pytest in Python

1. What Is pytest?

`pytest` is a popular testing framework for Python.

It helps you write, organize, and run automated tests with less boilerplate than Python's built-in `unittest` framework.

With pytest, a simple test can look like this:

```
def test_add():
    assert add(2, 3) == 5
```

There is no required test class and no need to use methods such as:

```
self.assertEqual(...)
```

pytest uses normal Python `assert` statements.

---

2. Why Use pytest?

Imagine you have:

```
def calculate_total(price, quantity):
    return price * quantity
```

You could manually test it:

```
print(calculate_total(10, 3))
print(calculate_total(5, 2))
```

But as your application grows, manually checking everything becomes difficult.

pytest lets you create automated checks:

```
def test_calculate_total():
    assert calculate_total(10, 3) == 30
```

Then run:

```
pytest
```

pytest executes the tests and reports the results.

---

3. pytest vs unittest

Both frameworks solve the same fundamental problem.

Feature	`unittest`	`pytest`
Included with Python	Yes	No
Installation	Not required	`pip install pytest`
Basic test style	Class based	Function based
Assertion style	`self.assertEqual()`	`assert`
Fixtures	`setUp()` / `tearDown()`	Fixtures
Parameterization	`subTest()` and other approaches	`@pytest.mark.parametrize`
Plugins	Available	Large ecosystem
Boilerplate	More	Less
Learning style	Structured	Lightweight

The choice depends on the project.

The important thing is understanding the underlying testing principles rather than memorizing one framework.

---

4. Installing pytest

Because pytest is a third-party package, install it with pip:

```
pip install pytest
```

If you are using a virtual environment, activate it first.

Then verify:

```
pytest --version
```

You should see the installed pytest version.

---

5. Your First pytest Test

Suppose:

```
def add(a, b):
    return a + b
```

Create:

```
test_calculator.py
```

```
from calculator import add


def test_add():
    assert add(2, 3) == 5
```

Run:

```
pytest
```

pytest discovers the test and executes it.

---

6. Why pytest Tests Are Short

With `unittest`, you might write:

```
import unittest


class TestCalculator(unittest.TestCase):


```

```
    def test_add(self):
        self.assertEqual(add(2, 3), 5)
```

With pytest:

```
def test_add():
    assert add(2, 3) == 5
```

pytest uses Python's existing `assert` syntax and provides detailed failure information when an assertion fails.

---

7. pytest Test Discovery

pytest automatically searches for tests according to naming conventions.

Common test files:

```
test_calculator.py
test_users.py
calculator_test.py
```

Common test functions:

```
def test_add():
    ...


def test_login():
    ...
```

The `test_` prefix is important for automatic discovery.

---

8. Test Classes in pytest

pytest does not require classes, but you can use them when they improve organization.

Example:

```
class TestCalculator:

    def test_add(self):
        assert add(2, 3) == 5

    def test_subtract(self):
        assert subtract(5, 2) == 3
```

Unlike `unittest`, you do not need:

```
class TestCalculator(unittest.TestCase):
```

You also do not need:

```
self.assertEqual(...)
```

---

9. The `assert` Statement

The most important syntax in basic pytest testing is:

```
assert actual == expected
```

Example:

```
assert 2 + 3 == 5
```

Another example:

```
result = add(10, 5)

assert result == 15
```

If the condition is true, the test passes.

If it is false, pytest reports the failure.

---

10. Testing Boolean Values

Suppose:

```
def is_even(number):
    return number % 2 == 0
```

Tests:

```
def test_even_number():
    assert is_even(4) is True


def test_odd_number():
    assert is_even(5) is False
```

You can also write:

```
assert is_even(4)
assert not is_even(5)
```

Using `is True` and `is False` can make the expected Boolean behavior especially explicit.

---

11. Testing `None`

Example:

```
def find_resource(resources, resource_id):
    for resource in resources:
        if resource["id"] == resource_id:
            return resource

    return None
```

Test:

```
def test_missing_resource():
    resources = [
        {"id": 1, "title": "Python"}
    ]

    result = find_resource(resources, 99)

    assert result is None
```

---

12. Testing Collections

Suppose:

```
def get_languages():
    return ["Python", "JavaScript", "Dart"]
```

Tests:

```
def test_languages():
    languages = get_languages()

    assert "Python" in languages
    assert len(languages) == 3
```

You can also compare the complete list:

```
assert languages == [
    "Python",
    "JavaScript",
    "Dart"
]
```

Use exact collection comparison when ordering and complete contents matter.

---

13. Testing Dictionaries

Example:

```
def get_user():
    return {
        "name": "Hafsa",
        "role": "developer"
    }
```

Test:

```
def test_get_user():
    user = get_user()

    assert user["name"] == "Hafsa"
    assert user["role"] == "developer"
```

You can also test:

```
assert "name" in user
```

---

14. Testing Exceptions

Suppose:

```
def divide(a, b):
    if b == 0:
        raise ValueError("Cannot divide by zero")

    return a / b
```

pytest provides:

```
import pytest
```

Then:

```
def test_divide_by_zero():
    with pytest.raises(ValueError):
        divide(10, 0)
```

The test passes only if `ValueError` is raised.

---

15. Checking the Exception Message

You can inspect the exception:

```
def test_divide_by_zero():
    with pytest.raises(ValueError) as error:
        divide(10, 0)

    assert str(error.value) == "Cannot divide by zero"
```

Use this when the exception message is part of the behavior you want to guarantee.

---

16. Testing Multiple Inputs

Suppose:

```
def is_even(number):
    return number % 2 == 0
```

You could write:

```
def test_is_even():
    assert is_even(2) is True
    assert is_even(4) is True
    assert is_even(5) is False
    assert is_even(7) is False
```

But pytest provides a cleaner way to run the same test with different inputs.

---

17. Parameterized Tests

Use:

```
@pytest.mark.parametrize()
```

Example:

```
import pytest


@pytest.mark.parametrize(
    "number, expected",
    [
        (2, True),
        (4, True),
```

```
        (5, False),
        (7, False),
    ]
)
def test_is_even(number, expected):
    assert is_even(number) is expected
```

pytest runs the test once for each input pair.

Conceptually:

```
2 → True
4 → True
5 → False
7 → False
```

This reduces repetitive test code.

---

18. Why Parameterization Is Useful

Without parameterization:

```
def test_one():
    assert is_even(2) is True


def test_two():
    assert is_even(4) is True


def test_three():
    assert is_even(5) is False
```

With parameterization:

```
@pytest.mark.parametrize(
    "number, expected",
    [
        (2, True),
        (4, True),
        (5, False),
    ]
)
def test_is_even(number, expected):
    assert is_even(number) is expected
```

The test logic exists once.

Only the test data changes.

---

19. Testing Boundary Values

Suppose:

```
def is_adult(age):
    return age >= 18
```

Important cases include:

```
@pytest.mark.parametrize(
    "age, expected",
    [
        (17, False),
        (18, True),
        (19, True),
    ]
)
def test_is_adult(age, expected):
    assert is_adult(age) is expected
```

The boundary value `18` deserves special attention because behavior changes there.

---

20. pytest Fixtures

One of pytest's most important features is the fixture system.

A fixture provides data or setup that tests can use.

Example:

```
import pytest


@pytest.fixture
def resources():
    return [
        {"id": 1, "title": "Python"},
        {"id": 2, "title": "Testing"},
    ]
```

A test can request the fixture simply by using its name:

```
def test_resource_count(resources):
    assert len(resources) == 2
```

pytest automatically provides the fixture value.

---

21. Why Fixtures Matter

Imagine ten tests need the same test data.

Without a fixture, you might repeat:

```
resources = [
    {"id": 1, "title": "Python"},
    {"id": 2, "title": "Testing"},
]
```

in every test.

With a fixture:

```
@pytest.fixture
def resources():
    return [
        {"id": 1, "title": "Python"},
        {"id": 2, "title": "Testing"},
    ]
```

Then:

```
def test_count(resources):
    assert len(resources) == 2


def test_first_resource(resources):
    assert resources[0]["title"] == "Python"
```

The setup is centralized.

---

22. Fixture Scope

Fixtures can have different lifetimes.

Common scopes include:

```
function
class
module
```

```
package
session
```

The default is:

```
function
```

That means a fresh fixture is normally created for each test.

Example:

```
@pytest.fixture
def resources():
    return []
```

Each test receives its own fixture instance.

This helps maintain test isolation.

---

23. Fixture Example With a Resource Manager

Suppose:

```
class ResourceManager:

    def __init__(self):
        self.resources = []

    def add(self, resource):
        self.resources.append(resource)

    def find(self, resource_id):
        for resource in self.resources:
            if resource["id"] == resource_id:
                return resource

        return None
```

Fixture:

```
@pytest.fixture
def manager():
    manager = ResourceManager()

    manager.add({
        "id": 1,
        "title": "Python"
```

```
    })

    manager.add({
        "id": 2,
        "title": "Testing"
    })

    return manager
```

Tests:

```
def test_find_resource(manager):
    result = manager.find(1)

    assert result["title"] == "Python"


def test_missing_resource(manager):
    assert manager.find(99) is None
```

---

24. `conftest.py`

pytest allows shared fixtures to live in:

```
conftest.py
```

Example:

```
project/
│
├── app/
│   └── resources.py
│
├── tests/
│   ├── conftest.py
│   ├── test_resources.py
│   └── test_users.py
│
└── README.md
```

Inside `conftest.py`:

```
import pytest
```

```
@pytest.fixture
def resources():
    return [
        {"id": 1, "title": "Python"},
        {"id": 2, "title": "Testing"},
    ]
```

Tests inside the appropriate test directory can use the fixture without importing it manually.

---

25. Fixture Setup and Cleanup

Fixtures can also perform cleanup.

Example:

```
@pytest.fixture
def temporary_resource():
    resource = create_resource()

    yield resource

    delete_resource(resource)
```

The sequence is:

```
Create resource
      ↓
Run test
      ↓
Delete resource
```

The code after `yield` is cleanup code.

This is useful for resources such as:

- temporary files
- database records
- connections
- directories
- external resources

---

26. Testing Files

Suppose:

```
def save_message(path, message):
    with open(path, "w") as file:
        file.write(message)
```

A test may use pytest's temporary path fixture:

```
def test_save_message(tmp_path):
    file_path = tmp_path / "message.txt"

    save_message(file_path, "Hello")

    assert file_path.read_text() == "Hello"
```

`tmp_path` gives the test a temporary directory.

This avoids polluting the actual project directory.

---

27. Testing With Temporary Data

Temporary resources are useful because tests should avoid unnecessary changes to real project data.

Instead of:

```
project/data/users.json
```

a test can work with:

```
temporary test directory
```

This keeps tests isolated and repeatable.

---

28. Testing APIs and External Services

Suppose:

```
def get_user_from_api(user_id):
    ...
```

A unit test should generally avoid making a real network request every time.

Instead, you can isolate the external dependency using mocking.

pytest can work with Python's built-in `unittest.mock`:

```
from unittest.mock import Mock
```

Example:

```
def test_get_user():
    api = Mock()

    api.get.return_value = {
        "id": 1,
        "name": "Hafsa"
    }

    result = api.get(1)

    assert result["name"] == "Hafsa"
```

Mocking is a larger topic, but the principle is:

```
Real external dependency
        ↓
Replace with controlled test double
        ↓
Test your own logic
```

---

29. `pytest` Markers

Markers let you categorize tests.

For example:

```
@pytest.mark.slow
def test_large_import():
    ...
```

You can then run specific categories.

First, you may configure the marker in your pytest configuration.

Then:

```
pytest -m slow
```

can run tests marked as `slow`.

Markers become especially useful in larger projects.

---

30. Common Marker Uses

You might organize tests using categories such as:

```
unit
integration
slow
database
api
```

Example:

```
@pytest.mark.integration
def test_database_connection():
    ...
```

Then:

```
pytest -m integration
```

runs the integration tests.

Markers are labels, not replacements for good test organization.

---

31. Skipping Tests

Sometimes a test should temporarily be skipped.

pytest provides:

```
@pytest.mark.skip(reason="Feature not implemented yet")
def test_future_feature():
    ...
```

You can also conditionally skip:

```
@pytest.mark.skipif(
    sys.platform == "win32",
    reason="Not supported on Windows"
)
def test_linux_feature():
    ...
```

Skipping should be intentional.

A skipped test should not become a permanent way of hiding a problem.

---

32. Expected Failures

Sometimes a test describes known behavior that is currently failing.

pytest provides:

```
@pytest.mark.xfail
def test_known_issue():
    assert some_function() == expected
```

This communicates that the failure is currently expected.

Use this carefully.

A test marked `xfail` should represent a known, intentional condition rather than simply hiding an unexpected failure.

---

33. Running pytest

Run all discovered tests:

```
pytest
```

Verbose mode:

```
pytest -v
```

Run a specific file:

```
pytest tests/test_calculator.py
```

Run a specific test:

```
pytest tests/test_calculator.py::test_add
```

Run tests matching a keyword:

```
pytest -k calculator
```

Run a marker:

```
pytest -m integration
```

---

34. Understanding pytest Output

A successful run might show:

```
5 passed
```

A failed run might show:

```
1 failed, 4 passed
```

pytest normally provides useful information about:

```
test file
test function
line number
expected value
actual value
assertion failure
```

Example:

```
assert calculate_total(10, 2) == 30
```

If the function returns `20`, pytest can show the difference between the expected and actual values.

---

35. Reading a Failed Test

When a test fails, follow this process:

```
Read test name
     ↓
Read assertion
     ↓
Check actual result
     ↓
Check expected result
     ↓
Inspect application code
     ↓
Identify root cause
     ↓
Fix code or test expectation
     ↓
Run tests again
```

Do not automatically change the expected value just to make the test pass.

---

36. Test Behavior, Not Implementation

Suppose:

```
def add(a, b):
    return a + b
```

Good:

```
def test_add():
    assert add(2, 3) == 5
```

The test should continue to work if the implementation becomes:

```
def add(a, b):
    result = a + b
    return result
```

The behavior has not changed.

Tests that focus too heavily on internal implementation can make refactoring unnecessarily difficult.

---

37. Testing Edge Cases

Suppose:

```
def calculate_average(numbers):
    return sum(numbers) / len(numbers)
```

Important cases include:

```
normal list
single value
negative values
zero values
empty list
```

The empty list needs an explicit decision.

For example, you could define:

```
def calculate_average(numbers):
    if not numbers:
        raise ValueError("Cannot calculate average of empty list")

    return sum(numbers) / len(numbers)
```

Then test:

```
def test_empty_average():
    with pytest.raises(ValueError):
        calculate_average([])
```

---

38. Regression Testing

Suppose a bug was discovered:

```
Resource filtering returns resources from the wrong category.
```

After fixing the code, create a test that protects the behavior:

```
def test_filter_resources_uses_category():
    resources = [
        {"title": "Python", "category": "programming"},
        {"title": "CSS", "category": "web"},
    ]

    result = filter_resources(resources, "programming")

    assert result == [
        {"title": "Python", "category": "programming"}
    ]
```

Now the same bug is less likely to return unnoticed.

---

39. Testing a haas.dev Resource Service

Imagine:

```
def filter_resources(resources, category):
    return [
        resource
        for resource in resources
        if resource["category"] == category
    ]
```

Create a fixture:

```
import pytest


@pytest.fixture
def resources():
    return [
        {
            "title": "Python Basics",
            "category": "Python"
        },
        {
```

```
            "title": "React Basics",
            "category": "React"
        },
        {
            "title": "Python Testing",
            "category": "Python"
        },
    ]
```

Then:

```
def test_filter_python_resources(resources):
    result = filter_resources(resources, "Python")

    assert len(result) == 2
    assert result[0]["title"] == "Python Basics"
    assert result[1]["title"] == "Python Testing"
```

Another test:

```
def test_filter_missing_category(resources):
    result = filter_resources(resources, "Flutter")

    assert result == []
```

And:

```
def test_filter_empty_resources():
    assert filter_resources([], "Python") == []
```

---

40. Organizing a pytest Project

A practical structure:

```
project/
│
├── app/
│   ├── __init__.py
│   ├── calculator.py
│   └── resources.py
│
├── tests/
│   ├── conftest.py
│   ├── test_calculator.py
```

```
│   └── test_resources.py
│
├── pyproject.toml
├── requirements.txt
└── README.md
```

The application code and tests have clear boundaries.

---

41. pytest and `pyproject.toml`

pytest can be configured through project configuration files, including:

```
pyproject.toml
```

For example, a project can configure test discovery or markers there.

A simplified configuration might look like:

```
[tool.pytest.ini_options]
testpaths = ["tests"]
```

Now pytest knows that tests are located in:

```
tests/
```

This becomes useful as a project grows.

---

42. Test Naming

Good:

```
def test_empty_cart_has_zero_total():
    ...
```

Good:

```
def test_missing_resource_returns_none():
    ...
```

Less useful:

```
def test_1():
    ...
```

A good test name communicates the behavior being protected.

---

43. Fixtures vs Repeated Setup

Without a fixture:

```
def test_python_resource():
    resources = [
        {"title": "Python", "category": "programming"}
    ]

    assert ...


def test_resource_count():
    resources = [
        {"title": "Python", "category": "programming"}
    ]

    assert ...
```

With a fixture:

```
@pytest.fixture
def resources():
    return [
        {"title": "Python", "category": "programming"}
    ]
```

Then:

```
def test_python_resource(resources):
    assert ...


def test_resource_count(resources):
    assert ...
```

The fixture removes repeated setup.

---

44. Avoid Overusing Fixtures

Fixtures are powerful, but too many layers of fixtures can make tests difficult to understand.

Bad:

```
test
 ↓
fixture A
```

```
 ↓
fixture B
 ↓
fixture C
 ↓
fixture D
```

A developer may have difficulty figuring out where the test data came from.

Prefer simple, clearly named fixtures.

---

45. Test Isolation

Each test should ideally be able to run independently.

Good:

```
def test_create_resource():
    manager = ResourceManager()

    manager.add(resource)

    assert ...
```

Another test:

```
def test_remove_resource():
    manager = ResourceManager()

    manager.add(resource)
    manager.remove(resource["id"])

    assert ...
```

Each test creates its own state.

---

46. Unit Tests vs Integration Tests

pytest can run different types of tests.

Unit test

Tests one small piece:

```
filter_resources()
```

Integration test

Tests multiple components together:

```
Resource Service
      ↓
Database
```

End-to-end test

Tests a complete workflow:

```
Open application
     ↓
Search resources
     ↓
Open resource
     ↓
Download PDF
```

pytest itself does not define your application's testing strategy.

It provides tools that can support these different testing levels.

---

47. Test Coverage

pytest can be combined with coverage tools.

A common package is:

```
pip install pytest-cov
```

Then:

```
pytest --cov
```

This can report which parts of the code were executed by tests.

Remember:

```
High coverage ≠ automatically high-quality tests
```

Coverage tells you about exercised code.

It does not tell you whether your assertions meaningfully verify behavior.

---

48. Common pytest Mistakes

Mistake 1: Forgetting the `test_` prefix

Incorrect:

```
def calculate_test():
    ...
```

Better:

```
def test_calculate():
    ...
```

Mistake 2: Testing only successful inputs

Also consider:

```
empty values
invalid values
boundaries
exceptions
```

Mistake 3: Reusing mutable state between tests

Shared mutable state can cause one test to affect another.

Prefer fresh fixtures.

Mistake 4: Using real external services unnecessarily

Tests become slower and less deterministic.

Use controlled test doubles where appropriate.

Mistake 5: Making fixtures too complicated

A test should remain understandable.

Mistake 6: Ignoring failing tests

A red test is information.

Investigate it instead of hiding it.

Mistake 7: Testing implementation details

Tests should primarily protect externally meaningful behavior.

Mistake 8: Treating coverage as the goal

The goal is confidence in important behavior, not simply a high percentage.

---

49. Best Practices

Keep tests focused

One test should generally verify one clear behavior.

Use descriptive names

A failure should tell you what behavior broke.

Use fixtures for reusable setup

But keep fixtures simple.

Test boundaries

Boundary values frequently reveal bugs.

Test invalid inputs

Applications need predictable behavior when inputs are incorrect.

Keep unit tests fast

Fast tests encourage frequent execution.

Keep tests independent

Avoid hidden dependencies between tests.

Use parameterization

Avoid unnecessary repetition when only the input changes.

Keep external dependencies controlled

Use mocks, fakes, or test environments where appropriate.

Run the entire suite regularly

A feature may break code outside the file you changed.

---

50. A Complete pytest Example

Application:

```
def calculate_total(price, quantity):
    if price < 0:
        raise ValueError("Price cannot be negative")

    if quantity < 0:
        raise ValueError("Quantity cannot be negative")

    return price * quantity
```

Tests:

```
import pytest

from shop import calculate_total


def test_calculate_total():
    assert calculate_total(10, 3) == 30


def test_zero_quantity():
    assert calculate_total(10, 0) == 0


def test_negative_price():
    with pytest.raises(ValueError):
        calculate_total(-10, 2)


def test_negative_quantity():
    with pytest.raises(ValueError):
        calculate_total(10, -2)
```

Run:

```
pytest
```

This suite checks:

```
normal behavior
zero quantity
invalid price
invalid quantity
```

---

51. Mini Project: Resource Testing

Create:

```
class ResourceManager:

    def __init__(self):
        self.resources = []

    def add(self, resource):
```

```
        self.resources.append(resource)

    def remove(self, resource_id):
        self.resources = [
            resource
            for resource in self.resources
            if resource["id"] != resource_id
        ]

    def find(self, resource_id):
        for resource in self.resources:
            if resource["id"] == resource_id:
                return resource

        return None
```

Create a pytest fixture:

```
import pytest


@pytest.fixture
def manager():
    manager = ResourceManager()

    manager.add({
        "id": 1,
        "title": "Python"
    })

    manager.add({
        "id": 2,
        "title": "Testing"
    })

    return manager
```

Write tests for:

```
adding resources
finding resources
missing resources
removing resources
```

```
empty manager
multiple resources
```

Example:

```
def test_find_resource(manager):
    result = manager.find(1)

    assert result["title"] == "Python"
```

---

52. 5 Minute Challenge

Create:

```
def calculate_average(numbers):
    ...
```

Define the behavior for:

```
[10, 20, 30] → 20
[5] → 5
[0, 0, 0] → 0
[] → ?
```

Choose what should happen for an empty list.

Then write pytest tests for every case.

Also add a parameterized test:

```
@pytest.mark.parametrize(
    "numbers, expected",
    [
        ([10, 20, 30], 20),
        ([5], 5),
        ([0, 0, 0], 0),
    ]
)
def test_calculate_average(numbers, expected):
    assert calculate_average(numbers) == expected
```

---

53. Exercises

Exercise 1: Positive Numbers

Create:

```
def is_positive(number):
    ...
```

Test:

```
10
1
0
-1
-10
```

Use parameterization.

Exercise 2: Password Validation

Create:

```
def validate_password(password):
    ...
```

Require at least 8 characters.

Test:

```
valid password
7 characters
8 characters
empty password
```

Use `pytest.raises()` if invalid passwords should raise an exception.

Exercise 3: Shipping Cost

Create:

```
def calculate_shipping(total):
    ...
```

Use:

```
total < 50 → shipping fee
total >= 50 → free shipping
```

Test:

```
49
50
```

```
51
```

Exercise 4: Resource Filtering

Create:

```
def filter_resources(resources, category):
    ...
```

Test:

```
matching category
missing category
empty list
multiple matching resources
```

Use a fixture for the resource data.

Exercise 5: Temporary Files

Create a function that writes text to a file.

Use pytest's `tmp_path` fixture to test it without creating permanent files in your project.

---

54. Mini Quiz

1. Is pytest included in Python's standard library?

A. Yes

B. No

C. Only on Windows

D. Only on Linux

Answer: B

---

2. How do you install pytest?

A. `python install pytest`

B. `pip install pytest`

C. `install pytest`

D. `python get pytest`

Answer: B

---

3. What is the common pytest test function naming convention?

A. `check_`

B. `run_`

C. `test_`

D. `verify_`

Answer: C

---

4. What syntax does pytest commonly use for assertions?

A. `self.assertEqual()`

B. `assert`

C. `check()`

D. `verify()`

Answer: B

---

5. What is a pytest fixture?

A. A Python compiler

B. Reusable test setup or test data

C. A database

D. A package manager

Answer: B

---

6. Which fixture provides a temporary directory?

A. `temp_dir`

B. `temporary`

C. `tmp_path`

D. `test_path`

Answer: C

---

7. Which decorator is used for parameterized tests?

A. `@pytest.parameters`

B. `@pytest.parametrize`

C. `@pytest.inputs`

D. `@pytest.data`

Answer: B

---

55. Interview Questions

Beginner

1. What is pytest?
2. Why is pytest popular?
3. How do you install pytest?
4. How do you run pytest?
5. What is pytest test discovery?
6. How do pytest tests differ from `unittest` tests?
7. What is a pytest fixture?
8. What does `pytest.raises()` do?
9. What is `conftest.py`?
10. What does `pytest -v` do?

Intermediate

11. What is parameterization in pytest?
12. Why are fixtures useful?
13. What are fixture scopes?
14. What is `tmp_path`?
15. What are pytest markers?
16. What is the difference between skipping and expected failure?
17. How can pytest test exceptions?
18. How do you run only one test?
19. How do you run tests matching a keyword?
20. How can pytest be integrated with coverage tools?

Practical

21. How would you test a function that depends on a database?
22. How would you test an API client without making real API calls?
23. How would you share test data across multiple test files?
24. How would you test multiple input combinations without duplicating code?
25. How would you investigate a failing pytest test?

---

56. pytest Cheat Sheet

Install

```
pip install pytest
```

Check version

```
pytest --version
```

Run all tests

```
pytest
```

Verbose output

```
pytest -v
```

Run one file

```
pytest tests/test_file.py
```

Run one test

```
pytest tests/test_file.py::test_name
```

Search tests by keyword

```
pytest -k calculator
```

Basic assertion

```
assert result == expected
```

Boolean

```
assert result is True
assert result is False
```

None

```
assert result is None
```

Membership

```
assert item in items
```

Exception

```
with pytest.raises(ValueError):
    function()
```

Parameterization

```
@pytest.mark.parametrize(
    "input_value, expected",
    [
        (1, True),
        (2, False),
    ]
)
def test_something(input_value, expected):
    assert function(input_value) is expected
```

Fixture

```
@pytest.fixture
def data():
    return [...]
```

Use fixture

```
def test_something(data):
    assert ...
```

Temporary directory

```
def test_file(tmp_path):
    ...
```

Marker

```
@pytest.mark.integration
def test_database():
    ...
```

Run marker

```
pytest -m integration
```

Skip

```
@pytest.mark.skip(reason="Not ready")
def test_feature():
    ...
```

Expected failure

```
@pytest.mark.xfail
def test_known_issue():
    ...
```

Coverage

```
pytest --cov
```

---

57. pytest Mental Model

When writing a pytest test, think:

```
What behavior should exist?
        ↓
What inputs matter?
        ↓
What should happen?
        ↓
What could go wrong?
        ↓
What setup does the test need?
        ↓
Write a focused test
        ↓
Run pytest
        ↓
Read failures
        ↓
Fix the root cause
        ↓
Run the suite again
```

The goal is not to write as many tests as possible.

The goal is to create reliable, understandable checks for important behavior.

---

58. Key Takeaways

- `pytest` is a popular third-party Python testing framework.
- Install it with `pip install pytest`.
- Tests are usually simple functions beginning with `test_`.
- pytest uses Python's normal `assert` statement.

- `pytest.raises()` checks expected exceptions.
- Fixtures provide reusable test data and setup.
- `conftest.py` is commonly used for shared fixtures.
- `@pytest.mark.parametrize` allows the same test logic to run with multiple inputs.
- `tmp_path` provides isolated temporary directories for file tests.
- Markers can organize and selectively run tests.
- pytest can work with mocks and other test doubles.
- Tests should be independent and deterministic.
- Test normal behavior, edge cases, boundaries, and invalid inputs.
- Test behavior rather than unnecessary implementation details.
- Coverage is useful for identifying untested code, but coverage percentage alone does not measure test quality.
- pytest can support unit, integration, and larger automated test suites.
- Good tests make debugging, refactoring, and maintenance safer.

The essential pattern remains:

```
Arrange
   ↓
Act
   ↓
Assert
```

And with pytest, that often becomes as simple as:

```
def test_expected_behavior():
    result = function(input)

    assert result == expected
```

Visit haas.dev for more resources and guides.

https://dev-roast-app.vercel.app

https://dev-roast-app.vercel.app/resources