Unit Test

1. Giới thiệu về Pytest

Pytest là một framework kiểm thử mạnh mẽ và linh hoạt trong Python, được sử dụng rộng rãi để viết và thực thi các test case cho các ứng dụng phần mềm. Pytest hỗ trợ kiểm thử đơn vị (unit testing), kiểm thử tích hợp (integration testing) và kiểm thử hồi quy (regression testing).

Các tính năng nổi bật của Pytest

- **Cú pháp đơn giản**: Không cần kế thừa từ unittest.TestCase, chỉ cần viết các hàm kiểm thử bắt đầu bằng test_tên_hàm.
- **Hỗ trợ kiểm tra ngoại lệ**: Dễ dàng kiểm tra các lỗi bằng pytest.raises().
- **Cơ chế đánh dấu (markers)**: Cho phép gán nhãn các test case để kiểm soát việc chạy kiểm thử.
- **Hỗ trợ fixture**: Cung cấp khả năng thiết lập dữ liệu trước khi test và dọn dẹp sau khi test.
- **Báo cáo lỗi chi tiết**: Khi một test case thất bại, Pytest hiển thị thông tin chi tiết giúp dễ dàng xác định nguyên nhân lỗi.

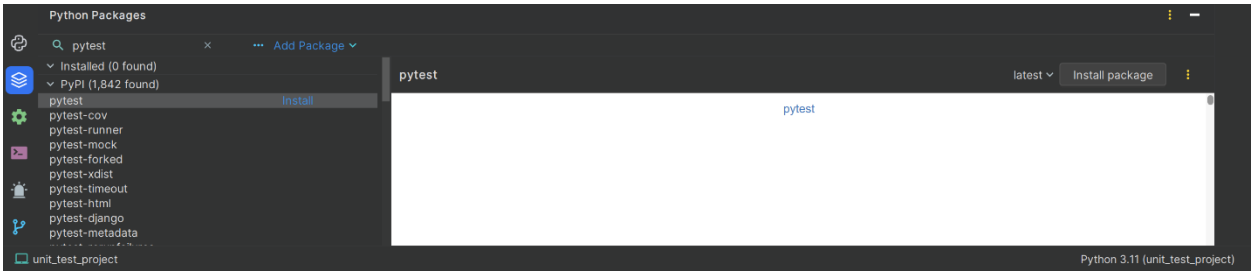
2. Cài đặt

Bước 1: Cài đặt thư viện **pytest** (có thể gõ lệnh bằng terminal hoặc trong phần Package ở PyCharm)

py -m pip install pytest

```
PS D:\Pycharm Projects\unit_test_project> py -m pip install pytest
```

Cài đặt thông qua Package



Bước 2: Trong thư mục gồm có 3 file:
+ *main.py:* dùng để thực thi chương trình
+ *đối_tượng/chức năng.py*: dùng để viết các hàm cho một đối tượng
+ *test_đối_tượng/chức_năng.py:* dùng để kiểm thử cho các hàm trong file *đối_tượng/chức_năng.py*

***Lưu ý:**
- Tên các hàm kiểm thử phải bắt đầu bằng **test_**
- File *đối_tượng/chức năng*.py và *test_ đối_tượng/chức_năng.py* phải ở cùng 1 folder để có thể thực hiện kiểm thử.

Ví dụ sau đây là minh họa cho đối tượng phân số đi kèm với các chức năng cộng trừ nhân chia phân số.

File *đối tượng/chức năng.py

```
# Tìm ước chung
def gcd(a: int, b: int) -> int:
    while b:
        a, b = b, a % b
    return abs(a)

# Tối giản phân số
def simplify(numerator: int, denominator: int) -> tuple:
    common_divisor = gcd(numerator, denominator)
    return numerator // common_divisor, denominator // common_divisor

# Nhập phân số
def input_fraction(numerator: int, denominator: int) -> tuple:
    if not isinstance(numerator, int) or not isinstance(denominator, int):
        raise TypeError("Numerator and denominator must be integers")
    if denominator == 0:
        raise ValueError("Denominator cannot be zero")
    return simplify(numerator, denominator)

# Cộng phân số
def add_fractions(frac1: tuple, frac2: tuple) -> tuple:
    numerator = frac1[0] * frac2[1] + frac2[0] * frac1[1]
```

```
    denominator = frac1[1] * frac2[1]
    return simplify(numerator, denominator)

# Trừ phân số
def subtract_fractions(frac1: tuple, frac2: tuple) -> tuple:
    numerator = frac1[0] * frac2[1] - frac2[0] * frac1[1]
    denominator = frac1[1] * frac2[1]
    return simplify(numerator, denominator)


# Nhân phân số
def multiply_fractions(frac1: tuple, frac2: tuple) -> tuple:
    numerator = frac1[0] * frac2[0]
    denominator = frac1[1] * frac2[1]
    return simplify(numerator, denominator)


# Chia phân số
def divide_fractions(frac1: tuple, frac2: tuple) -> tuple:
    if frac2[0] == 0:
        raise ValueError("Cannot divide by zero")
    numerator = frac1[0] * frac2[1]
    denominator = frac1[1] * frac2[0]
    return simplify(numerator, denominator)
```

File *main.py

```
from fraction import (
    input_fraction,
    add_fractions,
    subtract_fractions,
    multiply_fractions,
    divide_fractions,
)


def main():
    print("Fraction Calculator")
    num1 = int(input("Enter numerator of first fraction: "))
    den1 = int(input("Enter denominator of first fraction: "))
    num2 = int(input("Enter numerator of second fraction: "))
    den2 = int(input("Enter denominator of second fraction: "))

    try:
        frac1 = input_fraction(num1, den1)
        frac2 = input_fraction(num2, den2)
        print(f"Addition: {add_fractions(frac1, frac2)}")
        print(f"Subtraction: {subtract_fractions(frac1, frac2)}")
        print(f"Multiplication: {multiply_fractions(frac1, frac2)}")
        print(f"Division: {divide_fractions(frac1, frac2)}")
    except ValueError as e:
        print(f"Error: {e}")


if __name__ == "__main__":
    main()
```

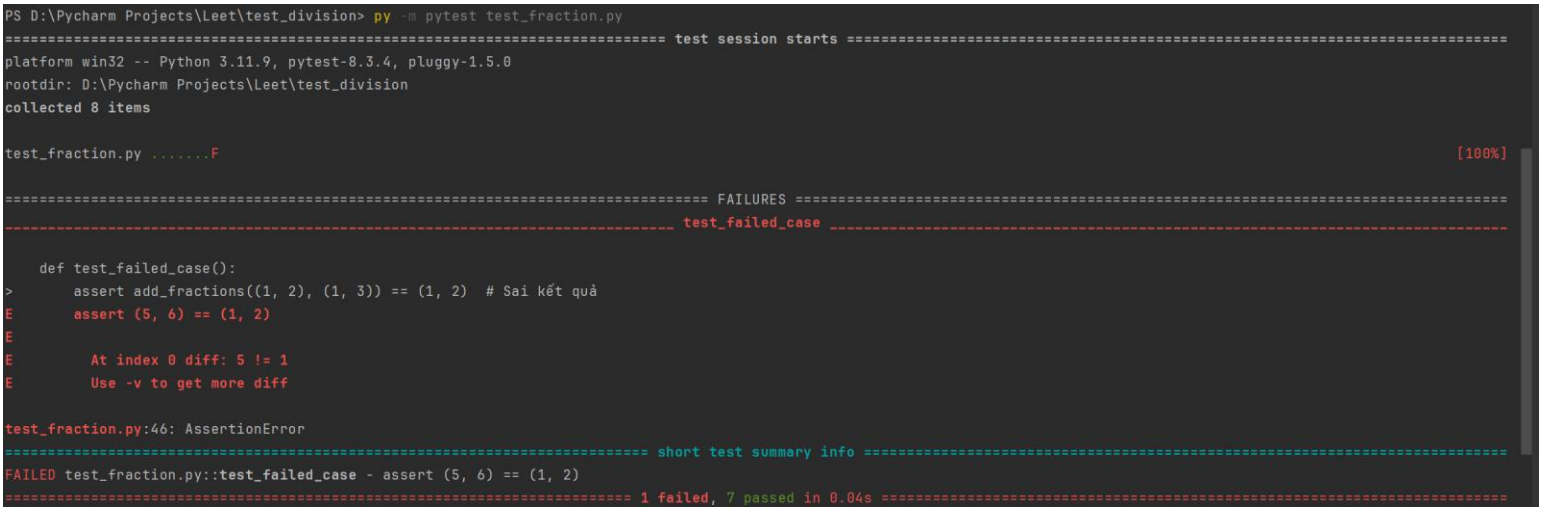
File *test_đối_tượng/chức_năng.py

```
import pytest
from fraction import (
    input_fraction,
    add_fractions,
    subtract_fractions,
    multiply_fractions,
    divide_fractions,
)


def test_add_fractions():
    assert add_fractions((1, 2), (1, 3)) == (5, 6)


def test_subtract_fractions():
    assert subtract_fractions((1, 2), (1, 3)) == (1, 6)


def test_multiply_fractions():
    assert multiply_fractions((1, 2), (1, 3)) == (1, 6)


def test_divide_fractions():
    assert divide_fractions((1, 2), (1, 3)) == (3, 2)


def test_divide_by_zero():
    with pytest.raises(ValueError, match="Cannot divide by zero"):
        divide_fractions((1, 2), (0, 1))


def test_input_fraction():
    assert input_fraction(3, 4) == (3, 4)
    assert input_fraction(2, 4) == (1, 2)
    with pytest.raises(ValueError, match="Denominator cannot be zero"):
        input_fraction(1, 0)
```

```
def test_invalid_fraction_format():
    with pytest.raises(TypeError, match=".*must be integers.*"):
        input_fraction(1.5, 2)


# Thiết kế test case sai kết quả
def test_failed_case():
    assert add_fractions((1, 2), (1, 3)) == (1, 2)  # Sai kết quả
```

Bước 3: Tiến hành kiểm thử bằng cách gõ lệnh sau vào Terminal

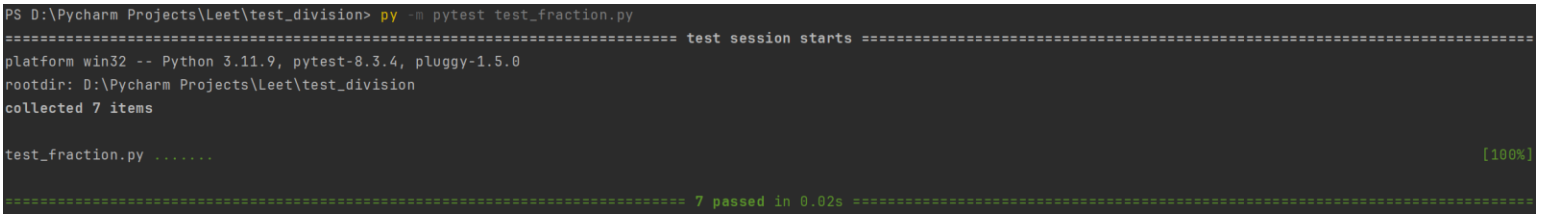
py -m pytest *test_đối_tượng/chức_năng.py*



Số dấu chấm . màu xanh lá tượng trưng cho số test case đã passed

Số chữ F màu đỏ tượng trưng cho số test case đã failed.

Dưới đây là hình minh họa trong trường hợp đã passed hết tất cả các test case.



3. Các chức năng phổ biến của pytest

3.1 Hỗ trợ kiểm tra ngoại lệ - pytest.raises()

- Kiểm tra xem một hàm có ném ra lỗi mong muốn hay không.

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

def test_divide_by_zero():
    with pytest.raises(ValueError, match="Cannot divide by zero"):
        divide(5, 0)
```

3.2 Chuẩn bị dữ liệu test - @pytest.fixture

- Dùng @pytest.fixture để tạo dữ liệu cần thiết cho các test case.

- Giúp tránh lặp lại code setup cho từng test case.

3.3 Markers – Gắn thẻ test case để tổ chức chạy test

Sử dụng @pytest.mark.<tên_marker> để đánh dấu test case, sau đó ta có thể chạy các test case cụ thể thông qua marker.

```
@pytest.mark.slow
def test_heavy_computation():
    import time
    time.sleep(5)
    assert True
```

Sử dụng lệnh: **pytest -m <tên thẻ>**

3.4 Chạy lại test bị lỗi hoặc dừng kiểm thử sau khi đạt n lỗi

pytest -m –maxfail=2 <file kiểm thử>

***Dưới đây là tổng hợp một số lệnh pytest phổ biến:**

```
pytest test_example.py          # Chạy tất cả test trong file test_example.py
pytest -v                       # Hiển thị chi tiết kết quả kiểm thử
pytest -s                       # Hiển thị output từ các câu lệnh print()
pytest --maxfail=2              # Dừng kiểm thử sau 2 lỗi đầu tiên
pytest -m marker_name           # Chạy các test có marker cụ thể
pytest --durations=3            # Hiển thị 3 test chạy chậm nhất
pytest --disable-warnings       # Bỏ qua cảnh báo khi chạy test
```