skypro

Тестирование с pytest. Шпаргалка

Тестирование

Тестирование ПО — это проверка, соответствует ли реальное поведение программы ожиданиям, проводится на наборе тестов.

Инструкция assert

`assert` — специальная инструкция, которая принимает на вход выражение, значением которого должно быть `True`, иначе выбрасывается исключение. Реализует шаблон: условие → исключение, если условие не выполнилось.

Синтаксис использования `assert`:

```
assert выражение, сообщение_об_ошибке
```

- `выражение` — это условие, которое вы проверяете. Если выражение оценивается как `False`, то генерируется ошибка `AssertionError`.
- `сообщение_об_ошибке` (опционально) — это сообщение, которое будет выводиться при возникновении `AssertionError`. Это помогает понять, почему тест не прошел.

Тестирование с `assert`:

```
def add_numbers(num1, num2):
    """Функция, складывающая два числа"""
    return num1 + num2

def is_even(num):
    """Функция, проверяющая, является ли число четным"""
    if num % 2 == 0:
        return True
    else:
        return False

def find_max(numbers):
    """Функция, находящая максимальное значение из списка чисел"""
    max_num = numbers[0]
    for num in numbers:
        if num > max_num:
            max_num = num
    return max_num

assert add_numbers(2, 3) == 5
assert is_even(4) == True
assert is_even(3) == False
assert find_max([1, 5, 3, 8, 2]) == 8
```

Фреймворк pytest

Файл `code.py`:

```
def up_first(msg):
    """Делает первую букву строки заглавной."""
    if msg:
        return msg[0].upper() + msg[1:]
    else:
        return msg
```

Файл `test_code.py` в директории `tests`:

```
from code import up_first

def test_up_first():
    assert up_first('skypro') == 'Skypro'

def test_up_first_for_empty():
    assert up_first('') == ''
```

Тестирование ошибок

```
with pytest.raises(ValueError) as exc_info:
    some_function()
```

- `pytest.raises` — функция из библиотеки `pytest`, которая ожидает, что последующий блок кода вызовет исключение. Это утверждение используется для тестирования «отрицательных» сценариев, где ошибка является ожидаемым результатом.
- Аргументы в скобках — это тип исключения, которое мы ожидаем увидеть. Например, `pytest.raises(ValueError)` означает, что мы ожидаем исключение типа `ValueError`.

Фикстуры

Синтаксис создания фикстуры:

```
@pytest.fixture
def fixture_name():
    return значение_которое_вернет_фикстура
```

- `@pytest.fixture` — это специальная конструкция — декоратор — которая указывает, что функция ниже является фикстурой — объектом, который будет автоматически создаваться pytest перед выполнением теста и при необходимости уничтожаться после его выполнения.
- `fixture_name` — имя функции, которое используется для ссылки на фикстуру в тестах.
- `значение_которое_вернет_фикстура` — это значение, которое должно быть создано и использовано в тестах как исходные данные.

Пример фикстуры и тестов:

```
import pytest

# Создаем фикстуру, которая запускается перед каждым тестом
@pytest.fixture
def coll(): # Имя фикстуры — любое
    return ['One', True, 3, [1, 'hello', [0]], 'hi', {}, '', [], False]

# Pytest сам прокидывает результат вызова функции там,
# где она указана в аргументе.
# Имя параметра совпадает с именем фикстуры
def test_func1(coll):
    assert func1(coll) == # тут ожидаемое значение

# Не важно, что предыдущий тест сделал с коллекцией.
# Здесь она будет новая, так как pytest вызывает coll() заново
def test_func2(coll):
    assert func2(coll) == # тут ожидаемое значение
```

conftest.py

> **Файл** `conftest.py` — это файл, который позволяет определять фикстуры, которые могут быть доступны для всех тестов в проекте, без необходимости импортировать их в каждый тестовый файл.

Файл `conftest.py`:

```
import pytest

@pytest.fixture
def number_list():
    return [1, 2, 3, 4, 5]
```

Файл `test_sum.py`:

```
def test_sum(number_list):
    # Проверяем, что сумма чисел в списке равна 15
    assert sum(number_list) == 15
```

Файл `test_max.py`:

```
def test_max(number_list):
    # Проверяем, что максимальное число в списке равно 5
    assert max(number_list) == 5
```

Параметризация

> **Параметризация тестов** — это запуск одного и того же теста с различными входными данными.

> `@pytest.mark.parametrize` — конструкция (декоратор), которая используется для запуска одного и того же теста с различными входными данными.

Синтаксис `@pytest.mark.parametrize`:

```
@pytest.mark.parametrize("параметры", [(значения_1), (значения_2), ..., (значения_n)])
```

- `"параметры"` — это **строка**, содержащая имена параметров, разделенные запятыми. Эти параметры будут использоваться в тестовой функции.

- `[значения]` — **список кортежей**, где каждый кортеж содержит набор значений для параметров. Количество элементов в каждом кортеже должно соответствовать количеству параметров.

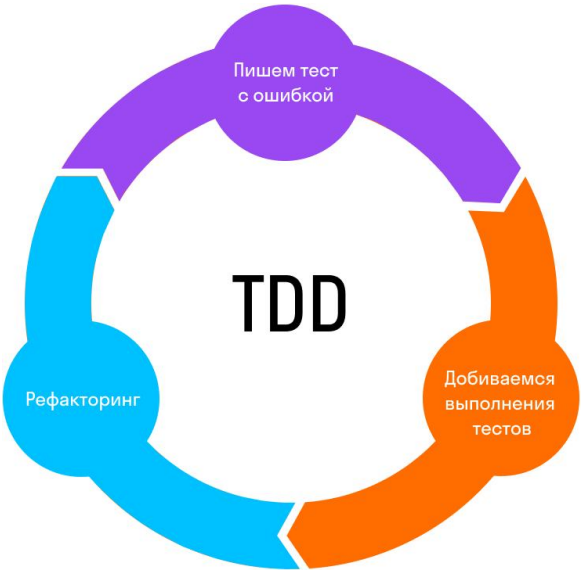
Пример параметризации:

```
import pytest

@pytest.mark.parametrize("x, y, expected", [(1, 2, 3), (4, 5, 9), (7, 8, 15
)])
def test_add(x, y, expected):
    assert x + y == expected
```

TDD (Test Driven Development)

TDD (Test Driven Development) — это методология разработки программного обеспечения, при которой тест пишется до написания кода.



Code coverage

Code coverage — это метрика, которая показывает, какой процент кода программы был протестирован. Это важный инструмент для оценки качества тестирования и выявления проблем в коде.

Установить библиотеку `pytest-cov`:

```
poetry add --group dev pytest-cov
```

Команды, чтобы запустить тесты с оценкой покрытия:

- `pytest --cov` — при активированном виртуальном окружении.
- `poetry run pytest --cov` — через poetry.
- `pytest --cov=src --cov-report=html` — чтобы сгенерировать отчет о покрытии в HTML-формате, где `src` — пакет с модулями, которые тестируем.