Тестирование: Pytest | Я.Шпора

Установка библиотеки pytest:

```
pip install pytest==7.1.2
```

Структура файлов и тестов в Pytest

Названия файлов с тестами должны начинаться или заканчиваться строкой `test`: *login_test.py*, *test_view.py*…

Код тестов можно размещать в функциях и в методах классов.

- Названия функций должны начинаться на `test`; подчёркивание необязательно: функцию можно назвать `test()`, `test_template()` или `testview()`; впрочем, последнее название нарушает стиль snake_case, лучше так не делать.
- Названия классов должны начинаться со слова `Test` (например, `TestUrls`); названия методов должны начинаться с `test`. Класс с тест-методами не должен содержать метод `__init__`.

Проверка утверждений в Pytest

Для проверки утверждений применяются выражения с ключевым словом `assert`.

```
# test_example.py
def one_more(x):
    return x + 1


def test_correct():
    assert one_more(4) == 5


def test_fail():
    assert one_more(3) == 5
```

Для проверки того, что исключение действительно вызывается, применяют контекстный менеджер и функцию `pytest.raises()`.

```
import pytest


def division(dividend, divisor):
    return dividend / divisor


def test_zero_division():
    with pytest.raises(ZeroDivisionError):  # Ожидается ошибка деления на
        # При вызове функции с такими аргументами возникнет ошибка.
        result = division(1, 0)
```

Запуск тестов Pytest

Запуск тестов выполняется командой из директории, где лежит файл `pytest.ini`:

```
pytest
```

По этой команде pytest найдёт в текущей и вложенных директориях все файлы, названия которых начинаются с `test_` или заканчиваются на `_test.py` — и выполнит в них

- все функции, названия которых начинаются с `test`;
- в классах, название которого начинается с `T` `est`, вызовет методы, названия которых начинаются с `test`.

Правила обнаружения тестов при желании можно изменить.

Уровень детализации при выводе результатов тестирования `pytest` настраивается ключами `-v` (подробный вывод) и `-vv` (очень подробный вывод).

```
pytest -v
```

Пример вывода:

```
...
_______ test_sort __________

def test_sort():
        """Тестируем функцию get_sort_list()."""
        result = get_sort_list('Яша, Саша, Миша, Даша')
>       assert result == ['Даша', 'Маша', 'Саша', 'Яша']
E       AssertionError: assert [' Даша', ' Маша', 'Саша', 'Яша'] == ['Даш
E         At index 0 diff: ' Даша' != 'Даша'
E         Full diff:
E         - ['Даша', 'Маша', 'Саша', 'Яша']
E         ?            ^
E         + [' Даша', ' Миша', ' Саша', 'Яша']
E         ?   +        + ^      +
...
```

При сравнении строк pytest с помощью символов `+` и `^` показывает отличия:

- `+` — в строке найден символ, которого нет в другой строке;
- `^` — в строках отличаются символы: `а` вместо `и`.

Гибкий запуск тестов

Для выборочного запуска в pytest применяется такой синтаксис:

```
pytest file_name.py::TestClassName::test_method_name
```

Запуск тестов из одного файла

```
pytest file_name.py
```

Запуск тестов только из класса `TestClassName` из файла *file_name.py*.

```
pytest file_name.py::TestClassName
```

Можно запустить сразу несколько тестов, перечислив их через пробел.

```
pytest test_example.py::test_fail test_example.py::test_correct
```

Если в проекте есть несколько директорий с тестами — можно запустить тесты в любой из директорий (или в нескольких сразу, указав их через пробел):

```
pytest pytest_trial
```

- при запуске тестов с ключом `-lf` (он же `--last-failed`) будут выполнены только те тесты, которые провалились в прошлый раз;
- при запуске тестов с ключом `-ff` (он же `--failed-first`) сначала будут выполнены провалившиеся тесты, а после них — все остальные.
- при запуске тестов с ключом `-nf` (он же `--new-first`) сначала будут выполнены новые тесты (которых нет в кеше pytest), а потом все остальные.

Для гибкой настройки и запуска тестов в pytest применяется система **маркеров** — декораторов, которыми можно обернуть тестовые функции, методы, классы или даже целые модули.

Чтобы увидеть полный список доступных маркеров — выполните команду

```
pytest --markers
```

Маркер «не выполнять тест»: `@pytest.mark.skip`. У него есть необязательный параметр `reason`, в котором можно указать сообщение о том, из-за чего тест был пропущен.

```
# Тест с этим маркером будет пропущен.
@pytest.mark.skip(reason='Тест пока не готов, завтра допишу!')
def test_will_be_skipped():
    assert True
```

Маркер «падающих» тестов: `@pytest.mark.xfail`. Применяется, когда какой-то тест надо обозначить как «ожидаемо падающий».

```
import pytest
```

```
@pytest.mark.xfail(reason='Пусть пока падает, завтра починю.')
def test_false():
    assert False
```

Декоратор `@pytest.mark.parametrize` позволяет запускать один тест с разными параметрами. Данные для теста передаются в аргументы декоратора.

```
# test_example.py
import pytest


def one_more(x):
    return x + 1


@pytest.mark.parametrize(
    'input_arg, expected_result',  # Названия аргументов, передаваемых в
    [(4, 5), (3, 5)]  # Список кортежей со значениями аргументов.
)
def test_one_more(input_arg, expected_result):  # Те же параметры, что и
    assert one_more(input_arg) == expected_result


...
```

Если ожидается, что с какими-то параметрами тест должен упасть — эти параметры указываются не в кортеже, а в аргументах функции `pytest.param()`; маркер указывается в именованном аргументе `marks`:

```
...
@pytest.mark.parametrize(
    'input_arg, expected_result',
    [
        (4, 5),
        # На этих данных ожидаем падение теста:
        pytest.param(3, 5, marks=pytest.mark.xfail)
    ],
    ids=['First parameter', 'Second parameter',]
```

```
)
def test_one_more(input_arg, expected_result):
    ...
```

Фикстуры в pytest

Фикстуры в pytest — это функции, обозначенные маркером `@pytest.fixture`.

Если в тестирующей функции необходимо применить объекты из фикстуры — название фикстуры передают в параметры функции.

```
import pytest


@pytest.fixture  # Декоратор, обозначающий, что эта функция - фикстура.
def give_me_a_string():
    return 'Какой чудесный день!'  # Фикстура возвращает строку.


# Если тестовой функции для работы нужна фикстура,
# она указывается в параметрах.
def test_string_fixture(give_me_a_string):
    # Переменная с именем фикстуры содержит в себе объект,
    # который вернула фикстура.
    # Проверим, что в объекте фикстуры объект с индексом [0] - строка "К"
    assert give_me_a_string[0] == 'К'
```

Добавим еще одну фикстуру, которая будет вызывать фикстуру `give_me_a_string()` и «упаковывать» строку в список.

```
import pytest


@pytest.fixture
def give_me_a_string():
    return 'Какой чудесный день!'


# Новая фикстура возвращает список со строкой из первой фикстуры.
```

```
@pytest.fixture
def pack_to_list(give_me_a_string):  # Фикстура может вызывать другие фик
    return [give_me_a_string]


# Тестовая функция использует обе фикстуры и проверяет их содержимое.
def test_string_fixture(pack_to_list, give_me_a_string):
    assert pack_to_list == [give_me_a_string]
```

Одну фикстуру можно применять ко всем тестам. Для этого при объявлении фикстуры в декораторе `@pytest.fixture` указывают параметр `autouse=True`.

```
# test_engine.py
import pytest

from engine_class import Engine


@pytest.fixture
def engine():
    """Фикстура возвращает экземпляр класса двигателя."""
    return Engine()


# Обозначаем фикстуру как автоматически вызываемую.
@pytest.fixture(autouse=True)
def start_engine(engine):
    """Фикстура запускает двигатель."""
    engine.is_running = True


# Вызываем только одну фикстуру.
# Запуск двигателя выполнится автоматически, без вызова.
def test_engine_is_running(engine):
    """Тест проверяет, работает ли двигатель."""
    assert engine.is_running
```

Фикстуры в pytest можно хранить в специальном файле с зарезервированным названием *conftest.py*; обычно этот файл кладут в корень проекта.

Pytest для Django

Для работы с Django есть плагин: pytest-django.

```
pip install pytest-django==4.5.2
```

Плагин подключается к Django-проекту в файле *pytest.ini*: в этом файле указывается расположение файла с настройками проекта — *settings.py*:

```
[pytest]
DJANGO_SETTINGS_MODULE = your_project.settings
```

В pytest-django можно применять и `assert`, и дополнительные встроенные функции, например, `assertRedirects()` или `assertFormError()`.

Фикстуры pytest-django

`client`: создаёт анонимный клиент.

```
def test_with_client(client):
    response = client.get('/')
    assert response.status_code == 200
```

`admin_client`: создаёт **клиент** с авторизованным пользователем с правами суперпользователя.

```
def test_closed_page(admin_client):
    response = admin_client.get('/only-for-users/')
    assert response.status_code == 200
```

`admin_user` создаёт **объект пользователя.**

`django_user_model`: возвращает модель пользователя, указанную в настройках проекта (по умолчанию это модель **auth.User**).

```
def test_with_authenticated_client(django_user_model):
    user = django_user_model.objects.create(username='yanote_user')
```

`db`: используется только для других фикстур — для тех, которым нужен доступ к базе данных. Все перечисленные выше фикстуры, которым нужен доступ к базе, вызывают фикстуру `db` автоматически.