Шпаргалка: параметризация в pytest

Иногда нужно протестировать один функционал с разными наборами данных.

Пример. Ты проверяешь поле, куда пользователи вводят ID. Он всегда состоит из цифр. Если ввести другие символы, должно появиться сообщение об ошибке.

Чтобы в этом убедиться, можно провести негативное тестирование. Неподходящих комбинаций будет пять:

```
# набор тестовых данных для негативных проверок
negative_cases = ['abcde', '1abcd', '123$#', '     ', '123 4']
```

Чтобы проверить все комбинации, придётся написать пять одинаковых тестов. Они будут различаться только тестовыми данными. Каждый из них будет выглядеть так:

```
def test_negative_input_1:
    assert not check_input('abcde')    # функция check_input возвращает False, если
                                       # введённые символы некорректны
```

Всего получится пять тестов:

- `test_negative_input_1`;
- `test_negative_input_2`;
- `test_negative_input_3`;
- `test_negative_input_4`;
- `test_negative_input_5`.

Каждый проверяет одну комбинацию. В итоге у тебя будет много одинакового кода. Его неудобно читать, хранить и редактировать.

> 📌 Поэтому когда автотесты различаются только тестовыми данными, применяют **параметризацию**.

Что такое параметризация

Это способ, который позволяет выполнить тест с разными значениями. С параметризацией ты можешь написать одну тестовую функцию и запускать её с разными данными.

Параметризация делит автотест на две части: на логику и тестовые данные. Логика теста — это весь код, кроме тестовых данных: все импорты, действия и проверки. А тестовые данные собирают в один набор и передают в тест по очереди как аргументы функции.

Как написать параметризованный тест

Для этого в pytest применяют декоратор `@pytest.mark.parametrize`.

> 📌 Декораторы — это особые функции. Их используют, чтобы расширять возможности других функций.

Декораторы не влияют на исходный код: они берут функцию как единое целое и делают с ней что-то дополнительное. Например, с помощью декоратора можно замерить время выполнения теста. Или сделать его параметризованным.

Как использовать декоратор `@pytest.mark.parametrize`

Декоратор `@pytest.mark.parametrize` отделяет тестовые данные от логики теста. Он позволяет передавать тестовые данные в функцию по очереди, как аргументы.

Чтобы использовать декоратор:

1. Напиши декоратор `@pytest.mark.parametrize` над тестовой функцией.
2. Поставь круглые скобки. В них через запятую передай два аргумента:
    - название параметра в кавычках;
    - список с тестовыми данными.
3. Функции передай параметр в качестве аргумента. Здесь кавычки не нужны.

Вместе получится так:

Параметризованный тест с одним параметром

```
import pytest
# и другие нужные импорты

@pytest.mark.parametrize('параметр', [тестовые_данные_1, тестовые_данные_2])
def<имя тестовой функции>(<параметр>)
# дальше идёт тело теста
```

Пример теста с декоратором. Тест проверяет поле для ввода ID. Оно должно принимать только цифры. Чтобы в этом убедиться, нужно проверить пять некорректных комбинаций: `'abcde'`, `'1abcd'`, `'123$#'`, `'     '` и `'123 4'`.

С параметризацией получится так:

```
import pytest # декоратор встроен в pytest, поэтому нужен импорт


@pytest.mark.parametrize('id_input', ['abcde', '1abcd', '123$#', '     ', '123 4'])
def test_negative_input(id_input):
    assert not check_input(id_input)
```

Если запустить тест в PyCharm, получится такой вывод:

```
test_params.py::test_negative_input[abcde] PASSED                        [ 20%]
test_params.py::test_negative_input[1abcd] PASSED                        [ 40%]
test_params.py::test_negative_input[123$#] PASSED                        [ 60%]
test_params.py::test_negative_input[     ] PASSED                        [ 80%]
test_params.py::test_negative_input[123 4] PASSED                        [100%]
```

Запустился не один тест, а пять разных. Не пришлось дублировать код и делать копии одного теста с разными данными.

Как называть параметры

Старайся называть параметры просто, понятно и однозначно. Чтобы в любой момент прочитать тест и понять, что именно он делает.

Например:

✅ `login` — если проверяешь ввод логина;

✅ `address` — если проверяешь ввод адреса.

❌ Не называй параметры `test_input` или `data_input`. Это слишком абстрактные имена: они не помогут тебе прочитать код. Представь, что тебе придётся вернуться к работе над тестом спустя пару месяцев. Такие параметры добавят ненужных вопросов: непонятно, какие данные и куда вводит программа.

Когда не нужно использовать параметризацию

> 📢 Не всегда параметризация — это идеальное решение. Бывают случаи, когда лучше продублировать код и написать ещё один тест.

Иногда в параметризации не обойтись без условия `if`. Например, когда для одного из кейсов появляется особая проверка или действие.

Пример. Представь, что ты проверяешь работу кнопки в зависимости от прав пользователя. Уровней доступа всего три: пользователь, менеджер и админ. Если у пользователя админские права, клик по кнопке должен перевести его на новую страницу. В остальных случаях страница не откроется.

В тесте `test_user_rights` есть параметризация. Для `manager` и `customer` проверки совпадают, а для `admin` добавилось отдельное условие.

> 👉 Это пример, как делать **не нужно**:

```
import pytest


@pytest.mark.parametrize('user', ['manager', 'customer', 'admin'])
def test_user_rights(user):
    ... # тело теста
    if user == 'admin':
        ... # проверка только для админа
    ... # тело теста
```

Условия в теле теста — это плохая практика. Действие в `if` может сломать следующие тесты. К тому же тесты должны всегда делать одно и то же. А тут получается, что тест в одном кейсе совершает одни действия, а в следующем — уже другие. Поэтому такой сценарий выносят в отдельный тест.

Если написать отдельный тест для `admin`, получится больше кода. Но зато ты избавишься от условия.

👉 Так будет правильнее:

```
import pytest


@pytest.mark.parametrize('user', ['manager', 'customer'])
def test_user_rights(user):
    ... # тело теста

def test_user_rights_admin():
    ...  # тело теста для пользователя с правами админа
```

Более сложная параметризация

Параметризованный тест с одним параметром — это самый простой вариант. На практике могут встретиться случаи посложнее. Например, у тестового метода будет один параметр, но придётся использовать много тестовых данных. Или, наоборот, данных будет не так много, а вот параметров — больше одного.

Если тестовых данных много

Иногда нужно проверить много тестовых данных.

Пример. Ты тестируешь метод `check_address()`. Он отвечает за проверку адресов на сайте. Чтобы написать на него параметризованный тест, нужно передать в декоратор огромный список с адресами:

```
@pytest.mark.parametrize(                   # Всё это — декоратор...
     'address',
    [
    'Тверская улица, дом 13',
    'улица Академика Колмогорова, дом 7',
    'Мышкинский проезд, дом 95',
    'Стандартная улица, дом 21',
    'Набережная реки Фонтанки, дом 154'
    ]
)
def test_check_user_data(address):          # Тут начинается тестовая функция
    assert check_address(address)
```

Такой код сложно читать: декоратор занимает столько места, что функцию почти не видно.

👉 Поэтому, если тестовых данных много, их **выносят в отдельный объект**.

Например, в переменную:

```
# создали переменную для тестовых адресов
addresses = [
    'Тверская улица, дом 13',
    'улица Академика Колмогорова, дом 7',
    'Мышкинский проезд, дом 95',
    'Стандартная улица, дом 21',
    'Набережная реки Фонтанки, дом 154'
]

# передали в декоратор список в виде переменной
@pytest.mark.parametrize('address', addresses)
def test_check_user_data(address):
    assert check_address(address)
```

Это нужно, чтобы тебе и твоим коллегам было проще читать код. Чаще всего так делают, если параметр — это строка. Они нередко занимают много места.

```
import pytest

переменная = [тестовые_данные_1, тестовые_данные_2, тестовые_данные_3]
                                  1                  2
@pytest.mark.parametrize('параметр', переменная)
def<имя тестовой функции>(<параметр>)
# дальше идёт тело теста
```

Если параметра два

Представь ситуацию, когда нужно протестировать метод с двумя параметрами.

Пример. Ты проверяешь поле для загрузки файлов. Туда можно загрузить файлы с расширением `.pdf` или `.txt`. Расширение файла распознаётся по его имени — `filename`.

За размер файла отвечает параметр `file_size`. Файл типа `pdf` может весить до 5000 Кб, а `txt` — до 1000 Кб. Метод `check_file_size()` проверяет это условие:

```
def check_file_size(filename, file_size):
    # условие: имя файла оканчивается на '.pdf'
    if filename.endswith('.pdf'): # метод endswith() проверяет совпадение с последними символами строки
        return file_size < 5000   # размер не может превышать 5000 Кб

    # условие: имя файла оканчивается на '.txt'
    if filename.endswith('.txt'):
        return file_size < 1000  # размер не может быть больше 1000 Кб

    return False  # если файл больше лимита, возвращается False
```

Тестовая функция будет принимать два параметра: `filename` и `file_size`. Поэтому каждое значение тестового набора — это список из двух элементов. Первый элемент в нём связан с первым параметром функции, а второй — со вторым:

- `['great_gatsby.txt', 800]`;
- `['crazy_python.pdf', 1500]`;
- `['memology.txt', 333]`.

Тест получится таким:

```
import pytest


@pytest.mark.parametrize(
    'filename,file_size',            # Параметры передали в декоратор в виде единой строки
    [
        ['great_gatsby.txt', 800],   # Тестовые данные передали вторым аргументом,
        ['crazy_python.pdf', 1500],  # они тут в виде списка списков
        ['memology.txt', 333]
    ]
)
def test_files(filename, file_size):  # Тестовой функции передали оба аргумента
    assert check_file_size(filename, file_size) # Тестируемая функция с ассертом
```

Что в декораторе. Внутри `@pytest.mark.parametrize` указаны названия параметров, в кавычках через запятую. Обрати внимание:

> 📌 Параметры в декораторе — это подстроки одной строки. Например, `'filename,file_size'`.

Ты можешь поставить пробел после запятой, а можешь не ставить: `'filename, file_size'`. Это ни на что не влияет. А вот запятая между параметрами обязательна: без неё код не заработает.

Тестовые данные в декораторе — это один объект. Функция должна по очереди перебирать его элементы. Поэтому тестовые данные даны в виде списка списков. Его элементы — это отдельные списки. Первый элемент во вложенных списках относится к первому параметру, а второй — ко второму.

Что в тестовой функции. Ей передали оба параметра в качестве аргументов. Как обычно — без кавычек, через запятую.

Что в теле теста. Там параметры проверяются с помощью `assert` и метода `check_file_size()`. Он возвращает `True` или `False`.

Запуск теста даст такой результат:

```
test_file_size.py::test_files[great_gatsby.txt-800] PASSED              [ 33%]
test_file_size.py::test_files[crazy_python.pdf-1500] PASSED             [ 66%]
test_file_size.py::test_files[memology.txt-333] PASSED                  [100%]
```

Если параметров больше двух

В примере только два параметра. Но их может быть сколько угодно: три, пять, десять. Это зависит от параметров функции или метода, которые ты тестируешь. Как правило, больше трёх-четырёх параметров не делают. Иначе декоратор и функция будут слишком громоздкими.

Параметризацию с несколькими параметрами пишут по одной схеме:

Параметризованный тест с несколькими параметрами

```
import pytest

@pytest.mark.parametrize('параметр_1, параметр_2', [[данные_1, данные_2], [данные_1, данные_2]])
def<имя тестовой функции>(<параметр_1>, <параметр_2>)
# дальше идёт тело теста
```