# Pytest

## fixture

**Опредление**

(своими словами) Фикстура - функция / БД , которая может вызываться во время теста целевой функции и может быть доступна для других тестов в зависимости от настроек. То есть это внешние данные или внешний фрагмент кода, который может вызываться для тестирования в целевой функции как ресурс, тк для теста часто нужны некоторые сторонние данные (контекст).

(документация + вики) Test fixture (также называемая «тестовым контекстом») используется для настройки состояния системы и входных данных, необходимых для выполнения теста. Фикстура обеспечивает определенный, надежный и последовательный контекст для тестов. Это может включать среду (например, базу данных, настроенную с известными параметрами) или контент (например, набор данных).

@pytest.fixture

def add():

return 1

def test\_3(add):

assert 1 + add == 2

**Зависимости фикстур**

Могут быть сложные зависимости между фикстурами:

@pytest.fixture

def order():

return []

@pytest.fixture

def append\_first(order):

order.append(1)

@pytest.fixture

def append\_second(order, append\_first):

order.extend([2])

@pytest.fixture(autouse=True)

def append\_third(order, append\_second):

order += [3]

def test\_order(order):

assert order == [1, 2, 3]

При вызове ордер создается пустой массив в test\_order, и после того как основные фикстуры все сделали return вызываются фикстуры с autouse = True:

1. вызывается append\_third
2. сначала импортируется ордер
3. потом импортируется append\_second
4. для того чтобы импортировать append\_second нужно чтобы append\_second выполнила свой алгоритм
5. в append\_second также импортируется ордер, потом импортируется append\_first
6. итд пока не зависимости не достигнут конца
7. К этому моменту в test\_order ордер будет поочереди заполнен числами с помощью фикстур, которые ждали друг друга.

**Область действия фикстур**

scope

@pytest.fixture(scope=)

def add():

return []

def test\_3(add):

add.append(1)

def test\_4(add):

add.append(1)

assert len(add) == 1

## paramatarize

Встроенный декоратор pytest.mark.parametrize позволяет параметризовать аргументы для тестовой функции. Параметризация происходит вручную (неэффективно).

import pytest

@pytest.mark.parametrize("test\_input,expected", [("3+5", 8), ("2+4", 6), ("6\*9", 42)])

def test\_eval(test\_input, expected):

assert eval(test\_input) == expected

# Hypothesis

**Определение**

Гипотезис базируется на идее того, что

1) мы передаём свою функцию обёртки гипотезиса

2) итеративно обёртка тестирует функцию, генерируя и передавая ей в аргумент объекты

3) так варьируя передаваемый аргумент мы может наткнуться на потенциальный баг

Так например мы можем протестировать нашу функцию для набора целых чисел:

@given(..smth…)

def test\_integers(n):

print(f"called with {n}")

assert isinstance(n, int)

test\_integers()

Здесь @given — это стандартная точка входа в Hypothesis. Она принимает стратегию, которая описывает тип входных данных, которые должна принимать декорированная функция. Когда мы вызываем test\_integers, Hypothesis генерирует набор целых чисел.

Инфо

По умолчанию Hypothesis генерирует 100 случайных примеров.

**Генерация**

В зависимости от функции пользователь должен сам определить тип генерируемых объектов (или как в документации Domain):

@given( strategy.domain() )

def func(sample):

@given(strategy.lists(stategy.integers()))

def func(sample):

given - декоратор который создаёт цикл, на каждой итерации которого стратегия генерирует sample и передаёт в функцию.

strategy - стратегия это алгоритм который “знает” как генерировать сэмплы того или иного домена и возвращает эти семплы в ретёрн.

domain - тип объекта ( integer, string, list[integers], list[strings] ).

Вывести примеры генерации:

from hypothesis import strategy as st

print(st.lists(st.integers() | st.floats(allow\_nan=False)).example())

**Настройка генерации**

Тк генерирует сэмплы стратегия, то настройка генеарции происходит через параметры стратегии.

Список всех стратегий и аргументов-настроек для них

[Strategies Reference - Hypothesis 6.135.2 documentation](https://hypothesis.readthedocs.io/en/latest/reference/strategies.html)

1) Доступные настройки для генерации целых чисел

@given(st.integers(0, 200))

def test\_integers(n):

assert n < 50

2) Доступные настройки для генерации текста

@given(strategies.text(alphabet=’abc’, min\_size=0, max\_size=None)

def test\_strs(sample):

**Настройка проведения тестов**

Указать количество семплов которые сгенерируются в конкретном тесте:

from hypothesis import given, settings, strategies as st

@given(st.integers())

@settings(max\_examples=200)

def runs\_200\_times\_instead\_of\_100(n):

pass

Изменить настройки глобально для всех тестов:

from hypothesis import HealthCheck, settings

settings.register\_profile("fast", max\_examples=10)

**Фичи**

Добавить мануально несколько сэмплов и протестировать их перед основной генерацией:

@example("Hello world")

@example("some string with special significance")

@given(st.text())

def test\_strings(s):

pass

Добавление своих собственных стратегий

import pytest

from hypothesis import assume, given, strategies as st

@st.composite

def sums\_to\_n(draw, n=1):

lst = draw(st.lists(st.floats(0, 1), min\_size=1))

assume(sum(lst) > 0)

return [f / sum(lst) \* n for f in lst]

@given(sums\_to\_n(10))

def test(lst):

assert sum(lst) == pytest.approx(10)

Генерация данных прямо по середине теста

import pytest

from hypothesis import given, strategies as st

@given(st.data())

def test(data):

lst = data.draw(st.lists(st.floats(0.001, 1), min\_size=1))

lst = [f / sum(lst) for f in lst]

assert sum(lst) == pytest.approx(1)

**Распределение генерации**

Hypothesisследует философии, согласно которой недопустимо разглашать распределение генерации чисел, текстов и прочих типов данных. Пользователь может контролировать диапазон и количество тестов, но распределение или же то, какие именно семплы и как часто будут использоваться в тестах, должно быть недоступно ему. Это позволяет генерировать более “неожиданные” кейсы и убирает элемент предвзятости пользователя, чтобы проверить его функции на действительно всех edge кейсах.

Поэтому нельзя заранее знать какие значения будут генерироваться в тестах, но можно задать диапазон и другие параметры.

**Shrinking**

Шринкинг - это механизм гипотезис, который активируется при падении теста. Когда гипотезис обнаруживает ошибку, то он проводит дополнительные проверки, изменяя и минимизируя запрос вызвавший ошибку, чтобы найти конкретный элемент, вызывающий ошибку. Например если в текстовом аргументе есть арабская буква, то гипотезис найдет её и скажет что именно она вызывает ошибку.

Или пример функции вызывающую ошибку при обнаружении отрицательного числа во входном списке:

from hypothesis import given, strategies as st

@given(st.lists(st.integers()))

def test\_no\_negative\_numbers(xs):

for x in xs:

assert x >= 0

Гипотезис находит минимальное но вызывающее ошибку значение

Ответ пайтеста:

FAILED hypo.py::test\_no\_negative\_numbers - assert -1 >= 0

=================== 1 failed, 2 passed, 1 warning in 1.14s =

**Советы**

Следует проводить Round Tip тестирование, которое проверяет корректность сериализацию и десериализацию:

decode(encode(x)) == x

json.loads(json.dumps(data)) == data

Пример теста:

import base64

from hypothesis import given, strategies as st

@given(st.binary()) # генерируем произвольные байты

def test\_base64\_round\_trip(data: bytes):

encoded = base64.b64encode(data)

decoded = base64.b64decode(encoded)

assert decoded == data

# Schemathesis

Документация

[Welcome to Schemathesis! — Schemathesis 3.39.16 documentation](https://schemathesis.readthedocs.io/en/stable/index.html)

Осуществляет множество запросов по адресу API, если у него есть swagger и openAPI схема. На основе этой схемы он формирует множество вариантов тестов.

## Как работает конкретно

## CLI

Схематезис имеет интерфейс в консоли и позволяет так через консоль тестировать сайты для которых написаны документации swagger 3.0.х.:

schemathesis run <http://localhost:8080/openapi.json>

А также swagger 3.1.х. в экспериментальном режиме:

schemathesis run http://localhost:8080/openapi.json --experimental=openapi-3.1

### флаги

А также существует много флагов контролирующих количество проводимых тестов итд

--hypothesis-max-examples=1000

## Python

### 101

Более гибкое использование схематезис представлено в виде пайтон библиотеки.

Точно такой же функционал как и при schemathesis run <http://localhost:8080/openapi.json> можно реализовать как:

import schemathesis

schema = schemathesis.from\_uri('http://localhost:8080/openapi.json',)

@schema.parametrize()

def test\_app(case):

    case.call\_and\_validate()

Для схем 3.1. достаточно просто добавить в начале:

schemathesis.experimental.OPEN\_API\_3\_1.enable()

**Импорт схемы**

Стандартный импорт из работающего онлайн или локально веб приложения имеющего сваггер:

from hypothesis import settings

import schemathesis

schemathesis.experimental.OPEN\_API\_3\_1.enable()

schema = schemathesis.from\_uri('http://localhost:8000/openapi.json',)

Удобный импорт схемы прямо из объекта фастапи и фласк:

from project import app

# WSGI

schema = schemathesis.from\_wsgi("/api/openapi.json", app)

# Or ASGI

schema = schemathesis.from\_asgi("/api/openapi.json", app)

**Выбор эндпоинтов**

C помощью include exclude name можно контролировать то какие эндпоинты мы проверяем.

@schema.exclude(name = 'POST /todos2/').parametrize()

def test\_app(case):

case.call\_and\_validate()

**Настройки проведения тестов**

*Локально*

Указать количество генерируемых семплов для конкретного теста:

@schema.parametrize()

@settings(max\_examples=10)

def test\_app(case):

    case.call\_and\_validate()

*Глобально*

**Настройки генерации**

Так как schemathesis основан на hypothesis, то в schemathesis доступны все можно применять все способы манипуляций доступных в hypothesis

**Фичи**

Подставление нужных значений в конкретные параметры эндпоинтов:

@schema.parametrize()

@schema.override(path\_parameters={"user\_id": "42"}, query={"apiKey": "secret"})

def test\_api(case):

**Настройка тестов валидации**

По умолчанию при case.call\_and\_validate() выполяются следующие проверки:

from schemathesis.checks import (

not\_a\_server\_error,

status\_code\_conformance,

content\_type\_conformance,

response\_schema\_conformance,

negative\_data\_rejection,

response\_headers\_conformance,

use\_after\_free,

ensure\_resource\_availability,

ignored\_auth)  
  
Но можно сделать только выборочные проверки или добавить свои

def my\_check(response, case):

... # some awesome assertions

@schema.parametrize()

def test\_api(case):

case.call\_and\_validate(checks=(not\_a\_server\_error,), additional\_checks = (my\_check,))