

 AlekSandrDr вчера в 17:11

# Python Testing с pytest. ГЛАВА 3 pytest Fixtures

Автор оригинала: Okken Brian

Python

Перевод

Tutorial



Эта книга — недостающая глава, отсутствующая в каждой всеобъемлющей книге Python.

Frank Ruiz

Principal Site Reliability Engineer, Box, Inc.



The  
Pragmatic  
Programmers

## Python Testing with pytest

Simple, Rapid,  
Effective, and  
Scalable

Brian Okken  
*edited by Katharine Dvorak*



Примеры в этой книге написаны с использованием Python 3.6 и pytest 3.2. pytest 3.2 поддерживает Python 2.6, 2.7 и Python 3.3+

Исходный код для проекта Tasks, а также для всех тестов, показанных в этой книге, доступен по [ссылке](#) на веб-странице книги в [pragprog.com](#). Вам не нужно загружать исходный код, чтобы понять тестовый код; тестовый код представлен в удобной форме в примерах. Но что бы следовать вместе с задачами проекта, или адаптировать примеры тестирования для проверки своего собственного проекта (если у вас развязаны!), вы должны перейти на веб-страницу книги и скачать работу. Там же, на веб-странице книги есть ссылка для сообщений об ошибках и дискуссионный форум.

Под спойлером приведен список статей этой серии.

[Оглавление](#)

Теперь, когда вы видели основы pytest, обратим наше внимание на фикстуры, которые необходимы для структурирования тестового кода практически для любой нетривиальной программной системы. Fixtures — это функции, выполняемые pytest до (а иногда и после) фактически тестовых функций. Код в фикстуре может делать все, что вам необходимо. Вы можете использовать Fixtures, чтобы получить набор данных тестирования. Вы можете использовать Fixtures, чтобы получить систему в известном состоянии перед запуском теста. Fixtures также используются для получения данных для нескольких тестов.

Вот простой пример фикстуры, который возвращает число:

**ch3/test\_fixtures.py**

```
import pytest

@pytest.fixture()
def some_data():
    """Return answer to ultimate question."""
    return 42

def test_some_data(some_data):
    """Use fixture return value in a test."""
    assert some_data == 42
```

Декоратор `@pytest.fixture()` используется, чтобы сообщить pytest, что функция является фикстурой. Когда вы включаете имя фикстуры в параметров тестовой функции, pytest знает, как запустить её перед запуском теста. Фикстуры могут выполнять работу, а могут возвращать данные в тестовую функцию.

Тест `test_some_data()` имеет в качестве параметра имя фикстуры `some_data`. pytest определит это и найдет фикстуру с таким названием. Наименование значимо в pytest. pytest будет искать в модуле теста фикстуру с таким именем. Он также будет искать в файле `conftest.py`, и найдет его в этом.

Прежде чем мы начнем наше исследование фикстур (и файла `conftest.py`), мне нужно рассмотреть тот факт, что термин `fixture` имеет много значений в сообществе программирования и тестирования и даже в сообществе Python. Я использую `fixture`, `fixture function`, и `fixture method` взаимозаменяемо, чтобы сослаться на функции `@pytest.fixture()`, описанные в этой главе. Фикстура также может использоваться обозначения ресурса, который ссылается функцией фикстуры. Функции `Fixture` часто настраивают или извлекают некоторые данные, с которыми может работать тест. Иногда эти данные считаются фикстурой. Например, сообщество Django часто использует фикстуру для обозначения некоторых исходных данных, которые загружаются в базу данных в начале приложения.

Независимо от других смысловых значений, в pytest и в этой книге `test fixtures` относятся к механизму, который обеспечивает pytest, чтобы отделить код “подготовка к (getting ready for)” и “очистка после (cleaning up after)” от ваших тестовых функций.

pytest fixtures — одна из уникальных фишек, которые поднимают pytest над другими тестовыми средами, и являются причиной того, почему многие уважаемые люди переключаются на... и остаются с pytest. Тем не менее, фикстуры в pytest отличаются от фикстур в Django и отличаются от процедур `setup` и `teardown`, обнаруженных в `unittest` и `nose`. Есть много особенностей и нюансов если говорить о фикстурах. Как только вы получите хорошую ментальную модель того, как они работают, вам станет легче. Тем не менее, вам нужно поиграться с ними некоторое время, чтобы въехать, поэтому давайте начнем.

## Обмен Fixtures через conftest.py

Можно поместить фикстуры в отдельные тестовые файлы, но для совместного использования фикстур в нескольких тестовых файлах лучше использовать файл `conftest.py` где-то в общем месте, централизованно для всех тестов. Для проекта задач все фикстуры будут находиться в `tasks_proj/tests/conftest.py`.

Оттуда, fixtures могут быть разделены любым тестом. Вы можете поместить fixtures в отдельные тестовые файлы, если вы хотите, чтобы fixture использовался только в тестах этого файла. Аналогично, вы можете иметь другие файлы `conftest.py` в подкаталогах каталога `top tests`. Если это сделаете, fixtures, определенные в этих низкоуровневых файлах `conftest.py`, будут доступны для тестов в этом каталоге и подкаталогах. Однако до сих пор fixtures в проекте «Задачи» были предназначены для любого теста. Поэтому использование всех наших инструментов в `conftest.py` в корне тестирования, `tasks_proj/tests`, имеет наибольший смысл.

Хотя `conftest.py` является модулем Python, он не должен импортироваться тестовыми файлами. Не импортируйте `conftest` ни когда! Файл `conftest.py` считывается pytest и считается локальным плагином, что станет понятно, когда мы начнем говорить о плагинах в главе 5 «Плагины» стр. 95. Пока что считайте `tests/conftest.py` как место где мы можем поместить fixtures, для использования всеми тестами в каталоге тестов. Затем давайте переработаем некоторые наши тесты для `task_proj`, чтобы правильно использовать фикстуры.

## Использование Fixtures для Setup и Teardown

Большинство тестов в проекте Tasks предполагают, что база данных Tasks уже настроена, запущена и готова. И мы должны убрать какие-то записи в конце, если есть какая-то необходимость в очистке. И возможно понадобится также отключиться от базы данных. К счастью, боль

часть этого позаботилась в коде задач с `tasks.start_tasks_db(<directory to store db>, 'tiny' or 'mongo')` И `tasks.stop_tasks_db()` просто требуется вызвать их в нужный момент, и ещё нам понадобится временный каталог.

К счастью, `pytest` включает в себя отличную фикстуру под названием `tmpdir`. Мы можем использовать её для тестирования и не должны беспокоиться о очистке. Это не магия, просто хорошая практика кодирования от самых пытливых людей. (Не переживайте; мы разберем `tmpdir` более подробно распишем его с помощью `tmpdir_factory` в разделе «Использование `tmpdir` и `tmpdir_factory`» на стр. 71.)

С учетом всех этих составляющих, эта фикстура работает замечательно:

`ch3/a/tasks_proj/tests/conftest.py`

```
import pytest
import tasks
from tasks import Task

@pytest.fixture()
def tasks_db(tmpdir):
    """Подключение к БД перед тестами, отключение после."""
    # Setup : start db
    tasks.start_tasks_db(str(tmpdir), 'tiny')

    yield # здесь происходит тестирование

    # Teardown : stop db
    tasks.stop_tasks_db()
```

Значение `tmpdir` не является строкой-это объект, который представляет каталог. Однако он реализует `__str__`, поэтому мы можем использовать `str()`, чтобы получить строку для передачи в `start_tasks_db()`. Пока мы все еще используем "tiny" для TinyDB.

Функция `fixture` запускается перед тестами, которые ее используют. Однако, если в функции есть `yield`, то там произойдет остановка, контроль передается тестам и выполняется следующая за `yield` строка после завершения тестов. Поэтому подумайте о коде над `yield` как о «setup», а о коде после `yield` как о «teardown». Код после `yield` «teardown» будет выполняться независимо от того, что происходит во время тестов. Мы возвращаем данные с выходом в этой фикстуре. Но вы можете.

Давайте изменим один из наших тестов `tasks.add()`, чтобы использовать эту фикстуру:

`ch3/a/tasks_proj/tests/func/test_add.py`

```
import pytest
import tasks
from tasks import Task

def test_add_returns_valid_id(tasks_db):
    """tasks.add(<valid task>) должен возвращать целое число."""
    # GIVEN инициализированная БД задач
    # WHEN добавлена новая задача
    # THEN вернулся task_id типа int
    new_task = Task('do something')
    task_id = tasks.add(new_task)
    assert isinstance(task_id, int)
```

Основное изменение здесь заключается в том, что дополнительная фикстура в файле была удалена, и мы добавили `tasks_db` в список параметров теста. Мне нравится структурировать тесты в формате *GIVEN/WHEN/THEN* (ДАНО/КОГДА/ПОСЛЕ), используя комментарии, если это не очевидно из кода, что происходит. Я думаю, что это полезно в этом случае. Надеюсь, *GIVEN* инициализированные задачи `db` помогут выяснить, почему `tasks_db` используется в качестве инструмента для теста.

Убедитесь, что `Tasks` установлен

Мы все еще пишем тесты для проекта Tasks в этой главе, который был впервые установлен в главе 2. Если вы пропустили эту главу, обязательно установите задачи с `cd code;pip install ./tasks_proj/`.

## Трассировка Fixture Execution с `–setup-show`

Если вы запустите тест из последнего раздела, вы не увидите, какие фикстуры запущены:

```
$ cd /path/to/code/
$ pip install ./tasks_proj/ # если он еще не установлен
$ cd /path/to/code/ch3/a/tasks_proj/tests/func
$ pytest -v test_add.py -k valid_id
===== test session starts =====
collected 3 items
test_add.py::test_add_returns_valid_id PASSED
===== 2 tests deselected =====
===== 1 passed, 2 deselected in 0.02 seconds =====
```

Когда я разрабатываю fixtures, мне необходимо видеть, что работает и когда. К счастью, pytest предоставляет такой флаг для командной строки `– setup-show`, который делает именно это:

```
$ pytest --setup-show test_add.py -k valid_id
===== test session starts =====

collected 3 items / 2 deselected

test_add.py
SETUP      S tmpdir_factory
          SETUP      F tmpdir (fixtures used: tmpdir_factory)
          SETUP      F tasks_db (fixtures used: tmpdir)
          func/test_add.py::test_add_returns_valid_id (fixtures used: tasks_db,
              tmpdir, tmpdir_factory).
          TEARDOWN    F tasks_db
          TEARDOWN    F tmpdir
          TEARDOWN    S tmpdir_factory

===== 1 passed, 2 deselected in 0.18 seconds =====
```

Наш тест находится посередине, а pytest обозначил часть **SETUP** и **TEARDOWN** для каждой фикстуры. Начиная с `test_add_returns_valid` видите, что `tmpdir` работал перед тестом. И до этого `tmpdir_factory`. По-видимому, `tmpdir` использует его как фикстуру.

**F** и **S** перед именами фикстур указывают область. **F** для области действия и **S** для области сеанса. Я расскажу о сфере действия в разделе «Спецификация областей(Scope) Fixture» на стр. 56.

## Использование Fixtures для Test Data

Fixtures являются отличным местом хранения данных для тестирования. Вы можете вернуть всё что угодно. Вот фикстура, возвращающая смешанного типа:

**ch3/test\_fixtures.py**

```
@pytest.fixture()
def a_tuple():
    """Вернуть что-нибудь более интересное"""
    return (1, 'foo', None, {'bar': 23})

def test_a_tuple(a_tuple):
    """Demo the a_tuple fixture."""
    assert a_tuple[3]['bar'] == 32
```

Поскольку `test_a_tuple()` должен завершиться неудачей ( $23 \neq 32$ ), мы увидим, что произойдет, когда тест с фикстурой потерпит неудачу:

```
$ cd /path/to/code/ch3
$ pytest test_fixtures.py::test_a_tuple
===== test session starts =====

collected 1 item

test_fixtures.py F [100%]

===== FAILURES =====
_____ test_a_tuple _____

a_tuple = (1, 'foo', None, {'bar': 23})

    def test_a_tuple(a_tuple):
        """Demo the a_tuple fixture."""
>     assert a_tuple[3]['bar'] == 32
E     assert 23 == 32

test_fixtures.py:38: AssertionError
===== 1 failed in 0.17 seconds =====
```

Вместе с разделом трассировки стека `pytest` отображает параметры значения функции, вызвавшей исключение или не прошедшей `assert`. В случае проведения тестов фикстуры — это параметры для теста, поэтому о них сообщается с помощью трассировки стека. Что произойдет, если `assert` (или `exception`) случится в `fixture`?

```
$ pytest -v test_fixtures.py::test_other_data
===== test session starts =====

test_fixtures.py::test_other_data ERROR [100%]

===== ERRORS =====
_____ ERROR at setup of test_other_data _____

@pytest.fixture()
def some_other_data():
    """Raise an exception from fixture."""
    x = 43
>     assert x == 42
E     assert 43 == 42

test_fixtures.py:21: AssertionError
===== 1 error in 0.13 seconds =====
```

Происходит пара вещей. Трассировка стека правильно показывает, что `assert` произошёл в функции фикстуры. Кроме того, `test_other_data` сообщается не как **FAIL**, а как **ERROR**. Это серьёзное различие. Если тест вдруг терпит неудачу, вы знаете, что сбой произошёл в самом тесте, а не зависит от какой-то фикстуры.

Но как насчет проекта `Tasks`? Для проекта `Tasks` мы, вероятно, могли бы использовать некоторые фикстуры данных, возможно, различные задачи с различными свойствами:

**ch3/a/tasks\_proj/tests/conftest.py**

```
# Памятка об интерфейсе Task constructor
# Task(summary=None, owner=None, done=False, id=None)
# summary то что требуется
# owner и done являются необязательными
# id задается базой данных

@pytest.fixture()
def tasks_just_a_few():
    """Все резюме и владельцы уникальны."""
```

```

    return (
        Task('Write some code', 'Brian', True),
        Task("Code review Brian's code", 'Katie', False),
        Task('Fix what Brian did', 'Michelle', False))

@pytest.fixture()
def tasks_mult_per_owner():
    """Несколько владельцев с несколькими задачами каждый."""
    return (
        Task('Make a cookie', 'Raphael'),
        Task('Use an emoji', 'Raphael'),
        Task('Move to Berlin', 'Raphael'),

        Task('Create', 'Michelle'),
        Task('Inspire', 'Michelle'),
        Task('Encourage', 'Michelle'),

        Task('Do a handstand', 'Daniel'),
        Task('Write some books', 'Daniel'),
        Task('Eat ice cream', 'Daniel'))

```

Вы можете использовать их непосредственно из тестов, или из других фикстур. Давайте создадим с их помощью непустые базы данных для тестирования.

## Использование Multiple Fixtures

Вы уже видели, что `tmpdir` использует `tmpdir_factory`. И вы использовали `tmpdir` в нашем `task_db` fixture. Давайте продолжим цепочку и добавим некоторые специализированные фикстуры для непустых баз проекта `tasks`:

**ch3/a/tasks\_proj/tests/conftest.py**

```

@pytest.fixture()
def db_with_3_tasks(tasks_db, tasks_just_a_few):
    """Подключение БД с 3 задачами, все уникальны."""
    for t in tasks_just_a_few:
        tasks.add(t)

@pytest.fixture()
def db_with_multi_per_owner(tasks_db, tasks_mult_per_owner):
    """Подключение БД с 9 задачами, 3 owners, с 3 задачами у каждого."""
    for t in tasks_mult_per_owner:
        tasks.add(t)

```

Все эти fixtures включают две фикстуры в свой список параметров: `tasks_db` и набор данных. Набор данных используется для добавления базы данных. Теперь тесты могут использовать их, если вы хотите, чтобы тест начинался с непустой базы данных, например:

**ch3/a/tasks\_proj/tests/func/test\_add.py**

```

def test_add_increases_count(db_with_3_tasks):
    """Test tasks.add() должен повлиять на tasks.count()."""
    # GIVEN db с 3 задачами
    # WHEN добавляется еще одна задача
    tasks.add(Task('throw a party'))

    # THEN счетчик увеличивается на 1
    assert tasks.count() == 4

```

Это также демонстрирует одну из главных причин использования fixtures: чтобы сфокусировать тест на том, что вы на самом деле тестируете, а не на том, что вы должны были сделать, чтобы подготовиться к тесту. Мне нравится использовать комментарии для GIVEN/WHEN/THEN и пытаться протолкнуть как можно больше данных (GIVEN) в фикстуры по двум причинам. Во-первых, это делает тест более читаемым и,

следовательно, более ремонтпригодным. Во-вторых, `assert` или `exception` в фикстуре приводит к ошибке (ERROR), в то время как `assert` или `exception` в тестовой функции приводит к ошибке (FAIL). Я не хочу, чтобы `test_add_increases_count()` отказал, если инициализация базы `db` завершилась неудачно. Это просто сбивает с толку. Я хочу, чтобы сбой (FAIL) `test_add_increases_count()` был возможен только в том случае, если `add()` действительно не смог изменить счетчик. Давайте запустим и посмотрим, как работают все фикстуры:

```
$ cd /path/to/code/ch3/a/tasks_proj/tests/func
$ pytest --setup-show test_add.py::test_add_increases_count

===== test session starts =====

collected 1 item

test_add.py
SETUP      S tmpdir_factory
          SETUP      F tmpdir (fixtures used: tmpdir_factory)
          SETUP      F tasks_db (fixtures used: tmpdir)
          SETUP      F tasks_just_a_few
          SETUP      F db_with_3_tasks (fixtures used: tasks_db, tasks_just_a_few)
func/test_add.py::test_add_increases_count (fixtures used: db_with_3_tasks, tasks_db, tasks_just_a_few, tmpdir, tmpdir_factory).
          TEARDOWN   F db_with_3_tasks
          TEARDOWN   F tasks_just_a_few
          TEARDOWN   F tasks_db
          TEARDOWN   F tmpdir
          TEARDOWN   S tmpdir_factory

===== 1 passed in 0.20 seconds =====
```

Получили снова кучу F-ов и S для функции и области сеанса. Давайте разберем, что это.

## Спецификация областей(Scope) Fixture

Фикстуры включают в себя необязательный параметр под названием **scope**, который определяет, как часто фикстура получает `setup` и `teardown`. Параметр `scope` для `@pytest.fixture()` может иметь значения функции, класса, модуля или сессии. `Scope` по умолчанию — это функция. Настройки `tasks_db` и все фикстуры пока не определяют область. Таким образом, они являются функциональными фикстурами.

Ниже приведено краткое описание каждого значения `Scope`:

- `scope='function'`

Выполняется один раз для каждой функции теста. Часть `setup` запускается перед каждым тестом с помощью `fixture`. Часть `teardown` запускается после каждого теста с использованием `fixture`. Это область используемая по умолчанию, если параметр `scope` не указан.

- `scope='class'`

Выполняется один раз для каждого тестового класса, независимо от количества тестовых методов в классе.

- `scope='module'`

Выполняется один раз для каждого модуля, независимо от того, сколько тестовых функций или методов или других фикстур при использовании модуля.

- `scope='session'`

Выполняется один раз за сеанс. Все методы и функции тестирования, использующие фикстуру области сеанса, используют один вызов `setup` и `teardown`.

Вот как выглядят значения `scope` в действии:

**ch3/test\_scope.py**

```

"""Demo fixture scope."""

import pytest

@pytest.fixture(scope='function')
def func_scope():
    """A function scope fixture."""

@pytest.fixture(scope='module')
def mod_scope():
    """A module scope fixture."""

@pytest.fixture(scope='session')
def sess_scope():
    """A session scope fixture."""

@pytest.fixture(scope='class')
def class_scope():
    """A class scope fixture."""

def test_1(sess_scope, mod_scope, func_scope):
    """Тест с использованием сессий, модулей и функций."""

def test_2(sess_scope, mod_scope, func_scope):
    """Демонстрация более увлекательна со множеством тестов."""

@pytest.mark.usefixtures('class_scope')
class TestSomething():
    """Demo class scope fixtures."""

    def test_3(self):
        """Test using a class scope fixture."""

    def test_4(self):
        """Again, multiple tests are more fun."""

```

Давайте используем `--setup-show` для демонстрации, что количество вызовов fixture и setup в паре с teardown выполняются в зависимости области:

```

$ cd /path/to/code/ch3/
$ pytest --setup-show test_scope.py
===== test session starts =====

collected 4 items

test_scope.py
SETUP      S sess_scope
  SETUP      M mod_scope
    SETUP      F func_scope
    test_scope.py::test_1 (fixtures used: func_scope, mod_scope, sess_scope).
    TEARDOWN F func_scope
    SETUP      F func_scope
    test_scope.py::test_2 (fixtures used: func_scope, mod_scope, sess_scope).
    TEARDOWN F func_scope
  SETUP      C class_scope
    test_scope.py::TestSomething::():test_3 (fixtures used: class_scope).
    test_scope.py::TestSomething::():test_4 (fixtures used: class_scope).
    TEARDOWN C class_scope
  TEARDOWN M mod_scope
TEARDOWN S sess_scope

===== 4 passed in 0.11 seconds =====

```

Теперь вы можете видеть не только **F** и **S** для функции и сеанса, но также **C** и **M** для класса и модуля.

Область(scope) определяется с помощью фикстуры. Я знаю, что это очевидно из кода, но это важный момент, чтобы убедиться, что вы пол рокаете (*Прим переводчика: прокать* — скорее всего автор имеет ввиду термин из романа Роберта Хайнлайна "Чужак в стране чужой").



Приблизительное значение "глубоко и интуитивно понимать"). Область(scope) задается в определении фикстуры, а не в месте её вызова. Тестовые функции, которые используют фикстуру, не контролируют, как часто устанавливается(SETUP) и сбрасывается(TEARDOWN) фикстура:

Фикстуры могут зависеть только от других фикстур из той же или более расширенной области(scope). Таким образом, function scope fixture зависит от других function scope fixture (по умолчанию и используется в проекте Tasks до сих пор). function scope fixture также может зависеть от класса, модуля и фикстур области сеанса, но в обратном порядке — никогда.

## Смена Scope для Tasks Project Fixtures

С учетом этих знаний о scope, давайте теперь изменим область действия некоторых фикстур проекта Task.

До сих пор у нас не было проблем со временем тестирования. Но, согласитесь, что бесполезно создавать временный каталог и новое соединение с базой данных для каждого теста. Пока мы можем обеспечить пустую базу данных, когда это необходимо, этого должно быть достаточно.

Чтобы использовать что-то вроде `tasks_db` в качестве области сеанса, необходимо использовать `tmpdir_factory`, так как `tmpdir` является областью функции и `tmpdir_factory` является областью сеанса. К счастью, это всего лишь одна строка изменения кода (ну, две, если вы с `tmpdir->tmpdir_factory` в списке параметров):

**ch3/b/tasks\_proj/tests/conftest.py**

```
"""Define some fixtures to use in the project."""

import pytest
import tasks
from tasks import Task

@pytest.fixture(scope='session')
def tasks_db_session(tmpdir_factory):
    """Connect to db before tests, disconnect after."""
    temp_dir = tmpdir_factory.mktemp('temp')
    tasks.start_tasks_db(str(temp_dir), 'tiny')
    yield
    tasks.stop_tasks_db()

@pytest.fixture()
def tasks_db(tasks_db_session):
    """An empty tasks db."""
    tasks.delete_all()
```

Здесь мы изменили `tasks_db` в зависимости от `tasks_db_session`, и мы удалили все записи, чтобы убедиться, что он пуст. Поскольку мы не изменили его название, ни одна из фикстур или тестов, которые уже включают его, не должен измениться.

Фикстуры данных просто возвращают значение, поэтому действительно нет причин, чтобы они работали все время. Один раз за сеанс достаточно:

**ch3/b/tasks\_proj/tests/conftest.py**

```
# Reminder of Task constructor interface
# Task(summary=None, owner=None, done=False, id=None)
# summary is required
# owner and done are optional
# id is set by database

@pytest.fixture(scope='session')
def tasks_just_a_few():
    """All summaries and owners are unique."""
    return (
        Task('Write some code', 'Brian', True),
        Task("Code review Brian's code", 'Katie', False),
        Task('Fix what Brian did', 'Michelle', False))
```

```
@pytest.fixture(scope='session')
def tasks_mult_per_owner():
    """Several owners with several tasks each."""
    return (
        Task('Make a cookie', 'Raphael'),
        Task('Use an emoji', 'Raphael'),
        Task('Move to Berlin', 'Raphael'),

        Task('Create', 'Michelle'),
        Task('Inspire', 'Michelle'),
        Task('Encourage', 'Michelle'),

        Task('Do a handstand', 'Daniel'),
        Task('Write some books', 'Daniel'),
        Task('Eat ice cream', 'Daniel'))
```

Теперь давайте посмотрим, будут ли все эти изменения работать с нашими тестами:

```
$ cd /path/to/code/ch3/b/tasks_proj
$ pytest
===== test session starts =====
collected 55 items
tests/func/test_add.py ...
tests/func/test_add_variety.py .....
tests/func/test_add_variety2.py .....
tests/func/test_api_exceptions.py .....
tests/func/test_unique_id.py .
tests/unit/test_task.py ....
===== 55 passed in 0.17 seconds =====
```

Похоже, все в порядке. Давайте проследим фикстуры для одного тестового файла, чтобы увидеть, что различные области работают в соответствии с нашими ожиданиями:

```
$ pytest --setup-show tests/func/test_add.py
===== test session starts =====
platform win32 -- Python 3.6.5, pytest-3.9.3, py-1.7.0, pluggy-0.8.0
rootdir: c:\_BOOKS\_pytest_si\bopytest-code\code\ch3\b\tasks_proj\tests, inifile: pytest.ini
collected 3 items

tests\func\test_add.py
SETUP      S tmpdir_factory
SETUP      S tasks_db_session (fixtures used: tmpdir_factory)
SETUP      F tasks_db (fixtures used: tasks_db_session)
func/test_add.py::test_add_returns_valid_id (fixtures used: tasks_db, tasks_db_session, tmpdir_factory).
TEARDOWN F tasks_db
SETUP      F tasks_db (fixtures used: tasks_db_session)
func/test_add.py::test_added_task_has_id_set (fixtures used: tasks_db, tasks_db_session, tmpdir_factory).
TEARDOWN F tasks_db
SETUP      S tasks_just_a_few
SETUP      F tasks_db (fixtures used: tasks_db_session)
SETUP      F db_with_3_tasks (fixtures used: tasks_db, tasks_just_a_few)
func/test_add.py::test_add_increases_count (fixtures used: db_with_3_tasks, tasks_db, tasks_db_session, tasks_just_a_few, tmpdir_factory).
TEARDOWN F db_with_3_tasks
TEARDOWN F tasks_db
TEARDOWN S tasks_db_session
TEARDOWN S tmpdir_factory
TEARDOWN S tasks_just_a_few

===== 3 passed in 0.24 seconds =====
```

Ага. Выглядит правильно. `tasks_db_session` вызывается один раз за сеанс, а более быстрый `task_db` теперь просто очищает базу данных каждым тестом.

## Specifying Fixtures with usefixtures

До сих пор, если вы хотели, чтобы тест использовал фикстуру, то вы помещали её в список параметров. Кроме того, можно отметить тест и класс с помощью `@pytest.mark.usefixtures('fixture1', 'fixture2')`. *usefixtures* принимает строку, состоящую из списка фикстур, разделённых запятыми. Это не особо имеет смысл делать с тестовыми функциями — это просто дольше набирать текст. Но это хорошо работает для тестов классов:

#### ch3/test\_scope.py

```
@pytest.mark.usefixtures('class_scope')
class TestSomething():
    """Demo class scope fixtures."""

    def test_3(self):
        """Test using a class scope fixture."""

    def test_4(self):
        """Again, multiple tests are more fun."""
```

Использование *usefixtures* почти то же самое, что указание имени фикстуры в списке параметров метода теста. Единственное отличие состоит в том, что тест может использовать возвращаемое значение фикстуры, только если оно указано в списке параметров. Тест, использующий фикстуру из-за использования *usefixtures*, не может использовать возвращаемое значение фикстуры.

## Использование autouse для Fixtures That Always Get Used (которые используются непрерывно)

До сих пор в этой главе все фикстуры, используемые тестами, были обертками тестов (или использовали *usefixtures* для этого одного примера класса). Однако вы можете использовать *autouse=True*, чтобы фикстура работала постоянно. Это хорошо работает для кода, который вы хотите запустить в определенное время, но тесты на самом деле не зависят от состояния системы или данных из фикстуры. Вот довольно наглядный пример:

#### ch3/test\_autouse.py

```
"""Демонстрация autouse fixtures."""

import pytest
import time

@pytest.fixture(autouse=True, scope='session')
def footer_session_scope():
    """Сообщает время в конце session(сеанса)."""
    yield
    now = time.time()
    print('--')
    print('finished : {}'.format(time.strftime('%d %b %X', time.localtime(now))))
    print('-----')

@pytest.fixture(autouse=True)
def footer_function_scope():
    """Сообщает продолжительность теста после каждой функции."""
    start = time.time()
    yield
    stop = time.time()
    delta = stop - start
    print('\ntest duration : {:.3} seconds'.format(delta))

def test_1():
    """Имитирует длительный тестовый тест."""
    time.sleep(1)

def test_2():
    """Имитирует немного более длительный тест."""
    time.sleep(1.23)
```

Тут мы демонстрируем добавление время тестирования после каждого теста, а также дату и текущее время в конце сеанса. Вот как всё это выглядит:

```
$ cd /path/to/code/ch3
$ pytest -v -s test_autouse.py
===== test session starts =====
collected 2 items
test_autouse.py::test_1 PASSED
test duration : 1.0 seconds
test_autouse.py::test_2 PASSED
test duration : 1.24 seconds
--
finished : 25 Jul 16:18:27
-----
===== 2 passed in 2.25 seconds =====
```

Функция *autouse* хорошо сработала. Но это скорее исключение, чем правило. Используйте фикстуры как декораторы, если у вас нет действительно большой причины не делать этого.

Теперь, когда вы видели *autouse* в действии, возможно вас интересует, почему мы не использовали его для *tasks\_db* в этой главе. В проекте Tasks я чувствовал, что важно сохранить возможность проверить, что произойдет, если мы попытаемся использовать функцию API до инициализации БД. Это должно привести к соответствующему исключению. Но мы не сможем это проверить, если принудительно инициализировать каждый тест.

## Переименование Fixtures

Название фикстур, перечисленные в списке параметров тестов и других фикстур, использующих их, обычно совпадает с именем функции фикстуры. Однако, *pytest* позволяет вам переименовывать фикстуры с параметром `name` в `@pytest.fixture()`:

**ch3/test\_rename\_fixture.py**

```
"""Демонстрация fixture renaming."""

import pytest

@pytest.fixture(name='lue')
def ultimate_answer_to_life_the_universe_and_everything():
    """Возвращает окончательный ответ."""
    return 42

def test_everything(lue):
    """Использует более короткое имя."""
    assert lue == 42
```

Здесь *lue* теперь является именем fixture, а не *fixture\_with\_a\_name\_much\_longer\_than\_lue*. Это имя даже появляется, если мы запускаем помощью `--setup-show`:

```
$ pytest --setup-show test_rename_fixture.py
===== test session starts =====
collected 1 items

test_rename_fixture.py
  SETUP      F lue
    test_rename_fixture.py::test_everything (fixtures used: lue).
  TEARDOWN   F lue

===== 1 passed in 0.01 seconds =====
```

Если вам нужно выяснить, где определен *lue*, следует добавить параметр `pytest --fixtures` и дать ему имя файла для теста. В нем перечислены все фикстуры, доступные для теста, в том числе те, которые были переименованы:

```
$ pytest --fixtures test_rename_fixture.py
===== test session starts =====
...

----- fixtures defined from test_rename_fixture -----
lue
    Return ultimate answer.

===== no tests ran in 0.01 seconds =====
```

Большая часть вывода не показана — там много чего. К счастью, фикстуры, которые мы определили, находятся внизу, вместе с тем, где они определены. Мы можем использовать это, чтобы найти определение *lue*. Давайте используем это в проекте «Tasks»:

```
$ cd /path/to/code/ch3/b/tasks_proj
$ pytest --fixtures tests/func/test_add.py
===== test session starts =====
...
tmpdir_factory
    Return a TempdirFactory instance for the test session.
tmpdir
    Return a temporary directory path object
    which is unique to each test function invocation,
    created as a sub directory of the base temporary
    directory. The returned object is a `py.path.local`_
    path object.

----- fixtures defined from conftest -----

tasks_db
    An empty tasks db.
tasks_just_a_few
    All summaries and owners are unique.
tasks_mult_per_owner
    Several owners with several tasks each.
db_with_3_tasks
    Connected db with 3 tasks, all unique.
db_with_multi_per_owner
    Connected db with 9 tasks, 3 owners, all with 3 tasks.
tasks_db_session
    Connect to db before tests, disconnect after.

===== no tests ran in 0.01 seconds =====
```

Классно! Все фикстуры из нашего *conftest.py* есть. И в нижней части встроенного списка находится *tmpdir* и *tmpdir\_factory*, которые мы тогда использовали.

## Параметризация Фикстур

В [Parametrized Testing]Параметризованном тестировании, на стр. 42, мы параметризовали тесты. Мы также можем параметризовать фикстуры. Мы по-прежнему используем наш список задач, список идентификаторов задач и функцию эквивалентности, как и раньше:

```
ch3/b/tasks_proj/tests/func/test_add_variety2.py

"""Test the tasks.add() API function."""

import pytest
import tasks
from tasks import Task

tasks_to_try = (Task('sleep', done=True),
                Task('wake', 'brian'),
                Task('breathe', 'BRIAN', True),
                Task('exercise', 'BrlaN', False))

task_ids = ['Task({}, {}, {})'.format(t.summary, t.owner, t.done)]
```

```

for t in tasks_to_try]

def equivalent(t1, t2):
    """Check two tasks for equivalence."""
    return ((t1.summary == t2.summary) and
            (t1.owner == t2.owner) and
            (t1.done == t2.done))

```

Но теперь, вместо параметризации теста, мы параметризуем фикстуру под названием `a_task`:

**ch3/b/tasks\_proj/tests/func/test\_add\_variety2.py**

```

@pytest.fixture(params=tasks_to_try)
def a_task(request):
    """Без идентификаторов."""
    return request.param

def test_add_a(tasks_db, a_task):
    """Использование фикстуры a_task (без ids)."""
    task_id = tasks.add(a_task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, a_task)

```

Запрос, указанный в параметре `fixture`, является другой встроенной фикстурой, представляющей вызывающее состояние фикстуры. Вы узнаете больше в следующей главе. Он имеет поле `param`, которое заполняется одним элементом из списка, назначенного `params` в

```
@pytest.fixture(params=tasks_to_try).
```

Элемент `a_task` довольно прост — он просто возвращает `request.param` в качестве значения для теста, используя его. Поскольку наш список задач состоит из четырех задач, фикстура будет вызываться четыре раза, а затем тест будет вызываться четыре раза:

```

$ cd /path/to/code/ch3/b/tasks_proj/tests/func
$ pytest -v test_add_variety2.py::test_add_a
===== test session starts =====
collected 4 items
test_add_variety2.py::test_add_a[a_task0] PASSED
test_add_variety2.py::test_add_a[a_task1] PASSED
test_add_variety2.py::test_add_a[a_task2] PASSED
test_add_variety2.py::test_add_a[a_task3] PASSED
===== 4 passed in 0.03 seconds =====

```

Мы не предоставили идентификаторы, так `pytest` сам придумал имена, добавив номер вызова(число) к имени фикстуры. Однако мы можем использовать тот же список строк, который мы использовали при параметризации наших тестов:

**ch3/b/tasks\_proj/tests/func/test\_add\_variety2.py**

```

@pytest.fixture(params=tasks_to_try, ids=task_ids)
def b_task(request):
    """Использование списка идентификаторов."""
    return request.param

def test_add_b(tasks_db, b_task):
    """Использование фикстуры b_task, с идентификаторами."""
    task_id = tasks.add(b_task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, b_task)

```

Этот вариант дает нам идентификаторы получше:

```

$ pytest -v test_add_variety2.py::test_add_b
===== test session starts =====
collected 4 items

```

```
test_add_variety2.py::test_add_b[Task(sleep,None,True)] PASSED
test_add_variety2.py::test_add_b[Task(wake,brian,False)] PASSED
test_add_variety2.py::test_add_b[Task(breathe,BRIAN,True)] PASSED
test_add_variety2.py::test_add_b[Task(exercise,BrIaN,False)] PASSED
===== 4 passed in 0.04 seconds =====
```

Мы также можем установить параметр `ids` в функцию, которую мы пишем, которая предоставляет идентификаторы. Вот как это выглядит, и мы используем функцию для генерации идентификаторов:

**ch3/b/tasks\_proj/tests/func/test\_add\_variety2.py**

```
def id_func(fixture_value):
    """Функция для генерации идентификаторов."""
    t = fixture_value
    return 'Task({}, {}, {})'.format(t.summary, t.owner, t.done)

@pytest.fixture(params=tasks_to_try, ids=id_func)
def c_task(request):
    """Использование функции (id_func) для генерации идентификаторов."""
    return request.param

def test_add_c(tasks_db, c_task):
    """Использование фикстуры с сгенерированными идентификаторами."""
    task_id = tasks.add(c_task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, c_task)
```

Функция будет вызвана из значения каждого элемента из параметризации. Поскольку параметризация представляет собой список объектов `id_func()` будет вызываться с объектом *Task*, что позволяет нам использовать методы доступа *namedtuple* для доступа к одному объекту *Task* для генерации идентификатора одного объекта *Task* за раз. Это немного чище, чем генерировать полный список раньше времени, и выгляди одинаково:

```
$ pytest -v test_add_variety2.py::test_add_c
===== test session starts =====
collected 4 items
test_add_variety2.py::test_add_c[Task(sleep,None,True)] PASSED
test_add_variety2.py::test_add_c[Task(wake,brian,False)] PASSED
test_add_variety2.py::test_add_c[Task(breathe,BRIAN,True)] PASSED
test_add_variety2.py::test_add_c[Task(exercise,BrIaN,False)] PASSED
===== 4 passed in 0.04 seconds =====
```

С параметризованными функциями вы можете запускать эту функцию несколько раз. Но с параметризованными фикстурами каждая тестовая функция, использующая эту фикстуру, будет вызываться несколько раз. И в этом сила, брат!

## Параметризация Fixtures в Tasks Project

Теперь давайте посмотрим, как мы можем использовать параметризованные фикстуры в проекте *Tasks*. До сих пор мы использовали *TinyC* всех тестов. Но мы хотим, чтобы наши варианты оставались открытыми до конца проекта. Поэтому любой код, который мы пишем, и любые тесты, которые мы пишем, должны работать как с *TinyDB*, так и с *MongoDB*.

Решение (в коде), для которого используется база данных, изолируется от вызова `start_tasks_db()` в фикстуре `tasks_db_session`:

**ch3/b/tasks\_proj/tests/conftest.py**

```
"""Определяем некоторые фикстуры для использования в проекте."""

import pytest
import tasks
from tasks import Task

@pytest.fixture(scope='session')
```

```
def tasks_db_session(tmpdir_factory):
    """Подключение к БД перед тестами, отключение после."""
    temp_dir = tmpdir_factory.mktemp('temp')
    tasks.start_tasks_db(str(temp_dir), 'tiny')
    yield
    tasks.stop_tasks_db()

@pytest.fixture()
def tasks_db(tasks_db_session):
    """Пустая база данных tasks."""
    tasks.delete_all()
```

Параметр `db_type` в вызове `start_tasks_db()` не является магическим. Он просто завершает переключение на подсистему, которая отвечает за остальные взаимодействия с базой данных:

### tasks\_proj/src/tasks/api.py

```
def start_tasks_db(db_path, db_type): # type: (str, str) -> None
    """Подключения функций API к БД."""
    if not isinstance(db_path, string_types):
        raise TypeError('db_path must be a string')
    global _tasksdb
    if db_type == 'tiny':
        import tasks.tasksdb_tinydb
        _tasksdb = tasks.tasksdb_tinydb.start_tasks_db(db_path)
    elif db_type == 'mongo':
        import tasks.tasksdb_pymongo
        _tasksdb = tasks.tasksdb_pymongo.start_tasks_db(db_path)
    else:
        raise ValueError("db_type должен быть 'tiny' или 'mongo'")
```

Чтобы протестировать MongoDB, нам нужно запустить все тесты с `db_type` равным `mongo`. Небольшая хитрость:

### ch3/c/tasks\_proj/tests/conftest.py

```
import pytest
import tasks
from tasks import Task

# @pytest.fixture(scope='session', params=['tiny',])
@pytest.fixture(scope='session', params=['tiny', 'mongo'])
def tasks_db_session(tmpdir_factory, request):
    """Connect to db before tests, disconnect after."""
    temp_dir = tmpdir_factory.mktemp('temp')
    tasks.start_tasks_db(str(temp_dir), request.param)
    yield # this is where the testing happens
    tasks.stop_tasks_db()

@pytest.fixture()
def tasks_db(tasks_db_session):
    """An empty tasks db."""
    tasks.delete_all()
```

Здесь я добавил `params=['tiny', 'mongo']` в фикстуру-декоратор. Ещё добавил `request` в список параметров `temp_dir` и установил `db_type` в `request.param` вместо того, чтобы просто выбрать "tiny" или "mongo".

Если установить `--verbose` или флаг `-v` при запуске в `pytest` параметризованных тестов или параметризованных фикстур, `pytest` присваивает разным прогонам на основе значения параметризации. И поскольку значения уже являются строками, это отлично работает.



## Installing MongoDB

Чтобы отслеживать тестирование MongoDB, убедитесь, что установлены MongoDB и *pymongo*. Лично я тестировал с изданием сообщества MongoDB, найденным тут <https://www.mongodb.com/download-center>. *pymongo* устанавливается с *pip*—*pip install pymongo*. Однако использовать MongoDB не обязательно поддерживать всю остальную часть книги; он используется в этом примере и в примере отладчика в **Главе 7**.

Вот что мы пока имеем:

```
$ cd /path/to/code/ch3/c/tasks_proj
$ pip install pymongo
$ pytest -v --tb=no
===== test session starts =====
collected 92 items
test_add.py::test_add_returns_valid_id[tiny] PASSED
test_add.py::test_added_task_has_id_set[tiny] PASSED
test_add.py::test_add_increases_count[tiny] PASSED
test_add_variety.py::test_add_1[tiny] PASSED
test_add_variety.py::test_add_2[tiny-task0] PASSED
test_add_variety.py::test_add_2[tiny-task1] PASSED
...
test_add.py::test_add_returns_valid_id[mongo] FAILED
test_add.py::test_added_task_has_id_set[mongo] FAILED
test_add.py::test_add_increases_count[mongo] PASSED
test_add_variety.py::test_add_1[mongo] FAILED
test_add_variety.py::test_add_2[mongo-task0] FAILED
...
===== 42 failed, 50 passed in 4.94 seconds =====
```

Хм. Облом. Похоже, нам нужно будет изрядно отладиться, прежде чем мы позволим кому-либо использовать версию Mongo. Вы узнаете, как отладить это в pdb: Отладка тестовых сбоев, на стр. 125. До тех пор мы будем использовать версию TinyDB.

## Упражнения

1. Создать тестовый файл `test_fixtures.py`.
2. Напишите несколько fixtures—functions данных с помощью декоратора `@pytest.fixture()`, которые будут возвращать некоторые данные. Возможно, список или словарь, или кортеж.
3. Для каждой фикстуры напишите хотя бы одну тестовую функцию, которая её использует.
4. Напишите два теста, которые используют одну и ту же фикстуру.
5. Запустить `pytest --setup-show test_fixtures.py`. Все фикстуры работают перед каждым тестом?
6. Добавьте `scope= 'module'` в фикстуру из упражнения 4.
7. Повторно запустите `pytest --setup-show test_fixtures.py`. Что изменилось?
8. Для фикстуры из упражнения 6 измените `return <data>` на `yield <data>`.
9. Добавить операторы печати до и после `yield`.
9. Запустите `pytest -s -v test_fixtures.py`. Имеет ли результат смысл?

## Что дальше

Реализация `pytest fixture` достаточно гибкая, чтобы использовать фикстуры, такие как *building blocks*, для создания тестового *setup* и *teardown* также для смены различных фрагментов системы (например, замена Mongo для TinyDB). Поскольку фикстуры настолько гибкие, я использую значительной степени, чтобы как можно больше настроить мои тесты на фикстуры.

В этой главе вы рассмотрели фикстуры `pytest`, которые пишете сами, а также пару встроенных (builtin) фикстур `tmpdir` и `tmpdir_factory`. В следующей главе вы подробно рассмотрите встроенные (builtin) фикстуры.



Теги: [pytest](#)

↑ +6 ↓

📌 15

👁 449

💬 Комментировать

29,0

Карма

148,9

Рейтинг

90

Подписчики

Александр Драгункин

@AleksandrDr

Пользователь

Поделиться публикацией

ПОХОЖИЕ ПУБЛИКАЦИИ

4 сентября 2017 в 15:21

Тестируем асинхронный код с помощью PyTest (перевод)

↑ +11

👁 7,6k

📌 76

💬 2

29 октября 2015 в 12:15

PyTest

↑ +20

👁 113k

📌 270

💬 9

10 ноября 2014 в 16:47

Как в Яндексе используют PyTest и другие фреймворки для функционального тестирования

↑ +58

👁 70,2k

📌 354

💬 10

ЗАКАЗЫ		Фриланс
Разработчик ПО машинного зрения	6 откликов · 62 просмотра	300000
Разработка WHATSAPP бота	11 откликов · 65 просмотров	100000 за проект
Ищу напарника на react native для совместной работы	5 откликов · 76 просмотров	100000 за проект
Разработка модуля поискового механизма (Elasticsearch)	9 откликов · 49 просмотров	50000 за проект
Сделать небольшое веб приложение fullstack	26 откликов · 131 просмотр	30000 за проект
Все заказы		Разместить заказ

Комментарии 0

Только полноправные пользователи могут оставлять комментарии. [Войдите, пожалуйста.](#)

САМОЕ ЧИТАЕМОЕ

- Сутки
- Неделя
- Месяц

Бунт на Пикабу. Пользователи массово уходят на Реддит

+144 91,5k 71 290

Смерть курьера «Яндекс.Еды» запустила волну жалоб на условия труда в компании

+57 57,1k 12 352

Как я хакера ловил

+106 17,5k 81 49

Как Мегафон спалился на мобильных подписках

+500 89,2k 174 462

Межпозвоночная грыжа? Работай над ней

+37 20,6k 163 51

Аккаунт	Разделы	Информация	Услуги	Приложения
Войти	Публикации	Правила	Реклама	<div> Загрузите в App Store</div> <div> доступно Google</div>
Регистрация	Новости	Помощь	Тарифы	
	Хабы	Документация	Контент	
	Компании	Соглашение	Семинары	
	Пользователи	Конфиденциальность		
	Песочница			