide instrumentation and advanced behavior

Example snippet:

```
def pytest_addoption(parser):
    parser.addoption('--runslow', action='store_true', help='run slow tests')
```

Debugging Tests

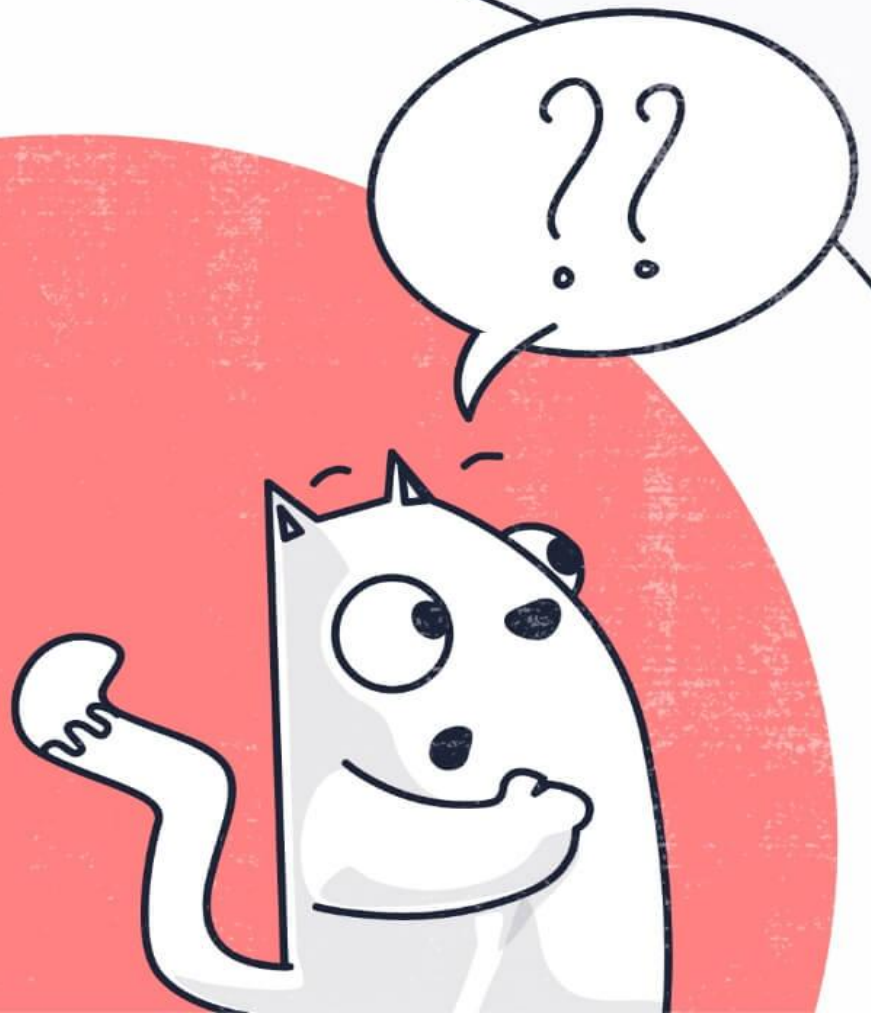
- pytest -k testname -s to see prints
- pytest --pdb or pytest -s --pdb to drop into pdb on failures
- Use pytest.set_trace() or import pdb; pdb.set_trace() inside tests
- caplog and capsys capture logging/stdout for assertions

Property-Based Testing with Hypothesis

- Install: pip install hypothesis
- Replace example-based tests with @given strategies
- Helps find edge cases you didn't think of
- pytest + smart input generation + automatic edge-case discovery

1. Generate many random test inputs
   - Using strategies, Hypothesis creates valid inputs (ints, strings, dicts, lists, custom objects).
2. Trigger test failures
   - It tries hundreds of combinations — not just the ones you write.
   - If a failure occurs, Hypothesis moves to step 3.
3. Shrink failures
   - Hypothesis tries to find the smallest possible failing example, so your error messages are easy to read.

CI Integration



How to say

This is out of scope

nicely

What is Flakiness in Pytest?

A flaky test is a test that:

- fails sometimes
- passes sometimes
- with no code changes
- Numerous reasons(async, network, time, shared state etc)

Organizing Complex Fixtures

Factory fixtures: return factory functions for dynamic creation

Use module or session scope for expensive setup (DB, containers)

Teardown order: fixtures with wider scope are torn down last

```
@pytest.fixture
def user_factory(db_session):
    def _create(**kwargs):
            u = User(**kwargs)
            db_session.add(u)
            db_session.commit()
            return u
    return _create
```

Test Doubles & Integration Tests

Test Doubles are stand-ins for real dependencies when writing tests.

- unittest.mock
- pytest-mock (mocker fixture)
- monkeypatch fixture (environment patching)

Integration tests check how multiple components work together.

Unit Test	Integration Test
Uses test doubles	Uses real components
Focus: one function	Focus: behaviour across boundaries
Fast	Slower
No I/O	Often requires I/O
Small, isolated	High-value scenarios

Best Practices

- Keep tests small, focused, and independent
- Name tests descriptively (what behavior they validate)
- Use fixtures to reduce duplication, not hide intent
- Run tests locally before pushing and in CI
- Pin test dependencies to minimize surprises

Troubleshooting Checklist

- Are tests isolated? Check global state.
- Is order dependence causing flakiness?
- Use --maxfail=1 -x to find first issue.
- Re-run failing tests with -k to speed debugging.
- Use --lf to focus on last failures.

Additional Resources

- Official docs: https://docs.pytest.org/
- pytest plugin index: https://plugins.pytest.org/
- Hypothesis docs: https://hypothesis.readthedocs.io/
- PyPI for pytest-cov, pytest-xdist, pytest-asyncio
- Example repos: point to company or community repos (replace)

Demo Commands

- python -m pytest
- python -m coverage run -m pytest
- coverage html
- python -m pytest -q -m "not slow“
- python -m pytest -q --runslow -m slow
- python -m pytest -q -n auto
- python –m pytest -q --cov=src --cov-report=html

Closing / Q&A

- Recap: Fixtures + Parametrization + Plugins = productive testing

Questions??

- Contact:

 - vairamuthu.thangavel@gmail.com

 - www.linkedin.com/in/vaira-muthu-thangavel

 - https://github.com/armeggaddon