
2-Pytest_Fixutre - PowerPoint
Tell me what you want to do
Share
17
pytest
fixtures
YASH
Technologies
More than what you think.
Notes
Comments
67%
Slide 17 of 35
English (India)
Search
ENG
IN
16:11
05-11-2025

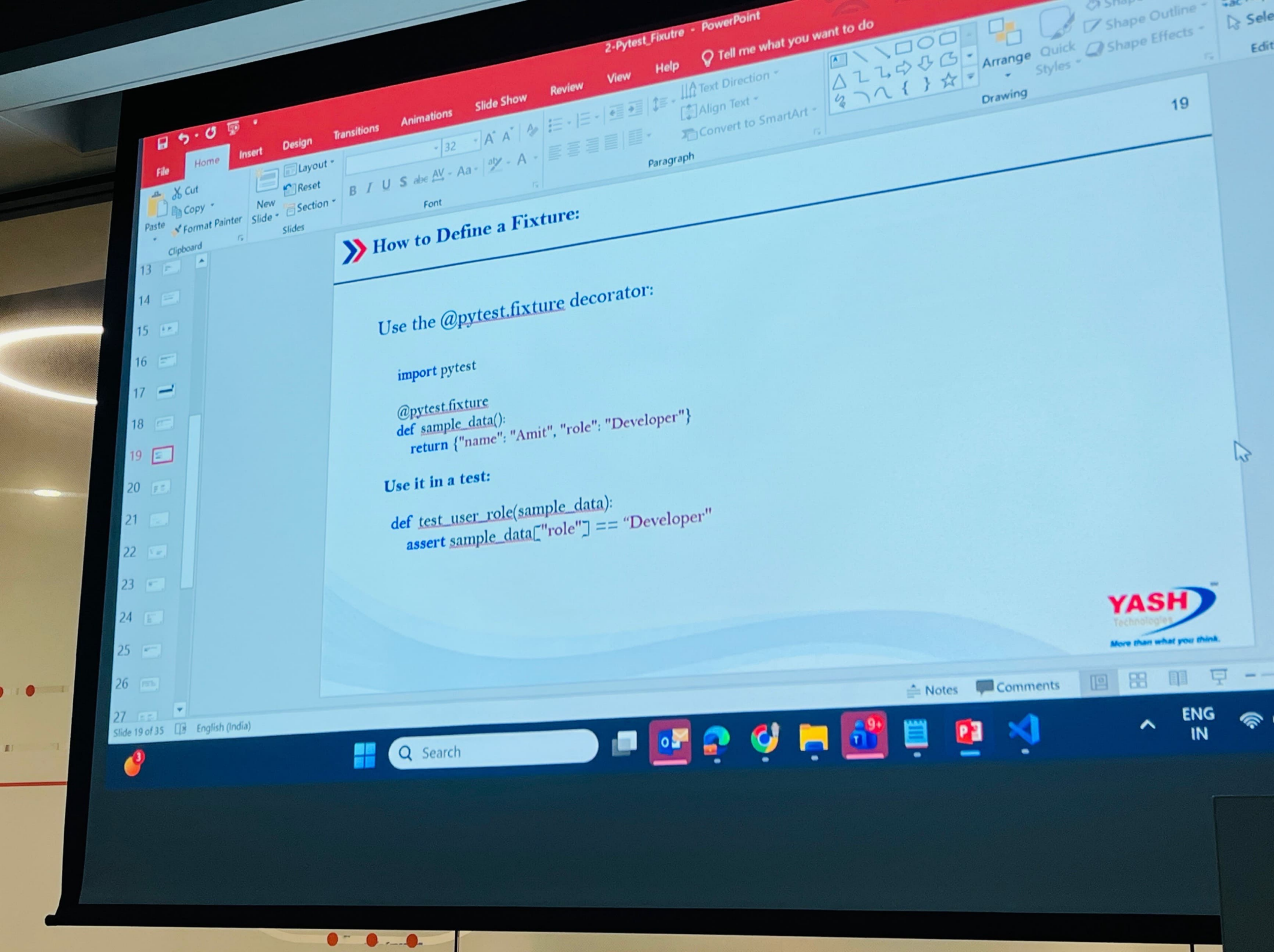
What Are Fixtures in Pytest?

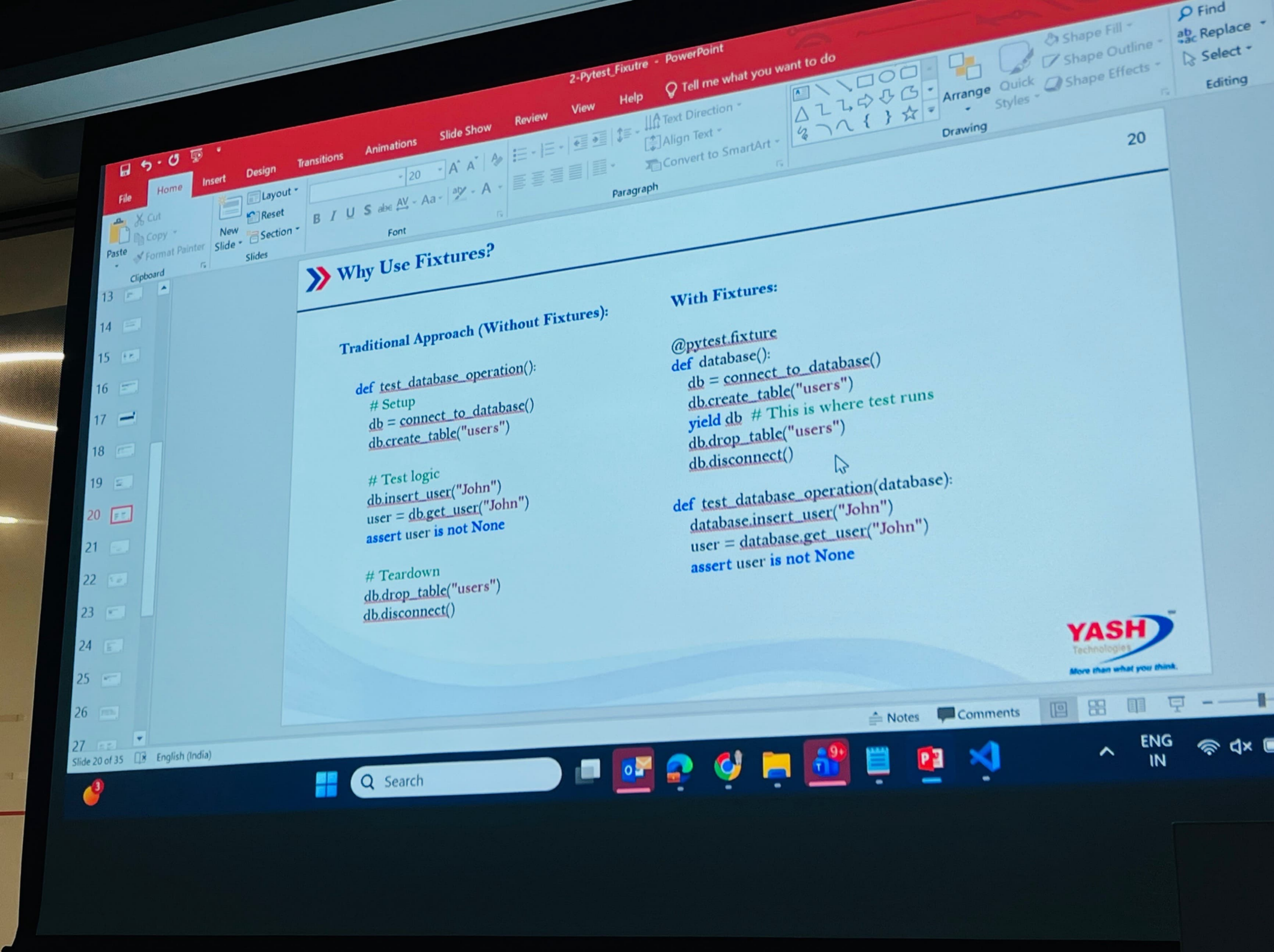
Fixtures are functions that run before and after test functions to set up preconditions and clean up afterwards. They are used to **prepare test environments**, such as:

- Run before test execution (setup)
- Run after test execution (teardown)
- Can be reused across multiple tests
- Manage test dependencies

They help in **isolating test logic** from setup/cleanup code, making tests **cleaner, reusable, and maintainable**.

YASH Technologies
More than what you think.


2-Pytest_Fixutre - PowerPoint
Tell me what you want to do
File
Home
Insert
Design
Transitions
Animations
Slide Show
Review
View
Help
Clipboard
Slides
Font
Paragraph
Drawing
19
How to Define a Fixture:
Use the @pytest.fixture decorator:
import pytest
@pytest.fixture
def sample_data():
return {"name": "Amit", "role": "Developer"}
Use it in a test:
def test_user_role(sample_data):
assert sample_data["role"] == "Developer"
YASH
Technologies
More than what you think.
Notes
Comments
Slide 19 of 35
English (India)
Search
ENG
IN


2-Pytest_Fixutre - PowerPoint
Tell me what you want to do
20
Why Use Fixtures?
Traditional Approach (Without Fixtures):
def test_database_operation():
# Setup
db = connect_to_database()
db.create_table("users")
# Test logic
db.insert_user("John")
user = db.get_user("John")
assert user is not None
# Teardown
db.drop_table("users")
db.disconnect()
With Fixtures:
@pytest.fixture
def database():
db = connect_to_database()
db.create_table("users")
yield db # This is where test runs
db.drop_table("users")
db.disconnect()
def test_database_operation(database):
database.insert_user("John")
user = database.get_user("John")
assert user is not None
YASH
Technologies
More than what you think.
Notes
Comments
Slide 20 of 35
English (India)
Search
ENG
IN

Benefits:

- Clean separation of setup/teardown from test logic
- Reusable across multiple tests
- Automatic cleanup even if test fails
- Modular and maintainable

Where to define fixtures?

Usually, you put fixtures in a special file called **conftest.py**, so they can be shared across multiple test files.

Example project:

```
tests/
├── conftest.py
├── test_login.py
└── test_dashboard.py
```

YASH Technologies
More than what you think.

Fixture Example

In the "conftest.py" file insert the following:

```
import pytest
@pytest.fixture
def input_total( ):
    total = 100
    return total
```

In the "testrough1.py" file insert

```
import pytest
def test_total_divisible_by_5(input_total):
    assert input_total % 5 == 0

def test_total_divisible_by_10(input_total):
    assert input_total % 10 == 0

def test_total_divisible_by_20(input_total):
    assert input_total % 20 == 0

def test_total_divisible_by_9(input_total):
    assert input_total % 9 == 0
```

YASH Technologies
More than what you think.

Parametrized Fixtures

You can **parametrize fixtures** to run tests with different data:

```
@pytest.fixture(params=[1, 2, 3])
def number(request):
    return request.param

def test_even(number):
    assert number % 2 == 0
```

This will run the test 3 times **with** values 1, 2, **and** 3.

YASH Technologies
More than what you think.

Advanced Example with Multiple Parameters:

```
@pytest.mark.parametrize("operation, num1, num2, expected", [
    ("add", 5, 3, 8),
    ("subtract", 10, 4, 6),
    ("multiply", 3, 4, 12),
    ("divide", 15, 3, 5),
])

def test_calculator_operations(operation, num1, num2, expected):
    if operation == "add":
        assert num1 + num2 == expected
    elif operation == "subtract":
        assert num1 - num2 == expected
    elif operation == "multiply":
        assert num1 * num2 == expected
    elif operation == "divide":
        assert num1 / num2 == expected
```

Autouse Fixtures

You can make a fixture run **automatically** for all tests without mentioning its name.

```
@pytest.fixture(autouse=True)
def log_test_start():
    print("\nStarting test...")
```

This runs before every test automatically.

Fixture Scope

Fixtures have different scopes that control how often they are executed:

Scope	Description	When to Use
function (default)	Runs once per test function	Individual test isolation
class	Runs once per test class	Shared setup for class tests
module	Runs once per test module	Expensive setup (DB connection)
session	Runs once per entire test session	Very expensive setup (test server)

Scope Examples

```python
import pytest
# Function scope (default) - runs for every test
@pytest.fixture(scope="function")
def function_fixture():
    print("\n=== Setting up function fixture ===")
    yield "function_data"
    print("=== Tearing down function fixture ===")

# Class scope - runs once per class
@pytest.fixture(scope="class")
def class_fixture():
    print("\n=== Setting up class fixture ===")
    yield "class_data"
    print("=== Tearing down class fixture ===")

# Module scope - runs once per module
@pytest.fixture(scope="module")
def module_fixture():
    print("\n=== Setting up module fixture ===")
    yield "module_data"
    print("=== Tearing down module fixture ===")

# Session scope - runs once per test session
@pytest.fixture(scope="session")
def session_fixture():
    print("\n=== Setting up session fixture ===")
    yield "session_data"
    print("=== Tearing down session fixture ===")
```

YASH Technologies
More than what you think.


2-Pytest_Fixutre - PowerPoint
Tell me what you want to do
File
Home
Insert
Design
Transitions
Animations
Slide Show
Review
View
Help
Paste
Cut
Copy
Format Painter
Clipboard
New Slide
Layout
Reset
Section
Slides
Font
Paragraph
Text Direction
Align Text
Convert to SmartArt
Arrange
Quick Styles
Shape Fill
Shape Outline
Shape Effects
Drawing
Find
Replace
Select
Editing
28
Fixtures Use Case:
YASH
Technologies
More than what you think.
Notes
Comments
Slide 28 of 35
English (India)
Search
ENG
IN

Fixtures Use Case:

Step 1: Create conftest.py

All fixtures are typically defined in a file named conftest.py at the root of your test directory.

```python
# conftest.py
import pytest

@pytest.fixture
def sample_data():
    """Provides sample test data"""
    return [1, 2, 3, 4, 5]
```

```python
@pytest.fixture
def calculator():
    """Provides a calculator instance"""
    class Calculator:
        def add(self, a, b):
            return a + b

        def multiply(self, a, b):
            return a * b

    return Calculator()
```

YASH Technologies
More than what you think.

Step 2: Use Fixtures in Tests

```python
# test_example.py
def test_sum_data(sample_data):
    """Test receives sample_data fixture automatically"""
    assert sum(sample_data) == 15

def test_calculator_add(calculator):
    """Test receives calculator fixture"""
    result = calculator.add(5, 3)
    assert result == 8

def test_calculator_multiply(calculator):
    """Another test using the same fixture"""
    result = calculator.multiply(4, 3)
    assert result == 12
```

How It Works:

- Pytest sees sample_data parameter in test function
- Looks for a fixture named sample_data
- Executes the fixture function
- Passes the return value to the test

YASH Technologies
More than what you think.

Factory as Fixtures

The "factory as fixture" pattern is useful when you need multiple instances or customized data in a single test.

Problem: Regular fixtures return a single instance. What if you need multiple users, products, or objects?

Solution: Return a factory function instead of data.

```
@pytest.fixture
def user_factory():
    def make_user(name, age):
        return {"name": name, "age": age}
    return make_user

def test_users(user_factory):
    user1 = user_factory("Alice", 25)
    user2 = user_factory("Bob", 30)
    assert user1["age"] == 25
```

YASH Technologies
More than what you think.

Factory as Fixtures

The "factory as fixture" pattern is useful when you need multiple instances or customized data in a single test.

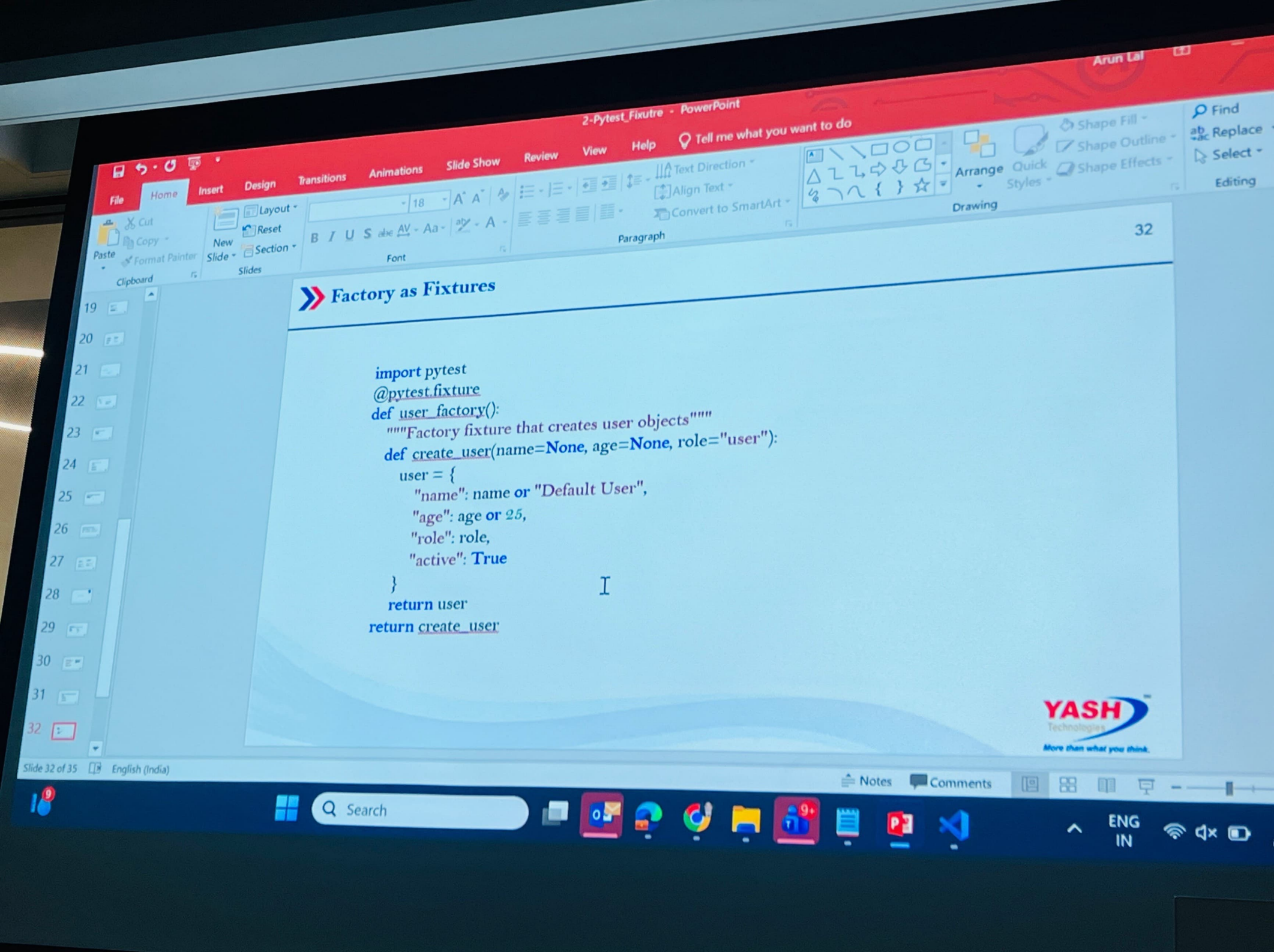
Problem: Regular fixtures return a single instance. What if you need multiple users, products, or objects?

Solution: Return a factory function instead of data.

```
@pytest.fixture
def user_factory():
    def make_user(name, age):
        return {"name": name, "age": age}
    return make_user

def test_users(user_factory):
    user1 = user_factory("Alice", 25)
    user2 = user_factory("Bob", 30)
    assert user1["age"] == 25
```

YASH Technologies
More than what you think.


2-Pytest_Fixutre - PowerPoint
Arun Lal
File Home Insert Design Transitions Animations Slide Show Review View Help Tell me what you want to do
Cut Copy Format Painter Paste Clipboard
New Slide Layout Reset Section Slides
Font
Paragraph
Text Direction Align Text Convert to SmartArt
Arrange Quick Styles Shape Fill Shape Outline Shape Effects Drawing
Find Replace Select Editing
32
Factory as Fixtures
import pytest
@pytest.fixture
def user_factory():
"""Factory fixture that creates user objects"""
def create_user(name=None, age=None, role="user"):
user = {
"name": name or "Default User",
"age": age or 25,
"role": role,
"active": True
}
return user
return create_user
YASH Technologies
More than what you think.
Slide 32 of 35
English (India)
Notes
Comments
Search
ENG IN

Factory as Fixtures


```python
def test_multiple_users(user_factory):
    """Test creating multiple users with different parameters"""
    admin_user = user_factory(name="Admin", age=35, role="admin")
    regular_user = user_factory(name="Regular User")
    guest_user = user_factory(role="guest", age=20)

    assert admin_user["role"] == "admin"
    assert regular_user["name"] == "Regular User"
    assert guest_user["age"] == 20
```