AlekSandrDr вчера в 17:10

Python Testing с pytest. Глава 2, Написание тестовых функций

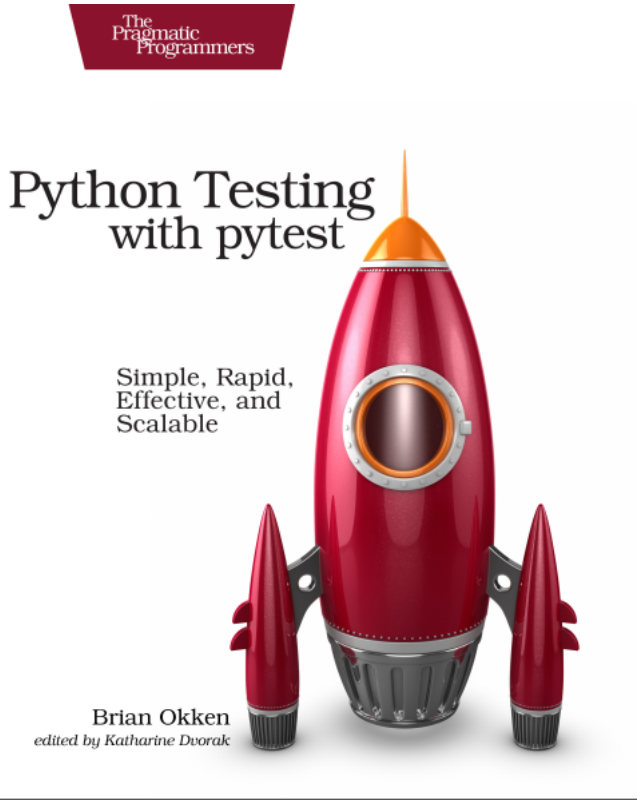
Автор оригинала: Okken Brian

Python

Перевод Tutorial

Вернуться Дальше

Вы узнаете, как организовать тесты в классы, модули и каталоги. Затем я покажу вам, как использовать маркеры, чтобы отметить тесты вы хотите запустить, и обсудить, как встроенные маркеры могут помочь вам пропустить тесты и отметить тесты, ожид неудачи. Наконец, я расскажу о параметризации тестов, которая позволяет тестам вызываться с разными данными.



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 примера Но что бы следовать вместе с задачами проекта, или адаптировать примеры тестирования для проверки своего собственного проекта (р у вас развязаны!), вы должны перейти на веб-страницу книги и скачать работу. Там же, на веб-странице книги есть ссылка для сообщени errata и дискуссионный форум.

Под спойлером приведен список статей этой серии.

Оглавление

В предыдущей главе вы запустили pytest. Вы видели, как запустить его с файлами и каталогами и сколько из опций работали. В этой главе узнаете, как писать тестовые функции в контексте тестирования пакета Python. Если вы используете pytest для тестирования чего-либо, кр пакета Python, большая часть этой главы будет полезна.

Мы напишем тесты для пакета Tasks. Прежде чем мы это сделаем, я расскажу о структуре распространяемого пакета Python и тестах для н также о том, как заставить тесты видеть тестируемый пакет. Затем я покажу вам, как использовать assert в тестах, как тесты обрабатывают непредвиденные исключения и тестируют ожидаемые исключения.

В конце концов, у нас будет много тестов. Таким образом, вы узнаете, как организовать тесты в классы, модули и каталоги. Затем я покажу как использовать маркеры, чтобы отметить, какие тесты вы хотите запустить, и обсудить, как встроенные маркеры могут помочь вам пропу тесты и отметить тесты, ожидая неудачи. Наконец, я расскажу о параметризации тестов, которая позволяет тестам вызываться с разными данными.

Прим.переводчика: Если вы используете версию Python 3.5 или 3.6 то при выполнении тестов Главы 2 могут возникнуть сообщени вот такого вида

```
============================================================================ warnings summary =============================
func/test_unique_id_4.py::test_unique_id_2
  c:\venv35\lib\site-packages\tinydb\database.py:52: DeprecationWarning: eids has been renamed to doc_ids
    warnings.warn('eids has been renamed to doc_ids', DeprecationWarning)
  c:\venv35\lib\site-packages\tinydb\database.py:52: DeprecationWarning: eids has been renamed to doc_ids
    warnings.warn('eids has been renamed to doc_ids', DeprecationWarning)
  c:\venv35\lib\site-packages\tinydb\database.py:52: DeprecationWarning: eids has been renamed to doc_ids
    warnings.warn('eids has been renamed to doc_ids', DeprecationWarning)
```

Эта проблема лечится исправлением `...\code\tasks_proj\src\tasks\tasksdb_tinydb.py` и повторной установкой пакета tasks

```
$ cd /path/to/code
$ pip install ./tasks_proj/`
```

Исправить надо именованные параметры `eids` на `doc_ids` и `eid` на `doc_id` в модуле `...\code\tasks_proj\src\tasks\tasksdb_tinydb.py`

Пояснения Смотри `#83783` здесь

Тестирование пакета

Чтобы узнать, как писать тестовые функции для пакета Python, мы будем использовать пример проекта Tasks, как описано в проекте Tasks странице xii. Задачи представляет собой пакет Python, который включает в себя инструмент командной строки с тем же именем, задачи.

Приложение 4 «Packaging and Distributing Python Projects» на стр. 175 включает объяснение того, как распределять ваши проекты локально внутри небольшой команды или глобально через PyPI, поэтому я не буду подробно разбираться в том, как это сделать; однако давайте бы рассмотрим, что находится в проекте «Tasks» и как разные файлы вписываются в историю тестирования этого проекта.

Ниже приведена файловая структура проекта Tasks:

```
tasks_proj/
├── CHANGELOG.rst
├── LICENSE
├── MANIFEST.in
├── README.rst
├── setup.py
├── src
│   └── tasks
│       ├── __init__.py
│       ├── api.py
│       ├── cli.py
│       ├── config.py
│       ├── tasksdb_pymongo.py
│       └── tasksdb_tinydb.py
└── tests
    ├── conftest.py
    ├── pytest.ini
    ├── func
    │   ├── __init__.py
    │   ├── test_add.py
    │   └── ...
    └── unit
        ├── __init__.py
        ├── test_task.py
        └── ...
```

Я включил полный список проекта (за исключением полного списка тестовых файлов), чтобы указать, как тесты вписываются в остальную проекта, и указать на несколько файлов, которые имеют ключевое значение для тестирования, а именно *conftest.py, pytest.ini*, различные

__init__.py файлы и *setup.py*.

Все тесты хранятся в *tests* и отдельно от исходных файлов пакета в *src*. Это не требование pytest, но это лучшая практика.

Все файлы верхнего уровня, *CHANGELOG.rst, LICENSE, README.rst, MANIFEST.in*, и *setup.py*, более подробно рассматриваются в Прилож 4, Упаковка и распространение проектов Python, на стр. 175. Хотя *setup.py* важен для построения дистрибутива из пакета, а также для возможности установить пакет локально, чтобы пакет был доступен для импорта.

Функциональные и модульные тесты разделены на собственные каталоги. Это произвольное решение и не обязательно. Однако организац тестовых файлов в несколько каталогов позволяет легко запускать подмножество тестов. Мне нравится разделять функциональные и моду тесты, потому что функциональные тесты должны ломаться, только если мы намеренно изменяя функциональность системы, в то время ка модульные тесты могут сломаться во время рефакторинга или изменения реализации.

Проект содержит два типа файлов `__init__.py`: найденные в каталоге `src/` и те, которые находятся в `tests/`. Файл `src/tasks/__init__.py` сообщает Python, что каталог является пакетом. Он также выступает в качестве основного интерфейса для пакета, когда кто-то использует `tasks`. Он содержит код для импорта определенных функций из `api.py`, так что `cli.py` и наши тестовые файлы могут обращаться к функци пакета, например `tasks.add()`, вместо того, чтобы выполнять `task.api.add ()`. Файлы `tests/func/__init__.py` и `tests/unit/__init__.py` Они указывают pytest подняться вверх на один каталог, чтобы найти корень тестового каталога и `pytest.ini`-файл.

Файл `pytest.ini` не является обязательным. Он содержит общую конфигурацию pytest для всего проекта. В вашем проекте должно быть н более одного из них. Он может содержать директивы, которые изменяют поведение pytest, например, настрйки списка параметров, которы всегда будут использоваться. Вы узнаете все о `pytest.ini` в главе 6 «Конфигурация» на стр. 113.

Файл conftest.py также является необязательным. Он считается pytest как “local plugin” и может содержать hook functions и fixtures. *Hook fun* являются способом вставки кода в часть процесса выполнения pytest для изменения работы pytest. Fixtures — это setup и teardown функци которые выполняются до и после тестовых функций и могут использоваться для представления ресурсов и данных, используемых тестами (Fixtures обсуждаются в главе 3, pytest Fixtures, на стр. 49 и главе 4, Builtin Fixtures, на стр. 71, а hook functions бсуждаются в главе 5 «Плаги стр. 95.) Hook functions и fixtures, которые используются в тестах в нескольких подкаталогах, должны содержаться в tests/conftest.py. Вы мож иметь несколько файлов conftest.py; например, можно иметь по одному в тестах и по одному для каждой поддиректории tests.

Если вы еще этого не сделали, вы можете загрузить копию исходного кода для этого проекта на веб-сайте книги. Альтернативно, вы можете работать над своим проектом с аналогичной структурой.

Вот test_task.py:

ch2/tasks_proj/tests/unit/test_task.py

```
"""Test the Task data type."""

# -*- coding: utf-8 -*-
from tasks import Task

def test_asdict():
    """_asdict() должен возвращать словарь."""
    t_task = Task('do something', 'okken', True, 21)
    t_dict = t_task._asdict()
    expected = {'summary': 'do something',
                'owner': 'okken',
                'done': True,
                'id': 21}
    assert t_dict == expected

def test_replace():
    """replace () должен изменить переданные данные в полях."""
    t_before = Task('finish book', 'brian', False)
    t_after = t_before._replace(id=10, done=True)
    t_expected = Task('finish book', 'brian', True, 10)
    assert t_after == t_expected

def test_defaults():
    """Использование вызова без параметров должно применить значения по умолчанию."""
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2
```

```
def test_member_access():
    """Проверка .field  функциональность namedtuple."""
    t = Task('buy milk', 'brian')
    assert t.summary == 'buy milk'
    assert t.owner == 'brian'
    assert (t.done, t.id) == (False, None)
```

В файле *test_task.py* указан этот оператор импорта:

```
from tasks import Task
```

Лучший способ позволить тестам импортировать tasks или что-то импортировать из tasks — установить tasks локально с помощью pip. Это возможно, потому что есть файл setup.py для прямого вызова pip.

Установите tasks, запустив `pip install .` или `pip install -e .` from the tasks_proj directory. Или другой вариант запустить `pip install -e tasks_proj` из каталога на один уровень выше:

```
$ cd /path/to/code
$ pip install ./tasks_proj/
$ pip install --no-cache-dir ./tasks_proj/
Processing ./tasks_proj
Collecting click (from tasks==0.1.0)
  Downloading click-6.7-py2.py3-none-any.whl (71kB)
  ...
Collecting tinydb (from tasks==0.1.0)
  Downloading tinydb-3.4.0.tar.gz
Collecting six (from tasks==0.1.0)
  Downloading six-1.10.0-py2.py3-none-any.whl
Installing collected packages: click, tinydb, six, tasks
  Running setup.py install for tinydb ... done
  Running setup.py install for tasks ... done
Successfully installed click-6.7 six-1.10.0 tasks-0.1.0 tinydb-3.4.0
```

Если вы хотите только выполнять тесты для tasks, эта команда подойдет. Если вы хотите иметь возможность изменять исходный код во вре установки tasks, вам необходимо использовать установку с опцией -e (для editable "редактируемый"):

```
$ pip install -e ./tasks_proj/
Obtaining file:///path/to/code/tasks_proj
Requirement already satisfied: click in
  /path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Requirement already satisfied: tinydb in
  /path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Requirement already satisfied: six in
  /path/to/venv/lib/python3.6/site-packages (from tasks==0.1.0)
Installing collected packages: tasks
  Found existing installation: tasks 0.1.0
    Uninstalling tasks-0.1.0:
      Successfully uninstalled tasks-0.1.0
  Running setup.py develop for tasks
Successfully installed tasks
```

Теперь попробуем запустить тесты:

```
$ cd /path/to/code/ch2/tasks_proj/tests/unit
$ pytest test_task.py
==================== test session starts ======================
collected 4 items
test_task.py ....
=================== 4 passed in 0.01 seconds ==================
```

Импорт сработал! Остальные тесты теперь могут безопасно использовать задачи импорта. Теперь напишем несколько тестов.

Использование операторов assert

Когда вы пишете тестовые функции, обычный оператор Python-а assert является вашим основным инструментом для сообщения о сбое тес
Простота этого в pytest блестящая. Это то, что заставляет многих разработчиков использовать pytest поверх других фреймворков.

Если вы использовали любую другую платформу тестирования, вы, вероятно, видели различные вспомогательные функции assert. Наприм
ниже приведен список некоторых форм assert и вспомогательных функций assert:

pytest	unittest
assert something	assertTrue(something)
assert a == b	assertEqual(a, b)
assert a <= b	assertLessEqual(a, b)
...	...

С помощью pytest вы можете использовать assert <выражение> с любым выражением. Если выражение будет вычисляться как False, когда
будет преобразовано в bool, тест завершится с ошибкой.

pytest включает функцию, называемую assert rewriting, которая перехватывает assert calls и заменяет их тем, что может рассказать вам бол
том, почему ваши утверждения не удались. Давайте посмотрим, насколько полезно это переписывание, если посмотреть на несколько оши
утверждения:

ch2/tasks_proj/tests/unit/test_task_fail.py

```
"""Используем the Task type для отображения сбоев тестов."""
from tasks import Task

def test_task_equality():
    """Разные задачи не должны быть равными."""
    t1 = Task('sit there', 'brian')
    t2 = Task('do something', 'okken')
    assert t1 == t2

def test_dict_equality():
    """Различные задачи, сравниваемые как dicts, не должны быть равны."""
    t1_dict = Task('make sandwich', 'okken')._asdict()
    t2_dict = Task('make sandwich', 'okkem')._asdict()
    assert t1_dict == t2_dict
```

Все эти тесты терпят неудачу, но интересна информация в трассировке:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\unit>pytest test_task_fail.py
============================= test session starts =============================

collected 2 items

test_task_fail.py FF

================================== FAILURES ===================================
_____________________________ test_task_equality ______________________________

    def test_task_equality():
        """Different tasks should not be equal."""
        t1 = Task('sit there', 'brian')
        t2 = Task('do something', 'okken')
>       assert t1 == t2
E       AssertionError: assert Task(summary=...alse, id=None) == Task(summary='...alse, id=None)
E         At index 0 diff: 'sit there' != 'do something'
E         Use -v to get the full diff
```

```
test_task_fail.py:9: AssertionError
_____________________________ test_dict_equality ______________________________

    def test_dict_equality():
        """Different tasks compared as dicts should not be equal."""
        t1_dict = Task('make sandwich', 'okken')._asdict()
        t2_dict = Task('make sandwich', 'okkem')._asdict()
>       assert t1_dict == t2_dict
E       AssertionError: assert OrderedDict([...('id', None)]) == OrderedDict([(...('id', None)])
E         Omitting 3 identical items, use -vv to show
E         Differing items:
E         {'owner': 'okken'} != {'owner': 'okkem'}
E         Use -v to get the full diff

test_task_fail.py:16: AssertionError
========================== 2 failed in 0.30 seconds ==========================
```

Вот это да! Это очень много информации. Для каждого неудачного теста точная строка ошибки отображается с помощью > указателя на от
Строки E показывают дополнительную информацию о сбое assert, чтобы помочь вам понять, что пошло не так.

Я намеренно поставил два несовпадения в `test_task_equality()`, но только первое было показано в предыдущем коде. Давайте попробуе
раз с флагом `-v`, как предложено в сообщении об ошибке :

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\unit>pytest -v test_task_fail.py
============================= test session starts =============================

collected 2 items

test_task_fail.py::test_task_equality FAILED
test_task_fail.py::test_dict_equality FAILED

================================== FAILURES ===================================
_____________________________ test_task_equality ______________________________

    def test_task_equality():
        """Different tasks should not be equal."""
        t1 = Task('sit there', 'brian')
        t2 = Task('do something', 'okken')
>       assert t1 == t2
E       AssertionError: assert Task(summary=...alse, id=None) == Task(summary='...alse, id=None)
E         At index 0 diff: 'sit there' != 'do something'
E         Full diff:
E         - Task(summary='sit there', owner='brian', done=False, id=None)
E         ?                ^^^  ^^^          ^^^^
E         + Task(summary='do something', owner='okken', done=False, id=None)
E         ?               +++ ^^^  ^^^          ^^^^

test_task_fail.py:9: AssertionError
_____________________________ test_dict_equality ______________________________

    def test_dict_equality():
        """Different tasks compared as dicts should not be equal."""
        t1_dict = Task('make sandwich', 'okken')._asdict()
        t2_dict = Task('make sandwich', 'okkem')._asdict()
>       assert t1_dict == t2_dict
E       AssertionError: assert OrderedDict([...('id', None)]) == OrderedDict([(...('id', None)])
E         Omitting 3 identical items, use -vv to show
E         Differing items:
E         {'owner': 'okken'} != {'owner': 'okkem'}
E         Full diff:
E         {'summary': 'make sandwich',
E         -  'owner': 'okken',
E         ?               ^...
E
E         ...Full output truncated (5 lines hidden), use '-vv' to show

test_task_fail.py:16: AssertionError
========================== 2 failed in 0.28 seconds ==========================
```

Ну, я думаю, что это чертовски круто! pytest не только смог найти оба различия, но и показал нам, где именно эти различия. В этом примере используется только equality assert; на веб-сайте pytest.org можно найти еще много разновидностей оператора assert с удивительной информацией об отладке трассировки.

Ожидание Исключений (expected exception)

Исключения(Exceptions) могут возникать в нескольких местах Tasks API. Давайте быстро заглянем в функции, найденные в *tasks/api.py*:

```
def add(task): # type: (Task) -\> int
def get(task_id): # type: (int) -\> Task
def list_tasks(owner=None): # type: (str|None) -\> list of Task
def count(): # type: (None) -\> int
def update(task_id, task): # type: (int, Task) -\> None
def delete(task_id): # type: (int) -\> None
def delete_all(): # type: () -\> None
def unique_id(): # type: () -\> int
def start_tasks_db(db_path, db_type): # type: (str, str) -\> None
def stop_tasks_db(): # type: () -\> None
```

Существует соглашение между CLI-кодом в *cli.py* и кодом API в *api.py* относительно того, какие типы будут передаваться в функции API. Вь API — это место, где я ожидаю, что исключения будут подняты, если тип неверен. Чтобы удостовериться, что эти функции вызывают исклю если они вызваны неправильно, используйте неправильный тип в тестовой функции, чтобы преднамеренно вызвать исключения TypeError использовать с pytest.raises (expected exception), например:

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
"""Проверка на ожидаемые исключения из-за неправильного использования API."""

import pytest
import tasks

def test_add_raises():
    """add() должно возникнуть исключение с неправильным типом param."""
    with pytest.raises(TypeError):
        tasks.add(task='not a Task object')
```

В `test_add_raises()`, с `pytest.raises(TypeError)`: оператор сообщает, что все, что находится в следующем блоке кода, должно вызвать исключение TypeError. Если исключение не вызывается, тест завершается неудачей. Если тест вызывает другое исключение, он завершает неудачей.

Мы только что проверили тип исключения в `test_add_raises()`. Можно также проверить параметры исключения. Для `start_tasks_db(db_p db_type)`, не только *db_type* должен быть строкой, это действительно должна быть либо 'tiny' или 'mongo'. Можно проверить, чтобы убедить сообщение об исключении является правильным, добавив excinfo:

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
def test_start_tasks_db_raises():
    """Убедитесь, что неподдерживаемая БД вызывает исключение."""
    with pytest.raises(ValueError) as excinfo:
        tasks.start_tasks_db('some/great/path', 'mysql')
    exception_msg = excinfo.value.args[0]
    assert exception_msg == "db_type must be a 'tiny' or 'mongo'"
```

Это позволяет нам более внимательно рассмотреть это исключение. Имя переменной после as (в данном случае excinfo) заполняется сведениями об исключении и имеет тип ExceptionInfo.

В нашем случае, мы хотим убедиться, что первый (и единственный) параметр исключения соответствует строке.

Marking Test Functions

pytest обеспечивает классный механизм, позволяющий помещать маркеры в тестовые функции. Тест может иметь более одного маркера, а маркер может быть в нескольких тестах.

Маркеры обретут для вас смысл после того, как вы увидите их в действии. Предположим, мы хотим запустить подмножество наших тестов качестве быстрого "smoke test", чтобы получить представление о том, есть ли какой-то серьезный разрыв в системе. Smoke tests по соглаш не являются всеобъемлющими, тщательными наборами тестов, но выбранным подмножеством, которое можно быстро запустить и дать разработчик достойное представление о здоровье всех частей системы.

Чтобы добавить набор тестов smoke в проект Tasks, нужно добавить `@mark.pytest.smoke` для некоторых тестов. Давайте добавим его к нескольким тестам `test_api_exceptions.py` (обратите внимание, что маркеры *smoke* и *get* не встроены в pytest; я просто их придумал):

ch2/tasks_proj/tests/func/test_api_exceptions.py

```
@pytest.mark.smoke
def test_list_raises():
    """list() должно возникнуть исключение с неправильным типом param."""
    with pytest.raises(TypeError):
        tasks.list_tasks(owner=123)

@pytest.mark.get
@pytest.mark.smoke
def test_get_raises():
    """get() должно возникнуть исключение с неправильным типом param."""
    with pytest.raises(TypeError):
        tasks.get(task_id='123')
```

Теперь давайте выполним только те тесты, которые помечены `-m marker_name`:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests>cd func

(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v -m "smoke" test_api_exceptions.py
============================= test session starts ==============================

collected 7 items

test_api_exceptions.py::test_list_raises PASSED
test_api_exceptions.py::test_get_raises PASSED

============================= 5 tests deselected ==============================
================== 2 passed, 5 deselected in 0.18 seconds ====================

(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v -m "get" test_api_exceptions.py
============================= test session starts ==============================

collected 7 items

test_api_exceptions.py::test_get_raises PASSED

============================= 6 tests deselected ==============================
================== 1 passed, 6 deselected in 0.13 seconds ====================
```

Помните, что `-v` сокращенно от `--verbose` и позволяет нам видеть имена тестов, которые выполняются. Использование-m 'smoke' запускае теста, помеченные @pytest.mark.smoke.

Использование `-m` 'get' запустит один тест, помеченный `@pytest.mark.get`. Довольно простой.

Все становится чудесатей и чудесатей! Выражение после `-m` может использовать `and`, `or` и `not` комбинировать несколько маркеров:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v -m "smoke and get" test_api_exceptions.py
============================= test session starts ==============================
```

```
collected 7 items

test_api_exceptions.py::test_get_raises PASSED

============================== 6 tests deselected ==============================
================== 1 passed, 6 deselected in 0.13 seconds ==================
```

Это мы провели тест только с маркерами `smoke` и `get`. Мы можем использовать и `not`:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v -m "smoke and not get" test_api_exceptions.py
============================== test session starts ==============================

collected 7 items

test_api_exceptions.py::test_list_raises PASSED

============================== 6 tests deselected ==============================
================== 1 passed, 6 deselected in 0.13 seconds ==================
```

Добавление `-m 'smoke and not get'` выбрало тест, который был отмечен с помощью `@pytest.mark.smoke`, но не `@pytest.mark.get`.

Заполнение Smoke Test

Предыдущие тесты еще не кажутся разумным набором `smoke test`. Мы фактически не касались базы данных и не добавляли никаких зада Конечно `smoke test` должен был бы сделать это.

Давайте добавим несколько тестов, которые рассматривают добавление задачи, и используем один из них как часть нашего набора тестов *smoke*:

ch2/tasks_proj/tests/func/test_add.py

```
"""Проверьте функцию API tasks.add ()."""

import pytest
import tasks
from tasks import Task

def test_add_returns_valid_id():
    """tasks.add(valid task) должен возвращать целое число."""
    # GIVEN an initialized tasks db
    # WHEN a new task is added
    # THEN returned task_id is of type int
    new_task = Task('do something')
    task_id = tasks.add(new_task)
    assert isinstance(task_id, int)

@pytest.mark.smoke
def test_added_task_has_id_set():
    """Убедимся, что поле task_id установлено tasks.add()."""
    # GIVEN an initialized tasks db
    #   AND a new task is added
    new_task = Task('sit in chair', owner='me', done=True)
    task_id = tasks.add(new_task)

    # WHEN task is retrieved
    task_from_db = tasks.get(task_id)

    # THEN task_id matches id field
    assert task_from_db.id == task_id
```

Оба этих теста имеют комментарий GIVEN к инициализированной БД tasks, но в тесте нет инициализированной базы данных. Мы можем определить fixture для инициализации базы данных перед тестом и очистки после теста:

ch2/tasks_proj/tests/func/test_add.py

```
@pytest.fixture(autouse=True)
def initialized_tasks_db(tmpdir):
    """Connect to db before testing, disconnect after."""
    # Setup : start db
    tasks.start_tasks_db(str(tmpdir), 'tiny')

    yield  # здесь происходит тестирование

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

Фикстура, tmpdir, используемая в данном примере, является встроенной (builtin fixture). Вы узнаете все о встроенных фикстурах в главе 4, F Fixtures, на странице 71, и вы узнаете о написании собственных фикстур и о том, как они работают в главе 3, pytest Fixtures, на странице 4 включая параметр autouse, используемый здесь.

autouse, используемый в нашем тесте, показывает, что все тесты в этом файле будут использовать fixture. Код перед `yield` выполняется пе каждым тестом; код после `yield` выполняется после теста. При желании yield может возвращать данные в тест. Вы рассмотрите все это и м другое в последующих главах, но здесь нам нужно каким-то образом настроить базу данных для тестирования, поэтому я больше не могу ж и должен показть вам сей прибор (фикстуру конечно!). (pytest также поддерживает старомодные функции setup и teardown, такие как те, что используется в **unittest** и **nose**, но они не так интересны. Однако, если вам все же интересно, они описаны в Приложении 5, xUnit Fixtures, р 183.)

Давайте пока отложим обсуждение фикстур и перейдем к началу проекта и запустим наш *smoke test suite*:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>cd ..

(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests>cd ..

(venv33) ...\bopytest-code\code\ch2\tasks_proj>pytest -v -m "smoke"
============================= test session starts =============================

collected 56 items

tests/func/test_add.py::test_added_task_has_id_set PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED

============================= 53 tests deselected =============================
================== 3 passed, 53 deselected in 0.49 seconds ==================
```

Тут показано, что помеченные тесты из разных файлов могут выполняться вместе.

Пропуск Тестов (Skipping Tests)

Хотя маркеры, обсуждаемые в методах проверки маркировки, на стр. 31 были именами по вашему выбору, pytest включает в себя несколь полезных встроенных маркеров: `skip`, `skipif`, и `xfail`. В этом разделе я расскажу про `skip` и `skipif`, а в следующем `-xfail`.

Маркеры `skip` и `skipif` позволяют пропускать тесты, которые не нужно выполнять. Для примера, допустим, мы не знали, как должна работ `tasks.unique_id()`. Каждый вызов её должен возвращает другой номер? Или это просто номер, который еще не существует в базе данных

Во-первых, давайте напишем тест (заметим, что в этом файле тоже есть фикстура `initialized_tasks_db`; просто она здесь не показана):

ch2/tasks_proj/tests/func/`test_unique_id_1.py`

```
"""Test tasks.unique_id()."""

import pytest
import tasks
```

```
def test_unique_id():
    """Вызов unique_id () дважды должен возвращать разные числа."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2
```

Затем дайте ему выполниться:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest test_unique_id_1.py
============================= test session starts =============================

collected 1 item

test_unique_id_1.py F

================================== FAILURES ===================================
_____________________________ test_unique_id ______________________________

    def test_unique_id():
        """Calling unique_id() twice should return different numbers."""
        id_1 = tasks.unique_id()
        id_2 = tasks.unique_id()
>       assert id_1 != id_2
E       assert 1 != 1

test_unique_id_1.py:11: AssertionError
========================== 1 failed in 0.30 seconds ===========================
```

Хм. Может быть, мы ошиблись. Посмотрев на API немного больше, мы видим, что docstring говорит """Return an integer that does not exist in db.""", что означает *Возвращает целое число, которое не существует в DB*. Мы могли бы просто изменить тест. Но вместо этого давайте отметим первый, который будет пропущен:

ch2/tasks_proj/tests/func/`test_unique_id_2.py`

```
@pytest.mark.skip(reason='misunderstood the API')
def test_unique_id_1():
    """Вызов unique_id () дважды должен возвращать разные числа."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2

def test_unique_id_2():
    """unique_id() должен вернуть неиспользуемый id."""
    ids = []
    ids.append(tasks.add(Task('one')))
    ids.append(tasks.add(Task('two')))
    ids.append(tasks.add(Task('three')))
    # захват уникального id
    uid = tasks.unique_id()
    # убеждаемся, что его нет в списке существующих идентификаторов
    assert uid not in ids
```

Отметить тест, который нужно пропустить, так же просто, как добавить `@pytest.марк.skip()` чуть выше тестовой функции.

Повторим :

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_unique_id_2.py
============================= test session starts =============================

collected 2 items

test_unique_id_2.py::test_unique_id_1 SKIPPED
test_unique_id_2.py::test_unique_id_2 PASSED
```

```
====================== 1 passed, 1 skipped in 0.19 seconds ======================
```

Теперь предположим, что по какой-то причине мы решили, что первый тест также должен быть действительным, и мы намерены сделать э работу в версии 0.2.0 пакета. Мы можем оставить тест на месте и использовать вместо этого skipif:

ch2/tasks_proj/tests/func/`test_unique_id_3.py`

```
@pytest.mark.skipif(tasks.__version__ < '0.2.0',
                    reason='not supported until version 0.2.0')
def test_unique_id_1():
    """Вызов unique_id () дважды должен возвращать разные числа."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2
```

Выражение, которое мы передаем в `skipif()`, может быть любым допустимым выражением Python. В этом конкретном, нашем случае, мы проверяем версию пакета. Мы включили причины как в *skip*, так и в *skipif*. Это не требуется в *skip*, но это требуется в *skipif*. Мне нравится включать обоснование причины (*reason*) для каждого *skip*, *skipif* или *xfail*. Вот вывод измененного кода:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest test_unique_id_3.py
============================= test session starts =============================

collected 2 items

test_unique_id_3.py s.

====================== 1 passed, 1 skipped in 0.20 seconds ======================
```

`s.` показывает, что один тест был пропущен(skipped), и один тест прошел(passed). Мы можем посмотреть, какой из них где-куда опцией `-v`:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_unique_id_3.py
============================= test session starts =============================

collected 2 items

test_unique_id_3.py::test_unique_id_1 SKIPPED
test_unique_id_3.py::test_unique_id_2 PASSED

====================== 1 passed, 1 skipped in 0.19 seconds ======================
```

Но мы все еще не знаем почему. Мы можем взглянуть на эти причины с `-rs`:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -rs test_unique_id_3.py
============================= test session starts =============================

collected 2 items

test_unique_id_3.py s.
=========================== short test summary info ===========================
SKIP [1] func\test_unique_id_3.py:8: not supported until version 0.2.0

====================== 1 passed, 1 skipped in 0.22 seconds ======================
```

Параметр `-r chars` содержит такой текст справки:

```
$ pytest --help
...
    -r chars
```

```
    show extra test summary info as specified by chars
    (показать дополнительную сводную информацию по тесту, обозначенному символами)
    (f)ailed, (E)error, (s)skipped, (x)failed, (X)passed,
    (p)passed, (P)passed with output, (a)all except pP.
...
```

Это не только полезно для понимания пробных пропусков, но также вы можете использовать его и для других результатов тестирования.

Маркировка тестов ожидающих сбоя

С помощью маркеров `skip` и `skipif` тест даже не выполняется, если он пропущен. С помощью маркера `xfail` мы указываем pytest запусти тестовую функцию, но ожидаем, что она потерпит неудачу. Давайте изменим наш тест `unique_id ()` снова, чтобы использовать `xfail`:

ch2/tasks_proj/tests/func/`test_unique_id_4.py`

```
@pytest.mark.xfail(tasks.__version__ < '0.2.0',
                   reason='not supported until version 0.2.0')
def test_unique_id_1():
    """Вызов unique_id() дважды должен возвращать разные номера."""
    id_1 = tasks.unique_id()
    id_2 = tasks.unique_id()
    assert id_1 != id_2

@pytest.mark.xfail()
def test_unique_id_is_a_duck():
    """Продемонстрирация xfail."""
    uid = tasks.unique_id()
    assert uid == 'a duck'

@pytest.mark.xfail()
def test_unique_id_not_a_duck():
    """Продемонстрирация xpass."""
    uid = tasks.unique_id()
    assert uid != 'a duck'
```

Running this shows:

Первый тест такой же, как и раньше, но с `xfail`. Следующие два теста такие же и отличаются только == vs.! =. Поэтому один из них должен пройти.

Выполнение этого показывает:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest  test_unique_id_4.py
============================= test session starts =============================

collected 4 items

test_unique_id_4.py xxX.

============== 1 passed, 2 xfailed, 1 xpassed in 0.36 seconds ================
```

X для XFAIL, что означает «ожидаемый отказ (*expected to fail*)». Заглавная X предназначен для XPASS или «ожидается, что он не сработает пройдет (*expected to fail but passed.*)».

`--verbose` перечисляет более подробные описания:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_unique_id_4.py
============================= test session starts =============================

collected 4 items

test_unique_id_4.py::test_unique_id_1 xfail
```

```
test_unique_id_4.py::test_unique_id_is_a_duck xfail
test_unique_id_4.py::test_unique_id_not_a_duck XPASS
test_unique_id_4.py::test_unique_id_2 PASSED

=============== 1 passed, 2 xfailed, 1 xpassed in 0.36 seconds ================
```

Вы можете настроить *pytest* так, чтобы тесты, которые прошли, но были помечены `xfail`, сообщались как FAIL. Это делается в *pytest.ini*:

```
[pytest]
xfail_strict=true
```

Я буду обсуждать *pytest.ini* подробнее в главе 6, Конфигурация, на стр. 113.

Выполнение подмножества тестов

Я говорил о том, как вы можете размещать маркеры в тестах и запускать тесты на основе маркеров. Подмножество тестов можно запустит несколькими другими способами. Можно выполнить все тесты или выбрать один каталог, файл, класс в файле или отдельный тест в файле классе. Вы еще не видели тестовых классов, поэтому посмотрите на них в этом разделе. Можно также использовать выражение для сопоставления имен тестов. Давайте взглянем на это.

A Single Directory

Чтобы запустить все тесты из одного каталога, используйте каталог как параметр для *pytest*:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj>pytest tests\func --tb=no
============================= test session starts =============================

collected 50 items

tests\func\test_add.py ..
tests\func\test_add_variety.py ............................
tests\func\test_api_exceptions.py .......
tests\func\test_unique_id_1.py F
tests\func\test_unique_id_2.py s.
tests\func\test_unique_id_3.py s.
tests\func\test_unique_id_4.py xxX.

==== 1 failed, 44 passed, 2 skipped, 2 xfailed, 1 xpassed in 1.75 seconds =====
```

Важная хитрость заключается в том, что использование `-v` показывает синтаксис для запуска определенного каталога, класса и теста.

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj>pytest -v tests\func --tb=no
============================= test session starts =============================
```

...

```
collected 50 items

tests\func\test_add.py::test_add_returns_valid_id PASSED
tests\func\test_add.py::test_added_task_has_id_set PASSED
tests\func\test_add_variety.py::test_add_1 PASSED
tests\func\test_add_variety.py::test_add_2[task0] PASSED
tests\func\test_add_variety.py::test_add_2[task1] PASSED
tests\func\test_add_variety.py::test_add_2[task2] PASSED
tests\func\test_add_variety.py::test_add_2[task3] PASSED
tests\func\test_add_variety.py::test_add_3[sleep-None-False] PASSED
...
tests\func\test_unique_id_2.py::test_unique_id_1 SKIPPED
tests\func\test_unique_id_2.py::test_unique_id_2 PASSED
...
tests\func\test_unique_id_4.py::test_unique_id_1 xfail
tests\func\test_unique_id_4.py::test_unique_id_is_a_duck xfail
tests\func\test_unique_id_4.py::test_unique_id_not_a_duck XPASS
```

```
tests\func\test_unique_id_4.py::test_unique_id_2 PASSED

==== 1 failed, 44 passed, 2 skipped, 2 xfailed, 1 xpassed in 2.05 seconds =====
```

Вы увидите синтаксис, приведенный здесь в следующих нескольких примерах.

Одиночный тест File/Module

Чтобы запустить файл, полный тестов, перечислите файл с относительным путем в качестве параметра к pytest:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest tests/func/test_add.py
========================== test session starts ==========================
collected 2 items   tests/func/test_add.py ..
======================== 2 passed in 0.05 seconds ========================
```

Мы уже делали это и не один раз.

Одиночная тестовая функция

Чтобы запустить одну тестовую функцию, добавьте `::` и имя тестовой функции:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v tests/func/test_add.py::test_add_returns_valid_id
========================== test session starts ==========================
collected 3 items
tests/func/test_add.py::test_add_returns_valid_id PASSED
======================== 1 passed in 0.02 seconds ========================
```

Используйте `-v`, чтобы увидеть, какая функция была запущена.

Одиночный Test Class

Here's an example:

Тестовые классы — это способ группировать тесты, которые по смыслу группируются вместе.
Вот пример:

ch2/tasks_proj/tests/func/`test_api_exceptions.py`

```
class TestUpdate():
    """Тест ожидаемых исключений с tasks.update()."""

    def test_bad_id(self):
        """non-int id должен поднять excption."""
        with pytest.raises(TypeError):
            tasks.update(task_id={'dict instead': 1},
                         task=tasks.Task())

    def test_bad_task(self):
        """A non-Task task должен поднять excption."""
        with pytest.raises(TypeError):
            tasks.update(task_id=1, task='not a task')
```

Так как это два связанных теста, которые оба тестируют функцию `update()`, целесообразно сгруппировать их в класс. Чтобы запустить толь
этот класс, сделайте так же, как мы сделали с функциями и добавьте `::`, затем имя класса в параметр вызова:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj>pytest -v tests/func/test_api_exceptions.py::TestUpdate
============================ test session starts ============================
```

```
collected 2 items

tests\func\test_api_exceptions.py::TestUpdate::test_bad_id PASSED
tests\func\test_api_exceptions.py::TestUpdate::test_bad_task PASSED

========================== 2 passed in 0.12 seconds ===========================
```

A Single Test Method of a Test Class

Если вы не хотите запускать весь тестовый класс, а только один метод — просто добавьте ещё раз `::` и имя метода:

```
$ cd /path/to/code/ch2/tasks_proj
$ pytest -v tests/func/test_api_exceptions.py::TestUpdate::test_bad_id
==================== test session starts ====================
collected 1 item
tests/func/test_api_exceptions.py::TestUpdate::test_bad_id PASSED
=================== 1 passed in 0.03 seconds ===================
```

Синтаксис группировки, отображаемый подробным списком

Помните, что синтаксис для запуска подмножества тестов по каталогу, файлу, функции, классу и методу не нужно запоминать. Формат та же, как и список тестовых функций при запуске `pytest -v`.

Набор тестов на основе базового имени теста

Параметр `-k` позволяет передать выражение для выполнения тестов, имена которых заданы выражением в качестве подстроки имени тест создания сложных выражений можно использовать `and`, `or` и `not` в выражении. Например, мы можем запустить все функции с именем `_rai`

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj>pytest -v -k _raises
============================ test session starts ============================

collected 56 items

tests/func/test_api_exceptions.py::test_add_raises PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
tests/func/test_api_exceptions.py::test_delete_raises PASSED
tests/func/test_api_exceptions.py::test_start_tasks_db_raises PASSED

============================ 51 tests deselected ============================
================== 5 passed, 51 deselected in 0.54 seconds ==================
```

Мы можем использовать `and` и `not` что бы исключить `test_delete_raises()` из сессии:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj>pytest -v -k "_raises and not delete"
============================ test session starts ============================

collected 56 items

tests/func/test_api_exceptions.py::test_add_raises PASSED
tests/func/test_api_exceptions.py::test_list_raises PASSED
tests/func/test_api_exceptions.py::test_get_raises PASSED
tests/func/test_api_exceptions.py::test_start_tasks_db_raises PASSED

============================ 52 tests deselected ============================
================== 4 passed, 52 deselected in 0.44 seconds ==================
```

В этом разделе вы узнали, как запускать определенные тестовые файлы, каталоги, классы и функции и как использовать выражения с `-k` д запуска определенных наборов тестов. В следующем разделе вы узнаете, как одна тестовая функция может превратиться во множество те случаев, позволяя тесту работать несколько раз с различными тестовыми данными.

[Parametrized Testing]: Параметризованное тестирование

Передача отдельных значений через функцию и проверка выходных данных, чтобы убедиться в их правильности, является распространен явлением в тестировании программного обеспечения. Однако единичного вызова функции с одним набором значений и одной проверкой правильности недостаточно для полной проверки большинства функций. Параметризованное тестирование-это способ отправить несколь наборов данных через один и тот же тест и иметь отчет pytest, если какой-либо из наборов не удался.

Чтобы помочь понять проблему, которую пытается решить параметризованное тестирование, давайте возьмем простой тест для `add()`:

ch2/tasks_proj/tests/func/`test_add_variety.py`

```
"""Проверка функции API tasks.add()."""

import pytest
import tasks
from tasks import Task

def test_add_1():
    """tasks.get () использует id, возвращаемый из add() works."""
    task = Task('breathe', 'BRIAN', True)
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    # все, кроме идентификатора, должно быть одинаковым
    assert equivalent(t_from_db, task)

def equivalent(t1, t2):
    """Проверяет эквивалентность двух задач."""
    # Сравнить все, кроме поля id
    return ((t1.summary == t2.summary) and
            (t1.owner == t2.owner) and
            (t1.done == t2.done))

@pytest.fixture(autouse=True)
def initialized_tasks_db(tmpdir):
    """Подключает к БД перед тестированием, отключает после."""
    tasks.start_tasks_db(str(tmpdir), 'tiny')
    yield
    tasks.stop_tasks_db()
```

При создании объекта tasks его полю `id` присваивается значение `None`. После добавления и извлечения из базы данных будет задано поле Поэтому мы не можем просто использовать `==`, чтобы проверить, правильно ли была добавлена и получена наша задача. Вспомогательная функция `equivalent()` проверяет все, кроме поля `id`. фикстура `autouse` используется, чтобы убедиться, что база данных доступна. Давайте убедимся, что тест прошел:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_add_variety.py::test_add_1
============================= test session starts =============================

collected 1 item

test_add_variety.py::test_add_1 PASSED

========================== 1 passed in 0.69 seconds ===========================
```

Тест кажется допустимым. Тем не менее, это просто проверка одной примерной задачи. Что делать, если мы хотим проверить множество вариантов задачи? Нет проблем. Мы можем использовать `@pytest.mark.parametrize(argnames, argvalues)` для передачи множества дан через один и тот же тест, например:

ch2/tasks_proj/tests/func/`test_add_variety.py`

```
@pytest.mark.parametrize('task',
                         [Task('sleep', done=True),
                          Task('wake', 'brian'),
                          Task('breathe', 'BRIAN', True),
                          Task('exercise', 'BrIaN', False)])
def test_add_2(task):
    """Демонстрирует параметризацию с одним параметром."""
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

Первый аргумент `parametrize()` — это строка с разделенным запятыми списком имен — 'task', в нашем случае. Второй аргумент — это спи значений, который в нашем случае представляет собой список объектов Task. pytest будет запускать этот тест один раз для каждой задачи сообщать о каждом отдельном тесте:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_add_variety.py::test_add_2
============================= test session starts =============================

collected 4 items

test_add_variety.py::test_add_2[task0] PASSED
test_add_variety.py::test_add_2[task1] PASSED
test_add_variety.py::test_add_2[task2] PASSED
test_add_variety.py::test_add_2[task3] PASSED

========================== 4 passed in 0.69 seconds ===========================
```

Использование `parametrize()` работает как нам надо. Однако давайте передадим задачи как кортежи, чтобы поглядеть, как будут работать несколько параметров теста:

ch2/tasks_proj/tests/func/`test_add_variety.py`

```
@pytest.mark.parametrize('summary, owner, done',
                         [('sleep', None, False),
                          ('wake', 'brian', False),
                          ('breathe', 'BRIAN', True),
                          ('eat eggs', 'BrIaN', False),
                          ])
def test_add_3(summary, owner, done):
    """Демонстрирует параметризацию с несколькими параметрами."""
    task = Task(summary, owner, done)
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

При использовании типов, которые легко преобразовать в строки с помощью pytest, идентификатор теста использует значения параметров отчете, чтобы сделать его доступным для чтения:

```
(venv35) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_add_variety.py::test_add_3
============================= test session starts =============================
platform win32 -- Python 3.5.2, pytest-3.5.1, py-1.5.3, pluggy-0.6.0 --
cachedir: ..\.pytest_cache
rootdir: ...\bopytest-code\code\ch2\tasks_proj\tests, inifile: pytest.ini
collected 4 items

test_add_variety.py::test_add_3[sleep-None-False] PASSED                 [ 25%]
test_add_variety.py::test_add_3[wake-brian-False] PASSED                 [ 50%]
test_add_variety.py::test_add_3[breathe-BRIAN-True] PASSED               [ 75%]
test_add_variety.py::test_add_3[eat eggs-BrIaN-False] PASSED             [100%]

========================== 4 passed in 0.37 seconds ===========================
```

Если хотите, вы можете использовать весь тестовый идентификатор, называемый узлом в терминологии pytest, для повторного запуска тес

```
(venv35) c:\BOOK\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_add_variety.py::test_add_3[sleep-None-False]
============================== test session starts ==============================

test_add_variety.py::test_add_3[sleep-None-False] PASSED                [100%]

========================== 1 passed in 0.22 seconds ===========================
```

Обязательно используйте кавычки, если в идентификаторе есть пробелы:

```
(venv35) c:\BOOK\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v "test_add_variety.py::test_add_3[eat eggs-BrIaN-Fa
e]"
============================== test session starts ==============================

collected 1 item

test_add_variety.py::test_add_3[eat eggs-BrIaN-False] PASSED             [100%]

========================== 1 passed in 0.56 seconds ===========================
```

Теперь вернемся к списку версий задач, но переместим список задач в переменную вне функции:

ch2/tasks_proj/tests/func/`test_add_variety.py`

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

@pytest.mark.parametrize('task', tasks_to_try)
def test_add_4(task):
    """Немного разные."""
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

Это удобно и код выглядит красиво. Но читаемость вывода трудно интерпретировать:

```
(venv35) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_add_variety.py::test_add_4
============================== test session starts ==============================

collected 5 items

test_add_variety.py::test_add_4[task0] PASSED                            [ 20%]
test_add_variety.py::test_add_4[task1] PASSED                            [ 40%]
test_add_variety.py::test_add_4[task2] PASSED                            [ 60%]
test_add_variety.py::test_add_4[task3] PASSED                            [ 80%]
test_add_variety.py::test_add_4[task4] PASSED                            [100%]

========================== 5 passed in 0.34 seconds ===========================
```

Удобочитаемость версии с несколькими параметрами хороша, как и список объектов задачи. Чтобы пойти на компромисс, мы можем использовать необязательный параметр ids для `parametrize()`, чтобы сделать наши собственные идентификаторы для каждого набора да задачи. Параметр `ids` должен быть списком строк той же длины, что и количество наборов данных. Однако, поскольку мы присвоили наше набору данных имя переменной `tasks_to_try`, мы можем использовать его для генерации идентификаторов:

ch2/tasks_proj/tests/func/`test_add_variety.py`

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

@pytest.mark.parametrize('task', tasks_to_try, ids=task_ids)
def test_add_5(task):
    """Demonstrate ids."""
    task_id = tasks.add(task)
    t_from_db = tasks.get(task_id)
    assert equivalent(t_from_db, task)
```

Давайте запустим это и посмотрим, как это выглядит:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_add_variety.py::test_add_5
============================= test session starts =============================

collected 5 items

test_add_variety.py::test_add_5[Task(sleep,None,True)] PASSED
test_add_variety.py::test_add_5[Task(wake,brian,False)0] PASSED
test_add_variety.py::test_add_5[Task(wake,brian,False)1] PASSED
test_add_variety.py::test_add_5[Task(breathe,BRIAN,True)] PASSED
test_add_variety.py::test_add_5[Task(exercise,BrIaN,False)] PASSED

========================== 5 passed in 0.45 seconds ===========================
```

И эти идентификаторы можно использовать для выполнения тестов:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v "test_add_variety.py::test_add_5[Task(exercise,BrIaN,F
e)]"
============================= test session starts =============================

collected 1 item

test_add_variety.py::test_add_5[Task(exercise,BrIaN,False)] PASSED

========================== 1 passed in 0.21 seconds ===========================
```

Нам определенно нужны кавычки для этих идентификаторов; в противном случае круглые и квадратные скобки будут путать shell. Вы такж
можете применить `parametrize()` к классам. При этом одни и те же наборы данных будут отправлены всем методам теста в классе:

ch2/tasks_proj/tests/func/`test_add_variety.py`

```
@pytest.mark.parametrize('task', tasks_to_try, ids=task_ids)
class TestAdd():
    """Демонстрация параметризации тестовых классов."""

    def test_equivalent(self, task):
        """Похожий тест, только внутри класса."""
        task_id = tasks.add(task)
        t_from_db = tasks.get(task_id)
        assert equivalent(t_from_db, task)

    def test_valid_id(self, task):
        """Мы можем использовать одни и те же данные или несколько тестов."""
        task_id = tasks.add(task)
        t_from_db = tasks.get(task_id)
        assert t_from_db.id == task_id
```

Вот он в действии:

```
(venv33) ...\bopytest-code\code\ch2\tasks_proj\tests\func>pytest -v test_add_variety.py::TestAdd
============================= test session starts =============================

collected 10 items

test_add_variety.py::TestAdd::test_equivalent[Task(sleep,None,True)] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(wake,brian,False)0] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(wake,brian,False)1] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(breathe,BRIAN,True)] PASSED
test_add_variety.py::TestAdd::test_equivalent[Task(exercise,BrIaN,False)] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(sleep,None,True)] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(wake,brian,False)0] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(wake,brian,False)1] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(breathe,BRIAN,True)] PASSED
test_add_variety.py::TestAdd::test_valid_id[Task(exercise,BrIaN,False)] PASSED

========================== 10 passed in 1.16 seconds ==========================
```

Вы также можете идентифицировать параметры, включив идентификатор рядом со значением параметра при передаче списка в декоратор `@pytest.mark.parametrize()`. Вы делаете это с помощью синтаксиса `pytest.param(<value\>, id="something")`:

В действии:

```
(venv35) ...\bopytest-code\code\ch2\tasks_proj\tests\func
$ pytest -v test_add_variety.py::test_add_6
=================================== test session starts ===================================

collected 3 items

test_add_variety.py::test_add_6[just summary] PASSED                                [ 33%]
test_add_variety.py::test_add_6[summary\owner] PASSED                               [ 66%]
test_add_variety.py::test_add_6[summary\owner\done] PASSED                          [100%]

================================ 3 passed, 6 warnings in 0.35 seconds =================================
```

Это полезно, когда `id` не может быть получен из значения параметра.

Упражнения

1. Загрузите проект для этой главы, `task_proj`, с веб-страницы этой главы и убедитесь, что вы можете установить его локально с помощь `install /path/to/tasks_proj`.
2. Изучите каталог тестов.
3. Запустите *pytest* с одним файлом.
4. Запускать *pytest* против одного каталога, например `tasks_proj/tests/func`. Используйте pytest для запуска тестов по отдельности, а та полный каталог одновременно. Там есть несколько неудачных тестов. Вы понимаете, почему они терпят неудачу?
5. Добавляйте xfail или пропускайте маркеры к ошибочным тестам, пока не сможете запустить pytest из каталога tests без аргументов и оп
6. У нас нет тестов для `tasks.count()`, среди прочих функций. Выберите непроверенную функцию API и подумайте, какие тестовые случа нужны, чтобы убедиться, что она работает правильно.
7. Что произойдет при попытке добавить задачу с уже установленным идентификатором? Есть некоторые отсутствующие тесты исключен `test_api_exceptions.py`. Посмотрите, можете ли вы заполнить недостающие исключения. (Это нормально посмотреть `api.py` для этог упражнения.)

Что дальше

В этой главе вы не увидели многих возможностей *pytest*. Но, даже с тем, что здесь описано, вы можете начать грузить свои тестовые компл Во многих примерах вы использовали фикстуру с именем `initialized_tasks_db`. Фикстуры могут отделять полученные и/или генерирован тестовые данные от реальных внутренностей тестовой функции.

Они также могут отделить общий код, чтобы несколько тестовых функций могли использовать одну и ту же настройку. В следующей главе в глубоко погрузитесь в чудесный мир фикстур pytest.

Вернуться Дальше

Теги: pytest

+7 16 699 Комментировать

29,0 Карма 148,9 Рейтинг 90 Подписчики

Александр Драгункин @AlekSandrDr
Пользователь

Поделиться публикацией

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