

 AlekSandrDr вчера в 17:12

Python Testing с pytest. Builtin Fixtures, Глава 4

Автор оригинала: Okken Brian

Python

Перевод

Tutorial

 Вернуться  Дальше

Встроенные фикстуры, которые поставляются с `pytest`, могут помочь вам сделать довольно полезные вещи в ваших тестах легко и непринужденно. Например, помимо обработки временных файлов, `pytest` включает встроенные фикстуры для доступа к параметрам командной строки, связи между сеансами тестирования, проверки выходных потоков, изменения переменных среды и опроса предупреждений.

The
Pragmatic
Programmers

Python Testing with pytest

Simple, Rapid,
Effective, and
ScalableBrian Okken
edited by Katharine Dvorak

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, чтобы проверить свой собственный проект (у вас развязаны!), вы должны перейти на веб-страницу книги, чтобы скачать работу. Там же, на веб-странице книги есть ссылка на сообщение `errata` и дискуссионный форум.

Под спойлером приведен список статей этой серии.

[Оглавление](#)

В предыдущей главе вы рассмотрели, что такое фикстуры, как их писать и как их использовать для тестовых данных, а также для `setup` и `teardown` кода.

Вы также использовали `conftest.py` для совместного использования фикстур между тестами в нескольких тестовых файлах. В конце главы `pytest` Fixtures на странице 49 проекта «Tasks» были установлены следующие фикстуры: `tasks_db_session`, `tasks_just_a_few`, `tasks_mult_per_owner`, `tasks_db`, `db_with_3_tasks` и `db_with_multi_per_owner`, определенные в `conftest.py`, которые могут использоваться тестовой функцией в проекте «Задачи», которая в них нуждается.

Повторное использование обычных фикстур настолько хорошая идея, что разработчики `pytest` включили некоторые часто требующиеся фикстуры в `pytest`. Вы уже видели, как `tmpdir` и `tmpdir_factory` используются проектом `Tasks` в разделе смены области для фикстур проекта `Tasks` на стр. 59. Вы разберете их более подробно в этой главе.

Встроенные фикстуры, которые поставляются с `pytest`, могут помочь вам сделать довольно полезные вещи в ваших тестах легко и непринужденно. Например, помимо обработки временных файлов, `pytest` включает встроенные фикстуры для доступа к параметрам команды, связи между сеансами тестирования, проверки выходных потоков, изменения переменных среды и опроса предупреждений. Встроенные фикстуры — это расширения функциональности ядра `pytest`. Давайте теперь посмотрим на несколько из наиболее часто используемых встроенных фикстур по порядку.

Использование `tmpdir` и `tmpdir_factory`

Если вы тестируете что-то, что считывает, записывает или изменяет файлы, вы можете использовать `tmpdir` для создания файлов или каталогов, используемых одним тестом, и вы можете использовать `tmpdir_factory`, когда хотите настроить каталог для нескольких тестов.

Фикстура `tmpdir` имеет область действия функции (`function scope`), и фикстура `tmpdir_factory` имеет область действия сеанса (`session scope`). Любой отдельный тест, которому требуется временный каталог или файл только для одного теста, может использовать `tmpdir`. Это также верно для фикстуры, которая настраивает каталог или файл, которые должны быть воссозданы для каждой тестовой функции.

Вот простой пример использования `tmpdir`:

ch4/test_tmpdir.py

```
def test_tmpdir(tmpdir):
    # tmpdir уже имеет имя пути, связанное с ним
    # join() расширяет путь, чтобы включить имя файла,
    # создаваемого при записи в
    a_file = tmpdir.join('something.txt')

    # можете создавать каталоги
    a_sub_dir = tmpdir.mkdir('anything')

    # можете создавать файлы в директориях (создаются при записи)
    another_file = a_sub_dir.join('something_else.txt')

    # эта запись создает 'something.txt'
    a_file.write('contents may settle during shipping')

    # эта запись создает 'anything/something_else.txt'
    another_file.write('something different')

    # вы также можете прочитать файлы
    assert a_file.read() == 'contents may settle during shipping'
    assert another_file.read() == 'something different'
```

Значение, возвращаемое из `tmpdir`, является объектом типа `py.path.local`. Это кажется все, что нам нужно для временных каталогов и файлов. Тем не менее, есть одна хитрость. Поскольку фикстура `tmpdir` определена как область действия функции (`function scope`), `tmpdir` используется для создания папок или файлов, которые должны быть доступны дольше, чем одна тестовая функция. Для фикстур с областью видимости, отличной от функции (класс, модуль, сеанс), доступен `tmpdir_factory`.

Фикстура `tmpdir_factory` очень похожа на `tmpdir`, но имеет другой интерфейс. Как описано в разделе «Спецификация областей (Scope)» на стр. 56, фикстуры функциональной области запускаются один раз для каждой тестовой функции, фикстуры модульной области запускаются один раз на модуль, фикстуры класса один раз для каждого класса, и тесты проверки области работают один раз за сеанс. Таким образом, ресурсы, созданные в записях области сеанса, имеют срок службы всего сеанса. Чтобы показать, насколько похожи `tmpdir` и `tmpdir_factory`, изменю пример `tmpdir`, где достаточно использовать `tmpdir_factory`:

ch4/test_tmpdir.py

```
def test_tmpdir_factory(tmpdir_factory):
    # вы должны начать с создания каталога. a_dir действует как
```

```
# объект, возвращенный из фикстуры tmpdir
a_dir = tmpdir_factory.mktemp('mydir')

# base_temp будет родительским каталогом 'mydir' вам не нужно
# использовать getbasetemp(), чтобы
# показать, что он доступен
base_temp = tmpdir_factory.getbasetemp()
print('base:', base_temp)

# остальная часть этого теста выглядит так же,
# как в Примере ' test_tmpdir ()', за исключением того,
# что я использую a_dir вместо tmpdir

a_file = a_dir.join('something.txt')
a_sub_dir = a_dir.mkdir('anything')
another_file = a_sub_dir.join('something_else.txt')

a_file.write('contents may settle during shipping')
another_file.write('something different')

assert a_file.read() == 'contents may settle during shipping'
assert another_file.read() == 'something different'
```

Первая строка использует `mktemp('mydir')` для создания каталога и сохраняет его в `a_dir`. Для остальной части функции можно использовать `a_dir` так же, как `tmpdir`, возвращенный из фикстуры `tmpdir`.

Во второй строке примера `tmpdir_factory` функция `getbasetemp()` возвращает базовый каталог, используемый для данного сеанса. Оператор `print` в примере нужен, чтобы можно было посмотреть каталог в вашей системе. Давайте посмотрим, где он находится:

```
$ cd /path/to/code/ch4
$ pytest -q -s test_tmpdir.py::test_tmpdir_factory
base: /private/var/folders/53/zv4j_zc506x2xq25l31qxvxn0000gn/T/pytest-of-okken/pytest-732
.
1 passed in 0.04 seconds
```

Этот базовый каталог зависит от системы и пользователя, и `pytest - NUM` изменяется для каждого сеанса с увеличением `NUM`. Базовый каталог остается один после сеанса. `pytest` очищает его, и в системе остаются только самые последние несколько временных базовых каталогов, что отлично, если вам приспичит проверить файлы после тестового запуска.

Вы также можете указать свой собственный базовый каталог, если вам нужно с помощью `pytest --basetemp=mydir`.

Использование временных каталогов для других областей

Мы получаем временные каталоги и файлы области **сеанса** из фикстуры `tmpdir_factory`, а каталоги и файлы области **функции** из фикстуры `tmpdir`. Но как насчет других областей? Что делать, если нам нужен временный каталог области видимости модуля или класса? Чтобы сделать это, мы создаем другую фикстуру области нужного размерчика и для этого следует использовать `tmpdir_factory`.

Например, предположим, что у нас есть модуль, полный тестов, и многие из них должны иметь возможность читать некоторые данные из файла `json`. Мы смогли положить фикстуру объема модуля в сам модуль, или в `conftest.py` файл, который настраивает файл данных следующим образом:

ch4/authors/conftest.py

```
"""Demonstrate tmpdir_factory."""

import json
import pytest

@pytest.fixture(scope='module')
def author_file_json(tmpdir_factory):
    """Пишем некоторых авторов в файл данных."""
    python_author_data = {
        'Ned': {'City': 'Boston'},
        'Brian': {'City': 'Portland'},
```

```

    'Luciano': {'City': 'Sau Paulo'}
}

file = tmpdir_factory.mktemp('data').join('author_file.json')
print('file:{}'.format(str(file)))

with file.open('w') as f:
    json.dump(python_author_data, f)
return file

```

Фикстура `author_file_json()` создает временный каталог с именем `data` и создает файл с именем `author_file.json` в каталоге данных. Затем записывает словарь `python_author_data` как `json`. Поскольку это фикстура области модуля, `json`-файл будет создан только один раз для каждого модуля, использующего тест:

ch4/authors/test_authors.py

```

"""Некоторые тесты, использующие временные файлы данных."""
import json

def test_brian_in_portland(author_file_json):
    """Тест, использующий файл данных."""
    with author_file_json.open() as f:
        authors = json.load(f)
        assert authors['Brian']['City'] == 'Portland'

def test_all_have_cities(author_file_json):
    """Для обоих тестов используется один и тот же файл."""
    with author_file_json.open() as f:
        authors = json.load(f)
        for a in authors:
            assert len(authors[a]['City']) > 0

```

Оба теста будут использовать один и тот же JSON-файл. Если один файл тестовых данных работает для нескольких тестов, нет смысла создавать его заново для обоих тестов.

Использование pytestconfig

С помощью встроенной фикстуры `pytestconfig` вы можете управлять тем, как `pytest` работает с аргументами и параметрами командной строки, файлами конфигурации, плагинами и каталогом, из которого вы запустили `pytest`. Фикстура `pytestconfig` является ярлыком для `request.config` и иногда упоминается в документации `pytest` как *"the pytest config object"* (объект конфигурации `pytest`).

Чтобы узнать, как работает `pytestconfig`, вы можете посмотреть, как добавить пользовательский параметр командной строки и прочитать значение параметра из теста. Прочитать значение параметров командной строки вы сможете непосредственно из `pytestconfig`, но чтобы добавить параметр и проанализировать его, вам нужно добавить функцию-ловушку (`hook`). Функции *hook*, которые я более подробно описываю в Главе 5, "Планы на стр. 95, являются еще одним способом управления поведением `pytest` и часто используются в плагинах. Однако добавление пользовательской опции командной строки и чтение ее из `pytestconfig` достаточно широко распространено, поэтому я хочу осветить это здесь.

Мы будем использовать *pytest hook* `pytest_addoption`, чтобы добавить несколько параметров к параметрам, уже доступным в командной строке `pytest`:

ch4/pytestconfig/conftest.py

```

def pytest_addoption(parser):
    parser.addoption("--myopt", action="store_true",
                    help="some boolean option")
    parser.addoption("--foo", action="store", default="bar",
                    help="foo: bar or baz")

```

Добавление параметров командной строки через `pytest_addoption` должно выполняться через плагины или в файле **conftest.py** расположенном в верхней части структуры каталога проекта. Вы не должны делать это в тестовом подкаталоге.

Параметры `--myopt` и `--foo <value>` были добавлены в предыдущий код, а строка справки была изменена, как показано ниже:

```
$ cd /path/to/code/ch4/pytestconfig
$ pytest --help
usage: pytest [options] [file_or_dir] [file_or_dir] [...]
...
custom options:
  --myopt                some boolean option
  --foo=FOO              foo: bar or baz
...
```

Теперь мы можем получить доступ к этим опциям из теста:

ch4/pytestconfig/test_config.py

```
import pytest

def test_option(pytestconfig):
    print('"foo" set to:', pytestconfig.getoption('foo'))
    print('"myopt" set to:', pytestconfig.getoption('myopt'))
```

Давайте посмотрим, как это работает:

```
$ pytest -s -q test_config.py::test_option
"foo" set to: bar
"myopt" set to: False
.1
passed in 0.01 seconds
$ pytest -s -q --myopt test_config.py::test_option
"foo" set to: bar
"myopt" set to: True
.1
passed in 0.01 seconds
$ pytest -s -q --myopt --foo baz test_config.py::test_option
"foo" set to: baz
"myopt" set to: True
.1
passed in 0.01 seconds
```

Поскольку `pytestconfig` является фикстурой, его также можно получить из других фикстур. Вы можете сделать фикстуры для имен опций, если хотите, например:

ch4/pytestconfig/test_config.py

```
@pytest.fixture()
def foo(pytestconfig):
    return pytestconfig.option.foo

@pytest.fixture()
def myopt(pytestconfig):
    return pytestconfig.option.myopt

def test_fixtures_for_options(foo, myopt):
    print('"foo" set to:', foo)
    print('"myopt" set to:', myopt)
```

Вы также можете получить доступ к встроенным параметрам, а не только к добавляемым, а также к информации о том, как был запущен р (каталог, аргументы и т.д.).

Вот пример нескольких значений и параметров конфигурации:

```
def test_pytestconfig(pytestconfig):
    print('args          : ', pytestconfig.args)
    print('inifile       : ', pytestconfig.inifile)
    print('invocation_dir   : ', pytestconfig.invocation_dir)
    print('rootdir          : ', pytestconfig.rootdir)
    print('-k EXPRESSION    : ', pytestconfig.getoption('keyword'))
    print('-v, --verbose    : ', pytestconfig.getoption('verbose'))
    print('-q, --quiet       : ', pytestconfig.getoption('quiet'))
    print('-l, --showlocals: ', pytestconfig.getoption('showlocals'))
    print('--tb=style      : ', pytestconfig.getoption('tbstyle'))
```

Мы вернемся к `pytestconfig`, когда я продемонстрирую `ini`-файлы в главе 6 "Конфигурация" на стр. 113.

Using cache

Обычно мы, тестировщики, думаем, что каждый тест максимально независим от других тестов. Следует убедиться, что не закрались зависимость от порядка. Хотелось бы иметь возможность запустить или перезапустить любой тест в любом порядке и получить тот же результат. Кроме того, надо, чтобы сеансы тестирования были повторяемыми и не изменяли поведение на основе предыдущих сеансов тестирования.

Однако иногда передача информации с одного тестового сеанса в другой может быть весьма полезной. Когда мы хотим передать информацию будущим тестовым сессиям, мы можем сделать это с помощью встроенной фикстуры `cache`.

Фикстура `cache` предназначена для хранения информации об одном тестовом сеансе и получения её в следующем. Отличным примером использования полномочий `cache` для пользы дела является встроенная функциональность `--last-failed` и `--failed-first`. Давайте посмотрим, как данные для этих флагов хранятся в кэше.

Вот текст справки для опций `--last-failed` и `--failed-first`, а также несколько параметров `cache`:

```
$ pytest --help
...
--lf, --last-failed rerun only the tests that failed at the last run (or
                    all if none failed)
--ff, --failed-first run all tests but run the last failures first. This
                    may re-order tests and thus lead to repeated fixture
                    setup/teardown
--cache-show show cache contents, don't perform collection or tests
--cache-clear remove all cache contents at start of test run.
...
```

Чтобы увидеть их в действии, будем использовать эти два теста:

ch4/cache/test_pass_fail.py

```
def test_this_passes():
    assert 1 == 1

def test_this_fails():
    assert 1 == 2
```

Давайте запустим их, используя `--verbose`, чтобы увидеть имена функций, и `--tb=no`, чтобы скрыть трассировку стека:

```
$ cd /path/to/code/ch4/cache
$ pytest --verbose --tb=no test_pass_fail.py
===== test session starts =====
collected 2 items
test_pass_fail.py::test_this_passes PASSED
```

```
test_pass_fail.py::test_this_fails FAILED
===== 1 failed, 1 passed in 0.05 seconds =====
```

Если вы запустите их снова с флагом `--ff` или `--failed-first`, то тесты, которые завершились неудачей ранее, будут выполнены первыми затем и весь сеанс:

```
$ pytest --verbose --tb=no --ff test_pass_fail.py
===== test session starts =====
run-last-failure: rerun last 1 failures first
collected 2 items
test_pass_fail.py::test_this_fails FAILED
test_pass_fail.py::test_this_passes PASSED
===== 1 failed, 1 passed in 0.04 seconds =====
```

Или вы можете использовать `--lf` или `--last-failed`, чтобы просто запустить тесты, которые провалились в прошлый раз:

```
$ pytest --verbose --tb=no --lf test_pass_fail.py
===== test session starts =====
run-last-failure: rerun last 1 failures
collected 2 items
test_pass_fail.py::test_this_fails FAILED
===== 1 tests deselected =====
===== 1 failed, 1 deselected in 0.05 seconds =====
```

Прежде чем мы рассмотрим, как сохраняются данные о сбоях и как вы можете использовать один и тот же механизм, давайте рассмотрим пример, который делает значение `--lf` и `--ff` еще более очевидным.

Вот параметризованный тест с одним сбоем:

ch4/cache/test_few_failures.py

```
"""Demonstrate -lf and -ff with failing tests."""

import pytest
from pytest import approx

testdata = [
    # x, y, expected
    (1.01, 2.01, 3.02),
    (1e25, 1e23, 1.1e25),
    (1.23, 3.21, 4.44),
    (0.1, 0.2, 0.3),
    (1e25, 1e24, 1.1e25)
]

@pytest.mark.parametrize("x,y,expected", testdata)
def test_a(x, y, expected):
    """Demo approx()."""
    sum_ = x + y
    assert sum_ == approx(expected)
```

И на выходе:

```
$ cd /path/to/code/ch4/cache
$ pytest -q test_few_failures.py
.F...
===== FAILURES =====
_____ test_a[1e+25-1e+23-1.1e+25] _____

x = 1e+25, y = 1e+23, expected = 1.1e+25
```

```
@pytest.mark.parametrize("x,y,expected", testdata)
def test_a(x, y, expected):
    """Demo approx()."""
    sum_ = x + y
> assert sum_ == approx(expected)
E assert 1.01e+25 == 1.1e+25 ± 1.1e+19
E + where 1.1e+25 ± 1.1e+19 = approx(1.1e+25)

test_few_failures.py:17: AssertionError
1 failed, 4 passed in 0.06 seconds
```

Может быть, вы можете определить проблему сразу. Но давайте представим, что тест длиннее и сложнее, и не так уж очевидно, что тут не. Давайте снова запустим тест, чтобы снова увидеть ошибку. Тестовый случай можно указать в командной строке:

```
$ pytest -q "test_few_failures.py::test_a[1e+25-1e+23-1.1e+25]"
```

Если вы не хотите копипастить(*copy/paste*) или приключилось несколько неудачных случаев, которые вы хотели бы перезапустить, то `--lf` намного проще. И если вы действительно отлаживаете сбой теста, еще один флаг, который может облегчить ситуацию, `--showlocals`, или краткости:

```
$ pytest -q --lf -l test_few_failures.py
F
===== FAILURES =====
_____ test_a[1e+25-1e+23-1.1e+25] _____

x = 1e+25, y = 1e+23, expected = 1.1e+25

@pytest.mark.parametrize("x,y,expected", testdata)
def test_a(x, y, expected):
    """Demo approx()."""
    sum_ = x + y
> assert sum_ == approx(expected)
E assert 1.01e+25 == 1.1e+25 ± 1.1e+19
E + where 1.1e+25 ± 1.1e+19 = approx(1.1e+25)

expected = 1.1e+25
sum_ = 1.01e+25
x = 1e+25
y = 1e+23
test_few_failures.py:17: AssertionError
===== 4 tests deselected =====
1 failed, 4 deselected in 0.05 seconds
```

Причина неудачи должна быть более очевидной.

Для того, чтобы оставить в памяти, что тест не смог в прошлый раз, есть маленькая хитрость. `pytest` хранит информацию об ошибке теста последнего тестового сеанса и вы можете просмотреть сохраненную информацию с помощью `--cache-show`:

```
$ pytest --cache-show
===== test session starts =====
----- cache values -----
cache/lastfailed contains:
{'test_few_failures.py::test_a[1e+25-1e+23-1.1e+25]': True}
===== no tests ran in 0.00 seconds =====
```

Или вы можете посмотреть в директории кэша:

```
$ cat .cache/v/cache/lastfailed
{
  "test_few_failures.py::test_a[1e+25-1e+23-1.1e+25]": true
}
```

Ключ `--clear-cache` позволяет очистить кэш перед сеансом.

Кэш можно использовать не только для `--lf` и `--ff`. Давайте напишем фикстуру, которая записывает, сколько времени занимают тесты, эка время, а при следующем запуске сообщает об ошибке в тестах, которые занимают в два раза дольше больше времени, чем, скажем, в про раз.

Интерфейс для кеш-фикстуры простой.

```
cache.get(key, default)
cache.set(key, value)
```

По соглашению, имена ключей начинаются с имени вашего приложения или плагина, за которым следует `/`, и продолжают разделять раздел имени ключа с `/`. Значение, которое вы храните, может быть любым, которое конвертируется в *json*, так как представлено в `.cache direct`

Вот наша фикстура, используемая для фиксации времени тестов:

ch4/cache/test_slower.py

```
@pytest.fixture(autouse=True)
def check_duration(request, cache):
    key = 'duration/' + request.node.nodeid.replace(':', '_')
    # идентификатор узла (nodeid) может иметь двоеточия
    # ключи становятся именами файлов внутри .cache
    # меняем двоеточия на что-то безопасное в имени файла
    start_time = datetime.datetime.now()
    yield
    stop_time = datetime.datetime.now()
    this_duration = (stop_time - start_time).total_seconds()
    last_duration = cache.get(key, None)
    cache.set(key, this_duration)
    if last_duration is not None:
        errorstring = "длительность теста превышает последний более чем в 2-а раза "
        assert this_duration <= last_duration * 2, errorstring
```

Поскольку фикстура является *autouse*, на неё не нужно ссылаться из теста. Объект *request* используется для получения *nodeid* что бы использовать в ключе. *nodeid* — уникальный идентификатор, который работает даже с параметризованными тестами. Мы добавляем ключ 'duration', чтобы быть добропорядочными жителями кэша. Код выше *yield* выполняется до тестовой функции; код после *yield* выполняется после тестовой функции.

Теперь нам нужны некоторые тесты, которые занимают разные промежутки времени:

ch4/cache/test_slower.py

```
@pytest.mark.parametrize('i', range(5))
def test_slow_stuff(i):
    time.sleep(random.random())
```

Поскольку вы, вероятно, не хотите писать кучу тестов для этого, я использовал *random* и параметризацию, чтобы легко сгенерить некоторые тесты, которые поспят в течение случайного количества времени, все короче секунды. Давайте посмотрим пару раз, как это работает:

```
$ cd /path/to/code/ch4/cache
$ pytest -q --cache-clear test_slower.py
.....
5 passed in 2.10 seconds

$ pytest -q --tb=line test_slower.py
...E..E
===== ERRORS =====
_____ ERROR at teardown of test_slow_stuff[1] _____
E   AssertionError: test duration over 2x last duration
    assert 0.35702 <= (0.148009 * 2)
_____ ERROR at teardown of test_slow_stuff[4] _____
E   AssertionError: test duration over 2x last duration
    assert 0.888051 <= (0.324019 * 2)
5 passed, 2 error in 3.17 seconds
```

Что ж, это было весело. Давайте посмотрим, что в кэше:

```
$ pytest -q --cache-show

----- cache values -----
cache\lastfailed contains:
{'test_slower.py::test_slow_stuff[2]': True,
 'test_slower.py::test_slow_stuff[4]': True}
cache\nodeids contains:
['test_slower.py::test_slow_stuff[0]',
 'test_slower.py::test_slow_stuff[1]',
 'test_slower.py::test_slow_stuff[2]',
 'test_slower.py::test_slow_stuff[3]',
 'test_slower.py::test_slow_stuff[4]']
cache\stepwise contains:
[]
duration\test_slower.py__test_slow_stuff[0] contains:
0.958055
duration\test_slower.py__test_slow_stuff[1] contains:
0.214012
duration\test_slower.py__test_slow_stuff[2] contains:
0.19001
duration\test_slower.py__test_slow_stuff[3] contains:
0.725041
duration\test_slower.py__test_slow_stuff[4] contains:
0.836048

no tests ran in 0.03 seconds
```

Вы можете легко увидеть данные `duration` (продолжительности) отдельно от данных кэша из-за префикса имен данных кэша. Тем не менее интересно, что функциональность `lastfailed` может работать с одной записью кэша. Наши данные о продолжительности занимают одну запись кэша для каждого теста. Давайте последуем примеру `lastfailed` и поместим наши данные в одну запись.

Мы читаем и записываем в кэш для каждого теста. Мы можем разделить фикстуру на фикстуру области видимости функции для измерения длительности и фикстуру области видимости сессии для чтения и записи в кэш. Однако, если мы сделаем это, мы не сможем использовать фикстуру кэша, потому что она имеет область видимости функции. К счастью, быстрый взгляд на реализацию на [GitHub](#) показывает, что фикширование просто возвращает `request.config.cache`. Это доступно в любой области.

Вот одна из возможных реорганизаций одной и той же функциональности:

`ch4/cache/test_slower_2.py`

```
Duration = namedtuple('Duration', ['current', 'last'])

@pytest.fixture(scope='session')
def duration_cache(request):
    key = 'duration/testdurations'
    d = Duration({}, request.config.cache.get(key, {}))
    yield d
    request.config.cache.set(key, d.current)

@pytest.fixture(autouse=True)
def check_duration(request, duration_cache):
    d = duration_cache
    nodeid = request.node.nodeid
    start_time = datetime.datetime.now()
    yield
    duration = (datetime.datetime.now() - start_time).total_seconds()
    d.current[nodeid] = duration
    if d.last.get(nodeid, None) is not None:
        errorstring = "test duration over 2x last duration"
        assert duration <= (d.last[nodeid] * 2), errorstring
```

Фикстура `duration_cache` принадлежит области сеанса. Она читает предыдущую запись или пустой словарь, если нет предыдущих кэшированных данных, прежде чем запускать какие-либо тесты. В предыдущем коде мы сохранили как извлеченный словарь, так и пустой `namedtuple` именованном `Duration` с методами доступа `current` и `last`. Затем мы передали этот `namedtuple` в `test_duration`, который является

функцией и запускается для каждой тестовой функции. По мере выполнения теста, то же `namedtuple` передается в каждый тест, и время дт текущего теста хранятся в словарь `d.current`. По окончании тестового сеанса собранный текущий словарь сохраняется в кеше.

После запуска его пару раз, давайте посмотрим на сохраненный кэш:

```
$ pytest -q --cache-clear test_slower_2.py
.....
5 passed in 2.80 seconds

$ pytest -q --tb=no test_slower_2.py
...E.E...
7 passed, 2 error in 3.21 seconds

$ pytest -q --cache-show
----- cache values -----
cache\lastfailed contains:
{'test_slower_2.py::test_slow_stuff[2]': True,
 'test_slower_2.py::test_slow_stuff[3]': True}
duration\testdurations contains:
{'test_slower_2.py::test_slow_stuff[0]': 0.483028,
 'test_slower_2.py::test_slow_stuff[1]': 0.198011,
 'test_slower_2.py::test_slow_stuff[2]': 0.426024,
 'test_slower_2.py::test_slow_stuff[3]': 0.762044,
 'test_slower_2.py::test_slow_stuff[4]': 0.056003,
 'test_slower_2.py::test_slow_stuff[5]': 0.18401,
 'test_slower_2.py::test_slow_stuff[6]': 0.943054}

no tests ran in 0.02 seconds
```

Выглядит лучше.

Использование capsys

Фкстура `capsys builtin` обеспечивает два бита функциональности: позволяет получить `stdout` и `stderr` из некоторого кода, и временно откл захват вывода. Давайте посмотрим на получение `stdout` и `stderr`.

Предположим, у вас есть функция для печати приветствия для `stdout`:

ch4/cap/test_capsys.py

```
def greeting(name):
    print('Hi, {}'.format(name))
```

Вы не можете проверить это, проверив возвращаемое значение. Вы должны как-то проверить `stdout`. Вы можете проверить результат с пом `capsys`:

ch4/cap/test_capsys.py

```
def test_greeting(capsys):
    greeting('Earthling')
    out, err = capsys.readouterr()
    assert out == 'Hi, Earthling\n'
    assert err == ''

    greeting('Brian')
    greeting('Nerd')
    out, err = capsys.readouterr()
    assert out == 'Hi, Brian\nHi, Nerd\n'
    assert err == ''
```

Захваченные `stdout` и `stderr` извлекаются из `capsys.readouterr()`. Возвращаемое значение — это то, что было зафиксировано с начала ф или с момента последнего вызова.

В предыдущем примере используется только `stdout`. Давайте посмотрим на пример, используя поток `stderr`:

```
def yikes(problem):
    print('YIKES! {}'.format(problem), file=sys.stderr)

def test_yikes(capsys):
    yikes('Out of coffee!')
    out, err = capsys.readouterr()
    assert out == ''
    assert 'Out of coffee!' in err
```

`pytest` обычно захватывает выходные данные тестов и тестируемого кода. В том числе инструкции `print`. Захваченный вывод отображается, отказов тестов только после завершения полного тестового сеанса. Параметр `-s` отключает эту функцию, и выходные данные отправляются `stdout` во время выполнения тестов. Обычно это отлично работает, так как это выходные данные из неудачных тестов, которые необходимо увидеть для отладки сбоев. Тем не менее, вы можете позволить каким то выходным данным сделать его через захват вывода `pytest` по умолчанию, чтобы напечатать отдельные вещи, не печатая все. Вы можете сделать это с `capsys`. Вы можете использовать `capsys.disable` чтобы временно пропустить вывод через механизм захвата.

Вот пример:

ch4/cap/test_capsys.py

```
def test_capsys_disabled(capsys):
    with capsys.disabled():
        print('\nalways print this') # всегда печатать это
        print('normal print, usually captured') # обычная печать, обычно захваченная
```

Теперь, `'always print this'` всегда будет выводиться:

```
$ cd /path/to/code/ch4/cap
$ pytest -q test_capsys.py::test_capsys_disabled
```

Как вы можете видеть, сообщение `always print this` выводится всегда с захватом вывода или без него, так как оно печатается внутри блс `capsys.disabled()`. Другой оператор `print` — это просто обычный оператор `print`, поэтому `normal print, usually captured` (обычная печать, обычно захваченная), видна только в выводе, когда мы передаем флаг `-s`, который является ярлыком для `--capture=no`, отключая захват в

Использование monkeypatch

"monkey patch" — это динамическая модификация класса или модуля во время выполнения. Во время тестирования "monkey patching" — э удобный способ взять на себя часть среды выполнения тестируемого кода и заменить либо входные зависимости, либо выходные зависис объектами или функциями, которые более удобны для тестирования. Встроенная фикстура `monkeypatch` позволяет сделать это в контексте одного теста. И когда тест заканчивается, независимо от того, пройден он или нет, оригинал восстанавливается, отменяя все изменения па Все это очень запутано, пока мы не перейдем к некоторым примерам. После изучения API мы рассмотрим, как `monkeypatch` используется в тестовом коде.

Фикстура `monkeypatch` обеспечивает следующие функции:

- `setattr(target, name, value=<notset>, raising=True)`: Установить атрибут.
- `delattr(target, name=<notset>, raising=True)`: Удалить атрибут.
- `setitem(dic, name, value)`: Задать запись в словаре.
- `delitem(dic, name, raising=True)`: Удалить запись в словаре.
- `setenv(name, value, prepend=None)`: Задать переменную окружения .

- `delenv(name, raising=True)`: Удалите переменную окружения.
- `syspath_prepend(path)`: Начало пути в `sys.путь`, который является списком папок для импорта Python.
- `chdir(path)`: Изменить текущий рабочий каталог.

Параметр `raising` указывает `pytest`, следует ли создавать исключение, если элемент еще не существует. Параметр `prepend` для `setenv()` может быть символом. Если он установлен, значение переменной среды будет изменено на значение `+ prepend + <old value>`.

Чтобы увидеть `monkeypatch` в действии, давайте посмотрим на код, который пишет dot-файл конфигурации. Поведение некоторых программ может быть изменено с помощью настроек и значений, заданных в dot-файле в домашнем каталоге пользователя. Вот несколько строк кода который читает и записывает cheese-файл персональных настроек:

ch4/monkey/cheese.py

```
import os
import json

def read_cheese_preferences():
    full_path = os.path.expanduser('~/.cheese.json')
    with open(full_path, 'r') as f:
        prefs = json.load(f)
    return prefs

def write_cheese_preferences(prefs):
    full_path = os.path.expanduser('~/.cheese.json')
    with open(full_path, 'w') as f:
        json.dump(prefs, f, indent=4)

def write_default_cheese_preferences():
    write_cheese_preferences(_default_prefs)

_default_prefs = {
    'slicing': ['manchego', 'sharp cheddar'],
    'spreadable': ['Saint Andre', 'camembert',
                  'bucheron', 'goat', 'humbolt fog', 'cambozola'],
    'salads': ['crumbled feta']
}
```

Давайте посмотрим, как мы могли бы проверить `write_default_cheese_preferences()`. Это функция, которая не принимает никаких параметров и ничего не возвращает. Но имеет побочный эффект, который мы можем проверить. Она записывает файл в домашний каталог текущего пользователя.

Один из подходов заключается в том, чтобы просто позволить ему работать нормально и проверить побочный эффект. Предположим, у нас есть тесты для `read_cheese_preferences()`, и я доверяю им, поэтому я могу использовать их при тестировании

`write_default_cheese_preferences()`:

ch4/monkey/test_cheese.py

```
def test_def_prefs_full():
    cheese.write_default_cheese_preferences()
    expected = cheese._default_prefs
    actual = cheese.read_cheese_preferences()
    assert expected == actual
```

Одна из проблем с этим заключается в том, что любой, кто запустит этот тестовый код, перезапишет свой собственный cheese-файл настроек, и в этом нет ничего хорошего.

Если у пользователя определен `HOME`, `set`, `os.path.expanduser()` заменит `~` всем, что находится в переменной окружения пользователя `HOME`. Давайте создадим временный каталог и перенаправим `HOME`, чтобы указать на этот новый временный каталог:

ch4/monkey/test_cheese.py

```
def test_def_prefs_change_home(tmpdir, monkeypatch):
    monkeypatch.setenv('HOME', tmpdir.mkdir('home'))
    cheese.write_default_cheese_preferences()
    expected = cheese._default_prefs
    actual = cheese.read_cheese_preferences()
    assert expected == actual
```

Это довольно хороший тест, но `HOME` кажется немного зависимым от операционной системы. И если заглянем в онлайн-документацию для `expanduser()`, где будет некоторая тревожная информация, в том числе «On Windows, HOME and USERPROFILE will be used if set, otherwise combination of....». Ого! Это может быть плохо для тех, кто тестирует под Windows. Может быть, мы должны принять другой подход.

Вместо того, чтобы исправлять переменную окружения `HOME`, давайте запатчим `expanduser`:

ch4/monkey/test_cheese.py

```
def test_def_prefs_change_expanduser(tmpdir, monkeypatch):
    fake_home_dir = tmpdir.mkdir('home')
    monkeypatch.setattr(cheese.os.path, 'expanduser',
                        (lambda x: x.replace('~', str(fake_home_dir))))
    cheese.write_default_cheese_preferences()
    expected = cheese._default_prefs
    actual = cheese.read_cheese_preferences()
    assert expected == actual
```

Во время теста все, что в модуле `cheese` вызывает `os.path.expanduser()` получает вместо этого наше лямбда-выражение. Эта небольшая функция использует функцию модуля регулярного выражения `re.sub` для замены `~` нашим новым временным каталогом. Теперь мы исполняем `setenv()` и `setattr()` для исправления переменных и атрибутов среды. Затем, `setitem()`.

Предположим, мы обеспокоены тем, что произойдет, если файл уже существует. Мы хотим быть уверенным, что он будет перезаписан по умолчанию, когда вызывается `write_default_cheese_preferences()`:

ch4/monkey/test_cheese.py

```
def test_def_prefs_change_defaults(tmpdir, monkeypatch):
    # запись в файл один раз
    fake_home_dir = tmpdir.mkdir('home')
    monkeypatch.setattr(cheese.os.path, 'expanduser',
                        (lambda x: x.replace('~', str(fake_home_dir))))
    cheese.write_default_cheese_preferences()
    defaults_before = copy.deepcopy(cheese._default_prefs)

    # изменение значений по умолчанию
    monkeypatch.setitem(cheese._default_prefs, 'slicing', ['provolone'])
    monkeypatch.setitem(cheese._default_prefs, 'spreadable', ['brie'])
    monkeypatch.setitem(cheese._default_prefs, 'salads', ['pepper jack'])
    defaults_modified = cheese._default_prefs

    # перезапись его измененными значениями по умолчанию
    cheese.write_default_cheese_preferences()

    # чтение и проверка
    actual = cheese.read_cheese_preferences()
    assert defaults_modified == actual
    assert defaults_modified != defaults_before
```

Поскольку `_default_prefs`-это словарь, мы можем использовать `monkeypatch.setitem()`, чтобы изменить элементы словаря только на время теста.

Мы использовали `setenv()`, `setattr()` и `setitem()`. Формы `del` очень похожи. Они просто удаляют переменную среды, атрибут или элемент словаря вместо того, чтобы что-то устанавливать. Последние два метода `monkeypatch` относятся к путям.

`syspath_prepend(path)` добавляет путь к `sys.path`, что приводит к тому, что ваш новый путь помещается в начало строки для каталогов и модулей. Что заключается в замене общесистемного модуля или пакета на `stub`-версию. Затем вы можете использовать файл `monkeypatch.syspath_prepend()`, чтобы добавить каталог вашей версии, а тестируемый код сначала найдет `stub`-версию.

`chdir(path)` изменяет текущий рабочий каталог во время теста. Это было бы полезно для тестирования сценариев командной строки и других утилит, которые зависят от текущего рабочего каталога. Вы можете создать временный каталог с любым содержимым, которое имеет смысл вашего скрипта, а затем использовать `monkeypatch.chdir(the_tmpdir)`.

Вы также можете использовать функции привязки `monkeypatch` в сочетании с `unittest.mock`, чтобы временно заменить атрибуты макетных объектами. Вы увидите это в Главе 7 "Использование `pytest` с другими инструментами" на стр. 125.

Использование `doctest_namespace`

Модуль `doctest` является частью стандартной библиотеки Python и позволяет помещать небольшие примеры кода функции в `docstrings` и тестировать их, чтобы убедиться, что они работают. Вы можете использовать `pytest` для поиска и запуска тестов `doctest` в коде Python с помощью флага `--doctest-modules`. С встроенной фикстурой `doctest_namespace`, вы можете создать фикстуру с `autouse`, чтобы добавить символы в пространство имен `pytest` используя во время работы `doctest` тесты. Это позволяет `docstrings` быть гораздо более читаемым.

`doctest_namespace` обычно используется для добавления импорта модулей в пространство имен, особенно когда соглашение Python заключено в сокращении имени модуля или пакета. Например, `numpy` часто импортируется с `import numpy as np`.

Давайте поиграем с примером. Допустим, у нас есть модуль с именем `unnecessary_math.py` с методами `multiply()` и `divide()`, которые мы проверим. Таким образом, мы располагаем некоторые примеры использования как в `docstring` файла, так и в `docstrings` функций:

ch4/dt/1/unnecessary_math.py

```
"""
This module defines multiply(a, b) and divide(a, b).

>>> import unnecessary_math as um

Here's how you use multiply:

>>> um.multiply(4, 3)
12
>>> um.multiply('a', 3)
'aaa'

Here's how you use divide:

>>> um.divide(10, 5)
2.0
"""

def multiply(a, b):
    """
    Returns a multiplied by b.

    >>> um.multiply(4, 3)
    12
    >>> um.multiply('a', 3)
    'aaa'
    """
    return a * b

def divide(a, b):
    """
    Returns a divided by b.
```

```
>>> um.divide(10, 5)
2.0
"""
return a / b
```

Поскольку имя `unnecessary_math` длинное, мы решили использовать `um` вместо этого, используя `import unnecessary_math as um` в верхней строке. Код в docstrings функций не включает оператор `import`, но продолжает использовать соглашение `um`. Проблема в том, что `pytest` обрабатывает каждую docstring кодом как другим тестом. Импорт в верхнюю docstring позволит первой части пройти, но код в docstrings функции не будет выполнен:

```
$ cd /path/to/code/ch4/dt/1
$ pytest -v --doctest-modules --tb=short unnecessary_math.py
===== test session starts =====

collected 3 items

unnecessary_math.py::unnecessary_math PASSED
unnecessary_math.py::unnecessary_math.divide FAILED
unnecessary_math.py::unnecessary_math.multiply FAILED

===== FAILURES =====
_____ [doctest] unnecessary_math.divide _____
034
035     Returns a divided by b.
036
037     >>> um.divide(10, 5)
UNEXPECTED EXCEPTION: NameError("name 'um' is not defined",)
Traceback (most recent call last):
...
File "<doctest unnecessary_math.divide[0]>", line 1, in <module>

NameError: name 'um' is not defined
...
_____ [doctest] unnecessary_math.multiply _____
022
023     Returns a multiplied by b.
024
025     >>> um.multiply(4, 3)
UNEXPECTED EXCEPTION: NameError("name 'um' is not defined",)
Traceback (most recent call last):
...
File "<doctest unnecessary_math.multiply[0]>", line 1, in <module>

NameError: name 'um' is not defined

/path/to/code/ch4/dt/1/unnecessary_math.py:23: UnexpectedException
===== 2 failed, 1 passed in 0.03 seconds =====
```

Один из способов исправить это-поместить инструкцию `import` в каждую docstring:

ch4/dt/2/unnecessary_math.py

```
"""
This module defines multiply(a, b) and divide(a, b).

>>> import unnecessary_math as um

Here's how you use multiply:

>>> um.multiply(4, 3)
```



```

12
>>> um.multiply('a', 3)
'aaa'

Here's how you use divide:

>>> um.divide(10, 5)
2.0
"""

def multiply(a, b):
    """
    Returns a multiplied by b.

    >>> import unnecessary_math as um
    >>> um.multiply(4, 3)
    12
    >>> um.multiply('a', 3)
    'aaa'
    """
    return a * b

def divide(a, b):
    """
    Returns a divided by b.

    >>> import unnecessary_math as um
    >>> um.divide(10, 5)
    2.0
    """
    return a / b

```

Это определенно устраняет проблему:

```

$ cd /path/to/code/ch4/dt/2
$ pytest -v --doctest-modules --tb=short unnecessary_math.py
===== test session starts =====

collected 3 items

unnecessary_math.py::unnecessary_math PASSED [ 33%]
unnecessary_math.py::unnecessary_math.divide PASSED [ 66%]
unnecessary_math.py::unnecessary_math.multiply PASSED [100%]

===== 3 passed in 0.03 seconds =====

```

Однако он также загромождает docstrings и не добавляет никакой реальной ценности читателям кода.

Встроенное фикстура `doctest_namespace`, используемая в `autouse` в файле `conftest.py` верхнего уровня, устранит проблему без изменени исходного кода:

ch4/dt/3/conftest.py

```

import pytest
import unnecessary_math

@pytest.fixture(autouse=True)
def add_um(doctest_namespace):
    doctest_namespace['um'] = unnecessary_math

```

Это указание `pytest` добавить имя `um` в `doctest_namespace`, как значение импортированного модуля `unnecessary_math`. С этим в файле `conftest.py` любые doctests, найденные в рамках этого `conftest.py` будут определять символ `um`.

Я расскажу о запуске doctest из pytest в главе 7 "Использование pytest с другими инструментами" на стр. 125.

Использование recwarn

Встроенная фикстура `recwarn` используется для проверки предупреждений, генерируемых тестируемым кодом. В Python вы можете добавлять предупреждения, которые очень похожи на утверждения, но используются для случаев, при которых не нужно останавливать выполнение. Например, предположим, что мы хотим прекратить поддерживать функцию, которую нам хотелось бы никогда не добавлять в пакет, но при этом включить для других пользователей. Мы можем поместить предупреждение в код и оставить его там для пары выпусков:

ch4/test_warnings.py

```
import warnings
import pytest

def lame_function():
    warnings.warn("Please stop using this", DeprecationWarning)
    # rest of function
```

Мы можем убедиться, что предупреждение выдается правильно с тестом:

ch4/test_warnings.py

```
def test_lame_function(recwarn):
    lame_function()
    assert len(recwarn) == 1
    w = recwarn.pop()
    assert w.category == DeprecationWarning
    assert str(w.message) == 'Please stop using this'
```

Значение `recwarn` действует как список предупреждений, и каждое предупреждение в списке имеет определенную `category` (категорию), `message` (сообщение), `filename` (имя файла) и `lineno` (номер строки), как показано в коде.

Сбор предупреждений происходит с начала теста. Иногда это неудобно, потому что та часть теста, в которой вы заботитесь о предупреждении находится где-то в конце. И тогда вы можете использовать `recwarn.clear()`, чтобы очистить список перед тестом, в котором вы озадачили сбор предупреждений.

В дополнение к `recwarn`, `pytest` может проверять предупреждения с помощью `pytest.warns()`:

ch4/test_warnings.py

```
def test_lame_function_2():
    with pytest.warns(None) as warning_list:
        lame_function()

    assert len(warning_list) == 1
    w = warning_list.pop()
    assert w.category == DeprecationWarning
    assert str(w.message) == 'Please stop using this'
```

Контекстный менеджер `pytest.warns()` предоставляет элегантный способ отделения части кода для проверки предупреждения. Элемент `pytest.warns()` и диспетчер контекста `pytest.warns()` обеспечивают аналогичную функциональность, поэтому решение о том, что использовать, является исключительно вопросом вкуса.

Упражнения

- 21.04.2019Python Testing с pytest. Builtin Fixtures, Глава 4 / Хабр
1. В `ch4/cache/test_slower.py` есть фикстура `autouse`, называемая `check_duration()`. Скопируйте её в `ch3/tasks_proj/tests/conftest`.

2. Выполните тесты из Главы 3.

3. Для реально очень быстрых тестов, 2х действительно быстро все еще очень быстро. Вместо 2х измените фикстуру, чтобы проверить н секунды плюс 2х на последнюю продолжительность.

4. Запустите `pytest` с измененной фикстурой. Результаты кажутся разумными?

Что дальше

В этой главе вы рассмотрели несколько встроенных фикстур `pytest`. Далее вы более подробно рассмотрите Плагины. Нюансы написания б плагинов могут стать книгой сами по себе; однако небольшие пользовательские плагины являются регулярной частью экосистемы `pytest`.



Теги: `pytest`

↑

+10

↓

📌


14

👁

393

💬

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

Разработка WHATSAPP бота

11 откликов · 65 просмотров

100% за пр

Ищу напарника на react native для совместной работы

5 откликов · 76 просмотров

100% за пр

Разработка модуля поискового механизма (Elasticsearch)

9 откликов · 49 просмотров

50% за пр

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			