

## **PyTest Concepts with Real-Life Use Cases**

### **01. Python Basics**

Used to validate core business logic such as string formatting, calculations, and conditions before automation layers.

Example: validating user input, OTP length, invoice totals, username formatting.

### **02. Pytest Assertions**

Used to verify application behavior like API responses, database values, and UI validations.

Example: asserting login success, checking API status codes, validating calculations.

### **03. Skip, Markers & xfail**

Used to control test execution in CI/CD pipelines.

Example: skipping unstable tests, marking smoke/sanity tests, handling known bugs without failing builds.

### **04. Parameterization & Fixtures**

Used for running the same test with multiple datasets and managing setup/teardown.

Example: testing login with multiple users, API tests with different payloads, DB setup and cleanup.

### **05. Pytest Customizations**

Used for enterprise automation.

Example: switching QA/PROD environments, reading configs, data-driven testing via CSV, reusable utilities.

### **06. Pytest-BDD**

Used for business-readable tests.

Example: banking transactions, e-commerce checkout flows, insurance claims validation using Gherkin scenarios.

## **Conclusion**

This structure mirrors real-world automation frameworks used in professional QA and SDET roles.

# Deep Dive: PyTest Markers, Fixtures, Parametrization & Customization

## 1. PyTest Markers

Markers are labels attached to test functions to control selection and execution.

Real-life use: separating smoke, sanity, regression, slow, API, UI tests.

Example:

```
@pytest.mark.sanity  
def test_login():  
    pass
```

Usage in CI:

```
pytest -m sanity
```

## 2. xfail & xpass

xfail = expected failure (known bug)

xpass = unexpected pass (bug fixed but test marked xfail)

Real-life:

- Use xfail when defect exists
- Use strict=True to catch XPASS

Example:

```
@pytest.mark.xfail(reason='Known bug', strict=True)  
def test_payment():  
    assert process_payment()
```

## 3. Parametrization

Parametrization runs the same test with multiple datasets.

Real-life:

- Login with multiple users
- API tests with multiple payloads
- DB validations

Example:

```
@pytest.mark.parametrize("user,pwd", [  
    ("admin", "admin123"),  
    ("guest", "guest123")  
])  
  
def test_login(user,pwd):  
    assert login(user,pwd)
```

## 4. Fixtures

Fixtures handle setup & teardown.

Scopes:

function, class, module, session

yield is used for teardown after test execution.

Real-life:

- DB connection setup
- Browser launch
- API auth token creation

## 5. conftest.py

Used to share fixtures across multiple test files.

Real-life:

- Centralized environment setup
- Shared DB connections
- Authentication fixtures

## 6. 04\_Pytest\_Parameterise\_and\_Setup\_or\_Teardown\_Tests Folder

Purpose:

- Data-driven tests
- Resource lifecycle management

Files typically include:

- Parametrized tests
- Setup/teardown fixtures

## 7. PyTest Customizations (05\_Pytest\_Customizations)

Enterprise-grade pytest features:

- Custom CLI options
- Environment-based execution
- Config parsing
- CSV data providers

## 8. pytest\_addoption & --cmdopt

Allows passing runtime parameters.

Example:

```
pytest --cmdopt=Prod
```

Used for:

- QA/PROD switching
- CI/CD pipelines

## 9. config Folder (qa.prop, prod.prop)

Stores environment-specific values.

Real-life:

- URLs
- Credentials
- Feature toggles

## 10. utils Folder

Reusable helper logic.

Examples:

- ConfigFileParser
- CSV readers
- Utility functions

## 11. test\_argtest.py

Validates command-line options and config reading.

Ensures correct environment config is loaded.

## 12. test\_data\_provider.py

Reads data from CSV and feeds into tests.

Used for data-driven automation.

## 13. test\_getconfigdata.py

Validates config values like URLs, users.

Ensures environment configuration correctness.

## Conclusion

This structure mirrors real corporate automation frameworks.

It demonstrates strong understanding of pytest internals,

CI/CD readiness, and enterprise testing patterns.

# Complete Guide: PyTest + Jenkins (Topic-wise & Real-World)

## 1. Python Basics in Testing

Used to validate core business logic before automation layers.

Real-life: input validation, OTP length, invoice totals, username rules.

## 2. PyTest Assertions

Used to verify behavior of applications.

Includes equality, membership, truthy/falsy, and exception testing.

Real-life: API responses, DB validations, UI messages.

## 3. Markers, Skip, xfail, xpass

Markers group tests (smoke, sanity, regression).

skip/skipif exclude tests temporarily.

xfail handles known bugs.

xpass detects when bugs are fixed.

Used heavily in CI/CD pipelines.

## 4. Parametrization

Runs same test with multiple datasets.

Real-life: login with many users, API payloads, DB rows.

Reduces duplication and increases coverage.

## 5. Fixtures & conftest.py

Fixtures manage setup/teardown.

Scopes: function, class, module, session.

conftest.py shares fixtures across tests.

Real-life: DB connections, auth tokens, browser setup.

## 6. PyTest Customizations

Enterprise features like pytest\_addoption.

- Allows QA/PROD switching.
- Uses config files and CSV data.
- Scales automation frameworks.

## 7. Jenkins Overview

Jenkins is an automation server for CI/CD.

Runs tests automatically on code changes.

## 8. Jenkins + PyTest Integration

Jenkins executes pytest commands.

Uses markers and CLI options.

Example: `pytest -m sanity --cmdopt=QA`

## 9. Jenkins Pipelines

Pipeline stages: checkout, install, test, report.

Defined using `Jenkinsfile`.

## 10. Jenkins + PyTest Concepts Mapping

Markers control which tests Jenkins runs.

`xfail` prevents known bugs from failing builds.

Fixtures handle setup during CI runs.

Parametrization increases coverage.

## 11. Reports & Feedback

Jenkins publishes console logs, JUnit XML, HTML reports.

Used by QA, Dev, and Managers.

## Conclusion

This merged structure reflects real enterprise automation.

PyTest handles testing logic.

Jenkins automates execution and delivery.

Together they form a complete CI/CD testing ecosystem.

# Jenkins – Deep Dive for PyTest & CI/CD

## 1. What is Jenkins?

Jenkins is an open-source automation server used to automate build, test, and deployment pipelines.

It continuously integrates code changes, runs automated tests, and reports results automatically.

## 2. Why Jenkins is Used (Real Life)

- Automatically run tests on every code change
- Detect bugs early
- Enforce quality gates
- Reduce manual testing effort
- Enable CI/CD pipelines

## 3. Jenkins + PyTest (How They Work Together)

Jenkins triggers PyTest on:

- Code commit
- Pull request
- Scheduled runs
- Manual triggers

Example command Jenkins runs:

```
pytest -v -m sanity --cmdopt=QA
```

## 4. Jenkins Pipeline Flow

1. Developer pushes code to GitHub
2. Jenkins job is triggered
3. Code is checked out
4. Dependencies installed
5. PyTest executed
6. Reports generated
7. Build marked PASS/FAIL

## 5. Jenkinsfile (Pipeline as Code)

A Jenkinsfile defines the CI pipeline using Groovy syntax.

Stages:

- Checkout
- Build
- Test
- Report

## 6. Sample Jenkinsfile for PyTest

```
pipeline {  
    agent any  
    stages {  
        stage('Checkout') {  
            steps { git 'https://github.com/user/repo.git' }  
        }  
        stage('Install') {  
            steps { sh 'pip install -r requirements.txt' }  
        }  
        stage('Test') {  
            steps { sh 'pytest -v -m sanity --cmdopt=QA' }  
        }  
    }  
}
```

## 7. Jenkins + Markers

Markers control which tests Jenkins executes.

Examples:

- Smoke tests on every commit
- Regression tests nightly
- Sanity tests before deployment

## 8. Jenkins + xfail / xpass

`xfail` prevents known bugs from failing the pipeline.

`xpass` alerts when bugs are fixed.

Used to keep CI stable while tracking defects.

## 9. Jenkins + Fixtures & Parametrization

Fixtures handle setup/teardown during CI runs.

Parametrization increases coverage without extra code.

## 10. Jenkins + Environment Switching

Jenkins passes environment variables:

- QA
- UAT
- PROD

PyTest reads these using `pytest_addoption`.

## 11. Jenkins Reports

Jenkins can publish:

- Console logs
- JUnit XML reports
- HTML reports
- Coverage reports

## 12. Real-Life Jenkins Use Cases

- Banking – payment validation
- E-commerce – checkout flow
- Healthcare – data validation
- Microservices – API automation

## Conclusion

Jenkins + PyTest form the backbone of modern automation.

Together they enable fast feedback, reliable releases, and scalable CI/CD pipelines.