Unittest

▼ Unit Testing

Unit test là một loại kiểm thử phần mềm nhằm xác định tính chính xác của các đơn vị nhỏ nhất trong code như các hàm, phương thức... đến các đơn vị lớn hơn như thư viện, các lớp...

Mục tiêu là đảm bảo mỗi phần của mã nguồn đều hoạt động như mong đợi

Test Driver: là một file test có cùng cấp độ truy cập với code và cùng ngôn ngữ với unit cần kiểm tra, nó sẽ chạy với đầu vào cho trước (qua terminal hay file lưu dữ liệu đầu vào) và kiểm tra đầu ra xem đã đúng mong đợi chưa

▼ Testing Driven Development

Là phương pháp phát triển phần mềm trong đó việc viết các unit test được đặt lên hàng đầu, trước cả việc viết mã nguồn chính

Quy trình của TDD thường bắt đầu bằng việc viết 1 unit test đơn giản để mô tả tính năng/ hành vi mà bạn muốn code sẽ thực hiện. Sau đó, bắt đầu viết code để “pass” unit test đó mà không gặp lỗi

Mỗi khi muốn có 1 tính năng mới, ta lại tạo unit test và viết code để thoả mãn unit test đó. Nâng dần các tiêu chí của unit test để code hoàn thiện nhất có thể. Lặp đi lặp lại quy trình đó

Ưu điểm: tạo ra mã nguồn có tính ổn định, ít lỗi và dễ bảo trì hơn cũng như cung cấp một bộ kiểm tra tự động cho phần mềm

▼ Pytest

▼ Fixture

Là cách để chuẩn bị dữ liệu, trạng thái (states) và dependencies cho các bài test. Fixture có thể được sử dụng để cung cấp các giá trị cần thiết cho nhiều test mà không phải tạo lại cho mỗi lần

Khi sử dụng fixture, pytest sẽ tự động chạy fixture trước để tạo dữ liệu và sau khi test sẽ xoá đi (ví dụ tạo database, môi trường...)

```
# Arrange
@pytest.fixture
def fruit_bowl():
    return [Fruit("apple"), Fruit("banana")]


def test_fruit_salad(fruit_bowl):
    # Act
    fruit_salad = FruitSalad(*fruit_bowl)

    # Assert
    assert all(fruit.cubed for fruit in fruit_salad.fruit)
```

Một số tính năng:

- ▼ Fixture gọi đến fixture khác

```
@pytest.fixture
def first_entry():
    return "a"

@pytest.fixture
def order(first_entry):
    return [first_entry]

def test_string(order):
    order.append("b")

    assert order == ["a", "b"]
```

- ▼ Một fixture gọi đến nhiều fixture khác

```
@pytest.fixture
def first_entry():
    return "a"

@pytest.fixture
def second_entry():
    return 2

@pytest.fixture
def order(first_entry, second_entry):
    return [first_entry, second_entry]
```

▼ Sử dụng lại fixture trong nhiều test khác nhau

```
@pytest.fixture
def first_entry():
    return "a"

@pytest.fixture
def order(first_entry):
    return [first_entry]


def test_string(order):
    order.append("b")

    assert order == ["a", "b"]


def test_int(order):
    order.append(2)
```

```
    assert order == ["a", 2]
```

▼ Autouse fixture

```
@pytest.fixture(autouse=True)
def append_first(order, first_entry):
    return order.append(first_entry)


def test_string_only(order, first_entry):
    assert order == [first_entry]


def test_string_and_int(order, first_entry):
    order.append(2)
    assert order == [first_entry, 2]
```

Trong ví dụ này, fixture tên là “append_first” là autouse fixture. Bởi vì nó tự động nên cả hai bài test đều bị ảnh hưởng bởi nó, mặc dù không có bài kiểm tra nào yêu cầu nó. Tuy nhiên, điều đó không có nghĩa là hai bài test không thể yêu cầu fixture này, chỉ là không cần thiết.

▼ Fixture scope

Fixture được tạo khi được yêu cầu bởi test và bị huỷ bỏ tuỳ vào scope:

- function: default scope, fixture bị huỷ ở cuối test
- class: fixture bị huỷ trong khi teardown test cuối cùng trong class
- module: fixture bị huỷ trong khi teardown test cuối trong module
- package: fixture bị huỷ trong khi teardown test cuối trong gói (gói là nơi định nghĩa fixture, gồm các gói con và các thư mục con với nó)
- session: fixture bị huỷ ở cuối phiên...

▼ Teardown/ Cleanup fixture

yield fixture: fixture sử dụng yield thay vì return để teardown sau khi chạy test. Ban đầu, pytest sẽ chạy các fixture cho đến trước yield rồi mới chuyển qua fixture tiếp. Đến khi hoàn thành, pytest chạy những đoạn sau fixture để teardown theo thứ tự ngược lại

```
@pytest.fixture
def mail_admin():
    return MailAdminClient()


@pytest.fixture
def sending_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    mail_admin.delete_user(user)


@pytest.fixture
def receiving_user(mail_admin):
    user = mail_admin.create_user()
    yield user
    user.clear_mailbox()
    mail_admin.delete_user(user)


def test_email_received(sending_user, receiving_user):
    email = Email(subject="Hey!", body="How's it going?"
    sending_user.send_email(email, receiving_user)
    assert email in receiving_user.inbox
```

Nếu yield fixture đưa ra exception trước khi yield, pytest sẽ không teardown theo đoạn mã phía sau yield mà sẽ teardown như thường lệ

Có thể sử dụng finalizer method thay vì yield nhưng phải cẩn thận vì pytest sẽ chạy finalizer một khi đã thêm vào code, kể cả khi nó sinh ra

exception. Vì thế, chỉ nên thêm finalizer một khi fixture đã làm xong gì đó cần teardown

```
 @pytest.fixture
def receiving_user(mail_admin, request):
    user = mail_admin.create_user()

    def delete_user():
        mail_admin.delete_user(user)

    request.addfinalizer(delete_user)
    return user


 @pytest.fixture
def email(sending_user, receiving_user, request):
    _email = Email(subject="Hey!", body="How's it going?
    sending_user.send_email(_email, receiving_user)

    def empty_mailbox():
        receiving_user.clear_mailbox()

    request.addfinalizer(empty_mailbox)
    return _email
```

▼ Sử dụng nhiều assert một cách an toàn

Nếu cần kiểm tra một function với nhiều yêu cầu và cần nhiều assert để kiểm tra, ta nên có cấu trúc hợp lý để không gặp exception

```
from src.utils.pages import LoginPage, LandingPage
from src.utils import AdminApiClient
from src.utils.data_types import User

class TestLandingPageSuccess:
```

```
    @pytest.fixture(scope="class", autouse=True)
    def login(self, driver, base_url, user):
        driver.get(urljoin(base_url, "/login"))
        page = LoginPage(driver)
        page.login(user)

    def test_name_in_header(self, landing_page, user):
        assert landing_page.header == f"Welcome, {user.n

    def test_sign_out_button(self, landing_page):
        assert landing_page.sign_out_button.is_displayed

    def test_profile_link(self, landing_page, user):
        profile_href = urljoin(base_url, f"/profile?id={
        assert landing_page.profile_link.get_attribute("
```

▼ Sử dụng markers để pass data cho fixture

```
@pytest.fixture
def fixt(request):
    marker = request.node.get_closest_marker("fixt_data"
    if marker is None:
        # Handle missing marker in some way...
        data = None
    else:
        data = marker.args[0]

    # Do something with the data
    return data


@pytest.mark.fixt_data(42)
def test_fixt(fixt):
    assert fixt == 42
```

▼ Factories as Fixture

“Factories as Fixture” là pattern hỗ trợ trong những tình huống kết quả của một fixture cần nhiều lần trong một test. Thay vì trả về dữ liệu trực tiếp, fixture sẽ gửi về một hàm có thể sinh ra dữ liệu → hàm này có thể được gọi nhiều lần trong một test

```
@pytest.fixture
def make_customer_record():
    created_records = []

    def _make_customer_record(name):
        record = models.Customer(name=name, orders=[])
        created_records.append(record)
        return record

    yield _make_customer_record

    for record in created_records:
        record.destroy()


def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

▼ Truyền tham số vào fixture

Fixture có thể được truyền vào các tham số trong trường hợp chúng được gọi nhiều lần

```
@pytest.fixture(scope="module", params=["smtp.gmail.com"
def smtp_connection(request):
    smtp_connection = smtplib.SMTP(request.param, 587, t
    yield smtp_connection
    print(f"finalizing {smtp_connection}")
    smtp_connection.close()
```

Trong ví dụ trên, fixture được truyền vào hai tham số khác nhau nên những test sử dụng fixture này sẽ phải chạy hai lần với hai tham số khác nhau mỗi lần chạy

```
$ pytest -q test_module.py
FFFF
================================ FAILURES ============
_____________________________ test_ehlo[smtp.gmail.com] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0004

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
        assert b"smtp.gmail.com" in msg
>       assert 0  # for demo purposes
E       assert 0

test_module.py:7: AssertionError
_____________________________ test_noop[smtp.gmail.com] _____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0004

    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
        assert response == 250
>       assert 0  # for demo purposes
E       assert 0

test_module.py:13: AssertionError
_____________________________ test_ehlo[mail.python.org] ____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0005

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
```

```
        assert response == 250
>       assert b"smtp.gmail.com" in msg
E       AssertionError: assert b'smtp.gmail.com' in b'ma

test_module.py:6: AssertionError
-------------------------- Captured stdout setup -------
finalizing <smtplib.SMTP object at 0xdeadbeef0004>
_____________________________ test_noop[mail.python.org] ____

smtp_connection = <smtplib.SMTP object at 0xdeadbeef0005

    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
        assert response == 250
>       assert 0  # for demo purposes
E       assert 0

test_module.py:13: AssertionError
------------------------- Captured stdout teardown -----
finalizing <smtplib.SMTP object at 0xdeadbeef0005>
========================== short test summary info ======
FAILED test_module.py::test_ehlo[smtp.gmail.com] - asser
FAILED test_module.py::test_noop[smtp.gmail.com] - asser
FAILED test_module.py::test_ehlo[mail.python.org] - Asse
FAILED test_module.py::test_noop[mail.python.org] - asse
4 failed in 0.12s
```

▼ Sử dụng mark với tham số

```
@pytest.fixture(params=[0, 1, pytest.param(2, marks=pyte
def data_set(request):
    return request.param


def test_data(data_set):
    pass
```

```
####### Ket qua #######
$ pytest test_fixture_marks.py -v
============================= test session starts ========
platform linux -- Python 3.x.y, pytest-8.x.y, pluggy-1.x
cachedir: .pytest_cache
rootdir: /home/sweet/project
collecting ... collected 3 items

test_fixture_marks.py::test_data[0] PASSED
test_fixture_marks.py::test_data[1] PASSED
test_fixture_marks.py::test_data[2] SKIPPED (uncondition

======================== 2 passed, 1 skipped in 0.12s ===
```

▼ Override fixture

Một fixture có thể bị ghi đè ở cấp độ thư mục nhất định. Nhất là những base fixture và super fixture rất dễ bị ghi đè.

```
tests/
    conftest.py
        # content of tests/conftest.py
        import pytest

        @pytest.fixture
        def username():
            return 'username'

    test_something.py
        # content of tests/test_something.py
        def test_username(username):
            assert username == 'username'

    subfolder/
        conftest.py
```

```
            # content of tests/subfolder/conftest.py
            import pytest

            @pytest.fixture
            def username(username):
                return 'overridden-' + username

        test_something_else.py
            # content of tests/subfolder/test_something_
            def test_username(username):
                assert username == 'overridden-username'
```

Hoặc cũng có thể ghi đè ở các test khác nhau trong cùng một thư mục

```
tests/
    conftest.py
        # content of tests/conftest.py
        import pytest

        @pytest.fixture
        def username():
            return 'username'

    test_something.py
        # content of tests/test_something.py
        import pytest

        @pytest.fixture
        def username(username):
            return 'overridden-' + username

        def test_username(username):
            assert username == 'overridden-username'

    test_something_else.py
        # content of tests/test_something_else.py
```

```
        import pytest

        @pytest.fixture
        def username(username):
            return 'overridden-else-' + username

        def test_username(username):
            assert username == 'overridden-else-username
```

▼ Monkeypatch/ mock

▼ Monkey patch: là một fixture trong pytest cho phép thay đổi thuộc tính và phương thức của đối tượng trong quá trình kiểm thử

Điều này hữu ích khi muốn thay thế một hàm hoặc một đối tượng với một phiên bản giả lập (mock) hoặc phiên bản đã thay đổi mà không cần sửa mã nguồn

```
# thay thế hàm
def test_monkeypatch_function(monkeypatch):
    def mock_return():
        return "mocked!"

    # Giả lập hàm `original_function` bằng `mock_return`
    monkeypatch.setattr("mymodule.original_function", mo

    from mymodule import original_function
    assert original_function() == "mocked!"

# thay thế thuộc tính
class MyClass:
    attribute = "original"

def test_monkeypatch_attribute(monkeypatch):
    monkeypatch.setattr(MyClass, "attribute", "mocked")
    assert MyClass.attribute == "mocked"
```

```
# thay thế biến môi trường
def test_monkeypatch_env(monkeypatch):
    monkeypatch.setenv("MY_VAR", "mocked_value")
    assert os.getenv("MY_VAR") == "mocked_value"
```

▼ Mock module: là một kỹ thuật trong kiểm thử phần mềm để thay thế các phần phụ thuộc của một hệ thống bằng các đối tượng giả lập mà mô phỏng hành vi của các đối tượng thật

```
from unittest.mock import patch

def external_function():
    return "original response"

def function_under_test():
    return external_function()

def test_mock_external_function():
    with patch('__main__.external_function', return_valu
        response = function_under_test()
        assert response == "mocked response"

# ví dụ 2
import requests
from unittest.mock import patch

def fetch_data(url):
    response = requests.get(url)
    return response.json()

@patch('requests.get') # decorator patch se thay request
                                            # trong tho
def test_fetch_data(mock_get):
        # mock_get la tham so dai dien cho mock object c
    # Đặt giá trị trả về cho mock object
    mock_get.return_value.json.return_value = {'key': 'v
```

```
    result = fetch_data('http://example.com/api')
    assert result == {'key': 'value'}

    # Kiểm tra xem phương thức có được gọi với URL đúng
    mock_get.assert_called_once_with('http://example.com
```

▼ Khi nào sử dụng monkeypatch và khi nào sử dụng mock

- Sử dụng monkeypatch khi bạn cần thay đổi các thuộc tính hoặc phương thức đơn giản trong pytest
- Sử dụng mocking khi bạn cần kiểm soát nhiều hơn về hành vi của đối tượng hoặc cần giả lập các đối tượng phức tạp

▼ Doctest

là module trong python cho phép kiểm thử mã nguồn python bằng cách kiểm tra trong chuỗi tài liệu (docstring) của nó

Doctest phù hợp cho những kiểm thử đơn giản, đảm bảo docstring luôn đúng hay giúp người dùng hiểu cách vận hành của đoạn mã

Cách dùng: thêm những yêu cầu đầu vào và đầu ra ngay sau khi khai báo hàm để kiểm tra doctest trong dấu “”” … “””

```
def add(a, b):
    """   # mock test bat dau tu day
    Trả về tổng của a và b.

    Ví dụ:
    >>> add(2, 3)
    5
    >>> add(-1, 1)
    0
    >>> add(0, 0)
```

```
    0
    """  # mock test ket thuc o day
    return a + b
```

Chạy doctest:

- Từ command line

```
python -m doctest -v your_module.py
```

- Từ mã nguồn

```
import doctest
doctest.testmod()
```

▼ Sử dụng thư mục và đường dẫn

Bạn có thể sử dụng tmp_path fixture - một phần của đối tượng pathlib.Path - để cung cấp thư mục tạm thời duy nhất cho bài test

```
CONTENT = "content"


def test_create_file(tmp_path):
    d = tmp_path / "sub"
    d.mkdir()
    p = d / "hello.txt"
    p.write_text(CONTENT, encoding="utf-8")
    assert p.read_text(encoding="utf-8") == CONTENT
    assert len(list(tmp_path.iterdir())) == 1
    assert 0
```

Sử dụng tmp_path_factory fixture:

```
import pytest

@pytest.fixture(scope="session")
def image_file(tmp_path_factory):
    img = compute_expensive_image()
    fn = tmp_path_factory.mktemp("data") / "img.png"
    img.save(fn)
    return fn


# contents of test_image.py
def test_histogram(image_file):
    img = load_image(image_file)
    # compute and test histogra
```

tmpdir và tmpdir_factory fixture:

tương tự như tmp nhưng sẽ trả về các đối tượng kế thừa py.path local chứ không trả về các đối tượng tiêu chuẩn pathlib.Path

▼ Hook function

Là các hàm đặc biệt cho phép tùy chỉnh hành vi của pytest tại nhiều điểm khác nhau trong quá trình chạy thử nghiệm. Các hook functions cung cấp cách để can thiệp vào quá trình chạy thử nghiệm, như trước khi thử nghiệm bắt đầu, sau khi thử nghiệm kết thúc, khi thu thập các thử nghiệm, và nhiều sự kiện khác.

Các hook functions được định nghĩa trong các plugin hoặc trong file `conftest.py` trong thư mục dự án. pytest tự động phát hiện và gọi các hook này tại những thời điểm thích hợp.

```
# Hàm này được gọi sau khi cấu hình pytest đã hoàn thành.
def pytest_configure(config):
    config.addinivalue_line(
        "markers", "custom_marker: mark a test as custom_ma
    )
```

```
# Hàm này cho phép sửa đổi danh sách các mục thử nghiệm sau
def pytest_collection_modifyitems(config, items):
    for item in items:
        if "custom_marker" in item.keywords:
            item.add_marker(pytest.mark.skip(reason="Skippi

# Hàm này được gọi ngay trước khi một thử nghiệm cụ thể đượ
def pytest_runtest_setup(item):
    if "skip_if_config" in item.keywords and not item.confi
        pytest.skip("Skipping this test due to --run-skippe
```

▼ Unittest

Là một module tích hợp sẵn trong thư viện chuẩn của python, được sử dụng để tạo và chạy các bài kiểm thử (unittest)

Gồm một số khái niệm quan trọng:

- test fixture: những chuẩn bị cần thiết để chạy một hay nhiều test cùng với hành động dọn dẹp như: tạo database tạm thời, chạy một process của server...
- test case: một đơn vị độc lập của testing, kiểm tra một phản hồi cụ thể cho một bộ đầu vào cụ thể. Có cung cấp một base class là TestCase
- test suite(bộ kiểm thử): là một tập hợp các test case, test suit hoặc cả hai. Sử dụng để tổng hợp các bài test nên được chạy cùng nhau
- test runner: là một thành phần điều phối việc thực hiện các thử nghiệm và cung cấp kết quả cho người dùng. Runner có thể sử dụng giao diện đồ họa, giao diện văn bản hoặc trả về một giá trị đặc biệt để cho biết kết quả thực hiện kiểm tra

▼ Ví dụ đơn giản về unittest:

```
import unittest

class TestStringMethods(unittest.TestCase):
```

```
    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())

    def test_split(self):
        s = 'hello world'
        self.assertEqual(s.split(), ['hello', 'world'])
        # check that s.split fails when the separator is n
        with self.assertRaises(TypeError):
            s.split(2)

if __name__ == '__main__':
    unittest.main()
```

▼ Chạy unittest

```
python -m unittest tests/test_something.py
```

▼ Test Discovery

Sử dụng lệnh `unittest` với tham số `discover` để tự động tìm và chạy tất cả các test case.

```
python -m unittest discover -s tests -p "test_*.py"

# tim cac test trong thu muc co ten 'tests'
# tim cac file co pattern test_
```

▼ Sử dụng lại các test code

```
def testSomething():
    something = makeSomething()
    assert something.name is not None
```

```
testcase = unittest.FunctionTestCase(testSomething,
                                     setUp=makeSomethingDB
                                     tearDown=deleteSometh
```

▼ Skipping test

Bỏ qua một số test mà không ảnh hưởng đến kết quả vì expect là nó sẽ fail

```
class MyTestCase(unittest.TestCase):

    @unittest.skip("demonstrating skipping")
    def test_nothing(self):
        self.fail("shouldn't happen")

    @unittest.skipIf(mylib.__version__ < (1, 3),
                     "not supported in this library versio
    def test_format(self):
        # Tests that work for only a certain version of th
        pass

    @unittest.skipUnless(sys.platform.startswith("win"), "
    def test_windows_support(self):
        # windows specific testing code
        pass

    def test_maybe_skipped(self):
        if not external_resource_available():
            self.skipTest("external resource not available
        # test code that depends on the external resource
        pass

# kqua----------------------------------------------
test_format (__main__.MyTestCase.test_format) ... skipped
test_nothing (__main__.MyTestCase.test_nothing) ... skippe
test_maybe_skipped (__main__.MyTestCase.test_maybe_skipped
```

```
test_windows_support (__main__.MyTestCase.test_windows_sup

----------------------------------------------------------
Ran 4 tests in 0.005s

OK (skipped=4)
```

Skip được cả các class

```
@unittest.skip("showing class skipping")
class MySkippedTestCase(unittest.TestCase):
    def test_not_run(self):
        pass
```

Sử dụng decorator expectedFailure để đánh dấu test case mà ta mong đợi sẽ thất bại

Hoặc có thể sử dụng decorator skip trong test case khi muốn skip test case nếu gặp điều kiện gì đó

```
def skipUnlessHasattr(obj, attr):
    if hasattr(obj, attr):
        return lambda func: func
    return unittest.skip("{!r} doesn't have {!r}".format(o
```

▼ Subtest

Sử dụng để thực hiện nhiều test nhỏ trong một test case khi các trường hợp kiểm thử tương tự nhau

```
import unittest

def divide(a, b):
    return a / b

class TestDivideFunction(unittest.TestCase):
```

```
    def test_divide(self):
        test_cases = [
            (10, 2, 5),
            (9, 3, 3),
            (5, 2, 2.5),
            (1, 1, 1),
            (10, 0, None)  # Đây là trường hợp đặc biệt có
        ]

        for a, b, expected in test_cases:
            with self.subTest(a=a, b=b, expected=expected)
                if b == 0:
                    with self.assertRaises(ZeroDivisionErr
                        divide(a, b)
                else:
                    self.assertEqual(divide(a, b), expecte

if __name__ == '__main__':
    unittest.main()
```

▼ Run

`run(result=None)`: Chạy kiểm thử, thu thập kết quả vào đối tượng `TestResult` được truyền vào. Nếu `result` không được cung cấp hoặc là `None`, một đối tượng kết quả tạm thời sẽ được tạo ra và sử dụng. Đối tượng kết quả sẽ được trả về cho caller của `run()`.

```
import unittest

# Một hàm đơn giản để kiểm thử
def divide(a, b):
    if b == 0:
        raise ZeroDivisionError("division by zero")
    return a / b
```

```
# Lớp kiểm thử
class TestDivideFunction(unittest.TestCase):

    def setUp(self):
        self.a = 10
        self.b = 5

    def test_divide_success(self):
        result = divide(self.a, self.b)
        self.assertEqual(result, 2)

    def test_divide_by_zero(self):
        with self.assertRaises(ZeroDivisionError):
            divide(self.a, 0)

if __name__ == '__main__':
    # Tạo một TestSuite
    suite = unittest.TestSuite()
    suite.addTest(TestDivideFunction('test_divide_success'
    suite.addTest(TestDivideFunction('test_divide_by_zero'

    # Tạo một đối tượng TestResult
    result = unittest.TestResult()

    # Chạy các kiểm thử và thu thập kết quả
    suite.run(result)

    # In kết quả
    print("Tests run:", result.testsRun)
    print("Errors:", result.errors)
    print("Failures:", result.failures)
    for test, reason in result.failures + result.errors:
        print(f"Test {test} failed due to {reason}")
```

▼ Lớp cơ bản TestCase

“setUp” để chuẩn bị bộ kiểm thử fixture và được gọi trước khi gọi mỗi phương thức kiểm thử trng class

“tearDown” được gọi ngay sau khi phương thức kiểm thử hoàn thành

```
import unittest

class TestExample(unittest.TestCase):

    def setUp(self):
        self.value = 42  # Thiết lập trước mỗi kiểm thử
        print("setUp được gọi")

    def test_add(self):
        self.assertEqual(self.value + 1, 43)

    def test_subtract(self):
        self.assertEqual(self.value - 1, 41)

if __name__ == '__main__':
    unittest.main()
```

“setUpClass” để chuẩn bị fixture cho cả class trước khi chạy, bao gồm tất cả các phương thức bên trong

“tearDownClass” tương tự

```
import unittest

class TestExample(unittest.TestCase):

    @classmethod
    def setUpClass(cls):
        cls.shared_resource = "Common resource"  # Thiết lậ
        print("setUpClass được gọi")

    def test_one(self):
```

```
        self.assertEqual(self.shared_resource, "Common res

    def test_two(self):
        self.assertEqual(self.shared_resource, "Common res

if __name__ == '__main__':
    unittest.main()
```

▼ Assert

Method	Checks that
assertEqual(a, b)	a == b
assertNotEqual(a, b)	a != b
assertTrue(x)	bool(x) is True
assertFalse(x)	bool(x) is False
assertIs(a, b)	a is b
assertIsNot(a, b)	a is not b
assertIsNone(x)	x is None
assertIsNotNone(x)	x is not None
assertIn(a, b)	a in b
assertNotIn(a, b)	a not in b
assertIsInstance(a, b)	isinstance(a, b)
assertNotIsInstance(a, b)	not isinstance(a, b)

Một số phương thức bổ sung:

Method	Checks that
assertRaises(exc, fun, *args, **kwds)	fun(*args, **kwds) raises exc
assertRaisesRegex(exc, r, fun, *args, **kwds)	fun(*args, **kwds) raises exc and the message matches regex r
assertWarns(warn, fun, *args, **kwds)	fun(*args, **kwds) raises warn
assertWarnsRegex(warn, r, fun, *args, **kwds)	fun(*args, **kwds) raises warn and the message matches regex r

assertLogs(logger, level)	The with block logs on logger with minimum level
assertNoLogs(logger, level)	The with block does not log on logger with minimum level

Phương thức `assertRaises` trong module `unittest` của Python được sử dụng để kiểm tra xem một ngoại lệ cụ thể có được ném ra trong một đoạn mã nào đó hay không. Điều này rất hữu ích khi bạn muốn đảm bảo rằng mã của bạn xử lý các tình huống lỗi một cách chính xác.

▼ IsolatedAsyncioTestCase

`unittest.IsolatedAsyncioTestCase` là một lớp con của `unittest.TestCase` trong Python. Nó cung cấp một API tương tự như `TestCase` và cho phép sử dụng các coroutine như các hàm kiểm thử.
Đặc điểm:

- **Tương thích với asyncio**
- Cô lập
- API tương tự 'TestCase'

```
import asyncio
import unittest

class MyAsyncTest(unittest.IsolatedAsyncioTestCase):

    async def test_coroutine(self):
        # Sử dụng coroutine như một hàm kiểm thử
        result = await self.async_method()
        self.assertEqual(result, 42)

    async def async_method(self):
        await asyncio.sleep(1)
        return 42

if __name__ == '__main__':
    unittest.main()
```

▼ Grouping test

Sử dụng TestSuit để nhóm các TestCase lại, giúp tổ chức các test thành nhóm có logic và hiệu quả hơn

```
import unittest

class TestClass1(unittest.TestCase):
    def test_method1(self):
        self.assertEqual(1 + 1, 2)

class TestClass2(unittest.TestCase):
    def test_method2(self):
        self.assertTrue(2 > 1)

# Tạo TestSuite
test_suite = unittest.TestSuite()

# Thêm các kiểm thử vào TestSuite
test_suite.addTest(TestClass1('test_method1'))
test_suite.addTest(TestClass2('test_method2'))

# Tạo trình chạy kiểm thử
test_runner = unittest.TextTestRunner()

# Chạy TestSuite
test_runner.run(test_suite)
```

▼ Loading and running test

Sử dụng unittest.TestLoader để lấy và chạy các tets ở file test khác

```
import unittest

# Tạo một lớp kiểm thử đơn giản
class MyTestCase(unittest.TestCase):
    def test_addition(self):
```

```
        self.assertEqual(1 + 1, 2)

# Tạo một module kiểm thử
class MyTestModule(unittest.TestCase):
    def test_subtraction(self):
        self.assertEqual(3 - 1, 2)

# Tạo một đối tượng TestLoader
test_loader = unittest.TestLoader()

# Tải các kiểm thử từ module
test_suite = test_loader.loadTestsFromModule(MyTestModule)

# Thêm các kiểm thử từ lớp kiểm thử vào TestSuite
test_suite.addTests(test_loader.loadTestsFromTestCase(MyTe

# Tạo trình chạy kiểm thử
test_runner = unittest.TextTestRunner()

# Chạy TestSuite
test_runner.run(test_suite)
```

▼ Kết quả test

Các kết quả/ thông tin về test có thể được lấy tại class TestResult

Bổ sung:

Quy trình TDD, thư viện unittest, code mock test

▼ Mock trong Unittest

Trong `unittest`, `mock object` được sử dụng để tạo ra các đối tượng giả mạo (mock) có thể thay thế các đối tượng thực tế trong các kiểm thử. Điều này giúp kiểm tra các hành vi của phần mềm trong môi trường kiểm thử mà không phụ thuộc vào các thành phần khác hoặc các dịch vụ ngoại vi thực sự

▼ Ví dụ cơ bản:

```
# sum_and_print.py
def sum_and_print(a, b):
    total = a + b
    print("Total:", total)
    return total

# test.py
import unittest
from unittest.mock import patch
from sum_and_print import sum_and_print

class TestSumAndPrint(unittest.TestCase):
    @patch('builtins.print')  # Thay thế hàm print với một
    def test_sum_and_print(self, mock_print):
        result = sum_and_print(2, 3)
        mock_print.assert_called_once_with("Total:", 5)
        self.assertEqual(result, 5)

if __name__ == '__main__':
    unittest.main()
```

Giả sử bạn có một chức năng đơn giản là tính tổng của hai số và sau đó in ra kết quả. Bạn muốn viết một kiểm thử cho hàm này mà không cần thực sự gọi hàm `print`, nhưng vẫn có thể kiểm tra xem nó đã được gọi với đúng kết quả hay không.

▼ Patch decorator

▼ Một số phương thức thường sử dụng