



Python



AlekSandrDr вчера в 17:09

Python Testing с pytest. Начало работы с pytest, Глава 1

Автор оригинала: Okken Brian

Перевод

Tutorial



Я обнаружил, что Python Testing с pytest является чрезвычайно полезным вводным руководством к среде тестирования pytest. Это уже приносит мне дивиденды в моей компании.

Chris Shaver

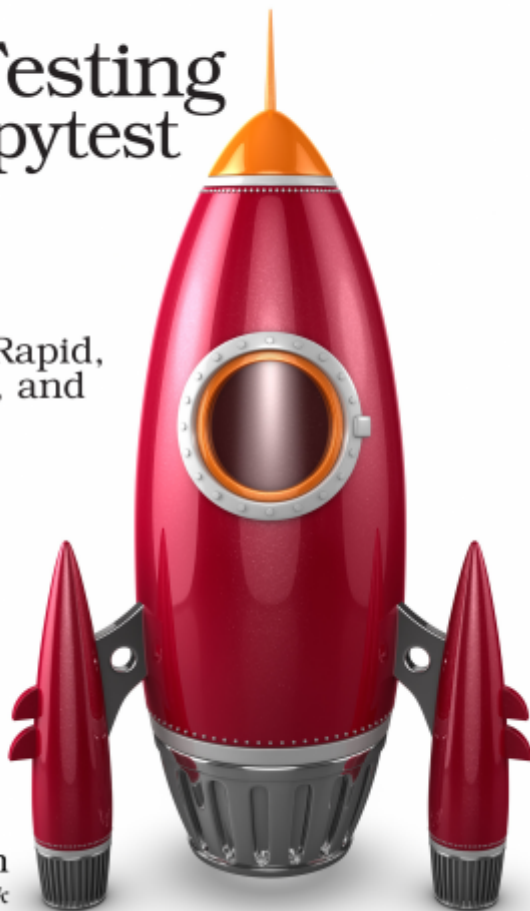
VP of Product, Uprising Technology

The Pragmatic Programmers

Python Testing with pytest

Simple, Rapid,
Effective, and
Scalable

Brian Okken
edited by Katharine Dvorak



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

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

Под спойлером приведен список статей этой серии.

► [Оглавление](#)

Поехали

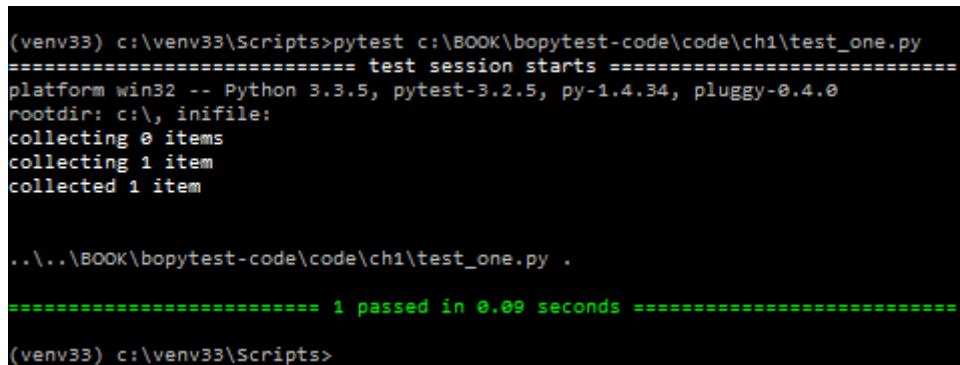
Это тест:

ch1/test_one.py

```
def test_passing():  
    assert (1, 2, 3) == (1, 2, 3)
```

Вот как это выглядит при запуске:

```
$ cd /path/to/code/ch1  
$ pytest test_one.py  
===== test session starts =====  
collected 1 items  
test_one.py .  
===== 1 passed in 0.01 seconds =====
```



```
(venv33) c:\venv33\Scripts>pytest c:\BOOK\bopytest-code\code\ch1\test_one.py  
===== test session starts =====  
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0  
rootdir: c:\, inifile:  
collecting 0 items  
collecting 1 item  
collected 1 item  
  
...\BOOK\bopytest-code\code\ch1\test_one.py .  
  
===== 1 passed in 0.09 seconds =====  
(venv33) c:\venv33\Scripts>
```

Точка после *test_one.py* означает, что один тест был запущен, и он прошел. Если вам нужна дополнительная информация, вы можете использовать `-v` или же `--verbose`:

```
$ pytest -v test_one.py  
===== test session starts =====  
collected 1 items  
test_one.py::test_passing PASSED  
===== 1 passed in 0.01 seconds =====
```

```
(venv33) c:\venv33\Scripts>pytest -v c:\BOOK\bopytest-code\code\ch1\test_one.py
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0 -- c:\venv
33\scripts\python.exe
cachedir: ..\..\cache
rootdir: c:\, inifile:
collecting 0 items
collecting 1 item
collected 1 item

..\..\BOOK\bopytest-code\code\ch1\test_one.py::test_passing PASSED

===== 1 passed in 0.04 seconds =====
(venv33) c:\venv33\Scripts>
```

Если у вас есть цветной терминал, PASSED и нижняя строка зеленые. Прекрасно!

Это неудачный тест:

ch1/test_two.py

```
def test_failing():
    assert (1, 2, 3) == (3, 2, 1)
```

То, как pytest показывает сбои при тестирование, является одной из многих причин, почему разработчики любят pytest. Давайте посмотрим, что из этого выйдет:

```
$ pytest test_two.py
===== test session starts =====
collected 1 items
test_two.py F
===== FAILURES =====
_____ test_failing _____
def test_failing():
>     assert (1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Use -v to get the full diff

test_two.py:2: AssertionError
===== 1 failed in 0.04 seconds =====
```

```
(venv33) c:\venv33\Scripts>pytest c:\BOOK\bopytest-code\code\ch1\test_two.py
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: c:\, inifile:
collecting 0 items
collecting 1 item
collected 1 item

..\..\BOOK\bopytest-code\code\ch1\test_two.py F

===== FAILURES =====
_____ test_failing _____

  def test_failing():
>     assert (1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Use -v to get the full diff

c:\BOOK\bopytest-code\code\ch1\test_two.py:2: AssertionError
===== 1 failed in 0.10 seconds =====
(venv33) c:\venv33\Scripts>
```

Отлично! Пробный тест *test_failing* получает свой раздел, чтобы показать нам, почему он не прошел.

И pytest точно сообщает, что первый сбой: index 0 — это несоответствие.

Значительная часть этого сообщения красного цвета, что делает его действительно выделяющимся (если у вас есть цветной терминал).

Это уже достаточно много информации, но есть строка с подсказкой, которая говорит Use `-v`, чтобы получить еще больше описаний различий.

Давайте заюзаем этот самый `-v`:

```
$ pytest -v test_two.py
===== test session starts =====
collected 1 items
test_two.py::test_failing FAILED
===== FAILURES =====
_____ test_failing _____

def test_failing():
>     assert (1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Full diff:
E         - (1, 2, 3)
E           ? ^     ^
E         + (3, 2, 1)
E           ? ^     ^
```

```
test_two.py:2: AssertionError
===== 1 failed in 0.04 seconds =====
```

```
(venv33) c:\venv33\Scripts>pytest -v c:\BOOK\bopytest-code\code\ch1\test_two.py
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0 -- c:\venv33\scripts\python.exe
cachedir: ..\..\cache
rootdir: c:\, inifile:
collecting 0 items
collecting 1 item
collected 1 item

..\..\BOOK\bopytest-code\code\ch1\test_two.py::test_failing FAILED

===== FAILURES =====
_____ test_failing _____

  def test_failing():
>     assert (1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Full diff:
E         - (1, 2, 3)
E         ? ^   ^
E         + (3, 2, 1)
E         ? ^   ^

c:\BOOK\bopytest-code\code\ch1\test_two.py:2: AssertionError
===== 1 failed in 0.10 seconds =====

(venv33) c:\venv33\Scripts>
```

Вот это да!

pytest добавляет символ "карет" (^), чтобы показать нам в чем именно отличие.

Если вы уже впечатлены тем, как легко писать, читать и запускать тесты с pytest и как легко читать выходные данные, чтобы увидеть, где случилась неудачка, ну... вы еще ничего не видели. Там, откуда это прилетело, чудес намного больше. Оставайтесь и позвольте мне показать вам, почему я думаю, что pytest-это абсолютно лучшая тестовая платформа.

В оставшейся части этой главы вы установите pytest, посмотрите на различные способы его запуска и выполните некоторые из наиболее часто используемых опций командной строки. В будущих главах вы узнаете, как написать тестовые функции, которые максимизируют мощность pytest, как вытащить код установки в разделы настройки и демонтажа, называемые фикстуры, и как использовать фикстуры и плагины, чтобы реально перегружать тестирование программного обеспечения.

Но сначала я должен извиниться. Извините, что тестирую `assert (1, 2, 3) == (3, 2, 1)`, это так скучно. Я слышу храп?! Никто бы не написал такой тест в реальной жизни. Тесты программного обеспечения состоят из кода, который проверяет другое программное обеспечение, которое увы не всегда будет работать положительно. А `(1, 2, 3) == (1, 2, 3)` всегда будет работать. Вот почему мы не будем использовать слишком глупые тесты, подобные этому, в остальной части книги. Мы рассмотрим тесты для реального программного проекта. Мы будем использовать пример проекта Tasks, которому требуется

тестовый код. Надеюсь, это достаточно просто, чтобы быть легко понято, но не так просто, чтобы быть скучным.

Еще одно полезное применение тестов программного обеспечения-это проверка ваших предположений о том, как работает тестируемое программное обеспечение, что может включать в себя тестирование вашего понимания сторонних модулей и пакетов и даже построение структур данных Python.

Проект Tasks использует структуру Task, основанную на фабричном методе namedtuple, который является частью стандартной библиотеки. Структура задачи используется в качестве структуры данных для передачи информации между пользовательским интерфейсом и API.

В остальной части этой главы я буду использовать Task для демонстрации запуска pytest и использования некоторых часто используемых параметров командной строки.

Вот задача:

```
from collections import namedtuple
Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
```

Before we jump into the examples, let's take a step back and talk about how to get pytest and install it.

Функция `factory namedtuple()` существует с Python 2.6, но я все еще обнаруживаю, что многие разработчики Python не знают, насколько это круто. По крайней мере, использование задачи для тестовых примеров будет интереснее, чем `(1, 2, 3) == (1, 2, 3)` или `(1, 2) == 3`.

Прежде чем перейти к примерам, давайте сделаем шаг назад и поговорим о том, где взять pytest и как установить его.

Добываем pytest

Штаб-квартира pytest <https://docs.pytest.org>. Это официальная документация. Но распространяется он через PyPI (индекс пакета Python) в <https://pypi.python.org/pypi/pytest>.

Как и другие пакеты Python, распространяемые через PyPI, используйте **pip** для установки pytest в виртуальную среду, используемую для тестирования:

```
$ pip3 install -U virtualenv
$ python3 -m virtualenv venv
$ source venv/bin/activate
$ pip install pytest
```

Если вы не знакомы с `virtualenv` или `pip`, я вас познакомлю. Ознакомьтесь с Приложением 1 «Виртуальные среды» на стр. 155 и в Приложении 2, на странице 159.

Как насчет Windows, Python 2 и venv?

Пример для `virtualenv` и `pip` должен работать на многих POSIX системах, таких как Linux и macOS, а также на многих версиях Python, включая Python 2.7.9 и более поздних.

Источник `venv/bin/activate` в строке не будет работать для Windows, используйте вместо этого `venv\Scripts\activate.bat`.

Выполните это:

```
C:\> pip3 install -U virtualenv
C:\> python3 -m virtualenv venv
C:\> venv\Scripts\activate.bat
(venv) C:\> pip install pytest
```

Для Python 3.6 и выше, вы можете обойтись `venv` вместо `virtualenv`, и вы не должны беспокоиться о том что бы установить его в первую очередь. Он включен в Python 3.6 и выше. Тем не менее, я слышал, что некоторые платформы по-прежнему ведут себя лучше с `virtualenv`.

Запускаем pytest

```
$ pytest --help
usage: pytest [options] [file_or_dir] [file_or_dir] [...]
...
```

For example, let's create a subdirectory called `tasks`, and start with this test file:

Без аргументов `pytest` исследует ваш текущий каталог и все подкаталоги для тестовых файлов и запустит тестовый код, который найдёт. Если вы передадите `pytest` имя файла, имя каталога или список из них, то будут найдены там вместо текущего каталога. Каждый

каталог, указанный в командной строке, рекурсивно исследуется для поиска тестового кода.

Для примера, давайте создадим подкаталог, называемый задачами, и начнем с этого тестового файла:

ch1/tasks/test_three.py

```
"""Проверим тип данных Task."""

from collections import namedtuple
Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
Task.__new__.__defaults__ = (None, None, False, None)

def test_defaults():
    """Использование параметров не должно вызывать значения по умолчанию."""
    t1 = Task()
    t2 = Task(None, None, False, None)
    assert t1 == t2

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

The `test_member_access()` test is to demonstrate how to access members by name and not by index, which is one of the main reasons to use namedtuples.

Let's put a couple more tests into a second file to demonstrate the `_asdict()` and `_replace()` functionality

Вы можете использовать `__new__.__defaults__` для создания объектов `Task` без указания всех полей. Тест `test_defaults()` предназначен для демонстрации и проверки того, как работают умолчания.

Тест `test_member_access()` должен продемонстрировать, как обращаться к членам по имени и не по индексу, что является одной из основных причин использования `namedtuples`.

Давайте добавим еще пару тестов во второй файл, чтобы продемонстрировать функции `_asdict()` и `_replace()`

ch1/tasks/test_four.py

```

"""Тест типа данных Task."""

from collections import namedtuple

Task = namedtuple('Task', ['summary', 'owner', 'done', 'id'])
Task.__new__.__defaults__ = (None, None, False, None)

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():
    """должно изменить переданное в fields."""
    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

```

Для запуска `pytest` у вас есть возможность указать файлы и каталоги. Если вы не укажете какие-либо файлы или каталоги, `pytest` будет искать тесты в текущем рабочем каталоге и подкаталогах. Он ищет файлы, начинающиеся с `test_` или заканчивающиеся на `_test`. Если вы запустите `pytest` из каталога `ch1`, без команд, вы проведете тесты для четырёх файлов:

```

$ cd /path/to/code/ch1
$ pytest
===== test session starts =====
collected 6 items
test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..
===== FAILURES =====
_____ test_failing _____
def test_failing():
>     assert (1, 2, 3) == (3, 2, 1)

```

```
E      assert (1, 2, 3) == (3, 2, 1)
E          At index 0 diff: 1 != 3
E          Use -v to get the full diff

test_two.py:2: AssertionError
===== 1 failed, 5 passed in 0.08 seconds =====
```

```
(venv33) c:\BOOK\bopytest-code\code\ch1>pytest
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: c:\BOOK\bopytest-code\code\ch1, inifile:
collecting 0 items
collecting 1 item
collecting 2 items
collecting 4 items
collecting 6 items
collected 6 items

test_one.py .
test_two.py F
tasks\test_four.py ..
tasks\test_three.py ..

===== FAILURES =====
_____ test_failing _____

    def test_failing():
>       assert (1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E           At index 0 diff: 1 != 3
E           Use -v to get the full diff

test_two.py:2: AssertionError
===== 1 failed, 5 passed in 0.20 seconds =====
(venv33) c:\BOOK\bopytest-code\code\ch1>
```

Чтобы выполнить только наши новые тесты задач, вы можете предоставить pytest все имена файлов, которые вы хотите запустить, или каталог, или вызвать pytest из каталога, где находятся наши тесты:

```
$ pytest tasks/test_three.py tasks/test_four.py
===== test session starts =====
collected 4 items
tasks/test_three.py ..
tasks/test_four.py ..
===== 4 passed in 0.02 seconds =====

$ pytest tasks
===== test session starts =====
collected 4 items
tasks/test_four.py ..
tasks/test_three.py ..
===== 4 passed in 0.03 seconds =====

$ cd /path/to/code/ch1/tasks
$ pytest
===== test session starts =====
collected 4 items
```

```
test_four.py ..
test_three.py ..
===== 4 passed in 0.02 seconds =====
```

```
(venv33) c:\BOOK\bopytest-code\code\ch1>pytest tasks/test_three.py tasks/test_four.py
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: c:\BOOK\bopytest-code\code\ch1, inifile:
collecting 0 items
collecting 2 items
collecting 4 items
collected 4 items

tasks\test_three.py ..
tasks\test_four.py ..

===== 4 passed in 0.07 seconds =====

(venv33) c:\BOOK\bopytest-code\code\ch1>pytest tasks
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: c:\BOOK\bopytest-code\code\ch1, inifile:
collecting 0 items
collecting 2 items
collecting 4 items
collected 4 items

tasks\test_four.py ..
tasks\test_three.py ..

===== 4 passed in 0.08 seconds =====

(venv33) c:\BOOK\bopytest-code\code\ch1>cd tasks

(venv33) c:\BOOK\bopytest-code\code\ch1\tasks>pytest
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: c:\BOOK\bopytest-code\code\ch1\tasks, inifile:
collecting 0 items
collecting 2 items
collecting 4 items
collected 4 items

test_four.py ..
test_three.py ..

===== 4 passed in 0.08 seconds =====

(venv33) c:\BOOK\bopytest-code\code\ch1\tasks>
```

Часть выполнения pytest, где pytest проходит и находит, какие тесты запускать, называется test discovery (тестовым обнаружением). pytest смог найти все те тесты, которые мы хотели запустить, потому что мы назвали их в соответствии с соглашениями об именах pytest.

Ниже приведен краткий обзор соглашений об именах, чтобы ваш тестовый код можно было обнаружить с помощью pytest:

- Тестовые файлы должны быть названы `test_<something>.py` или `<something>_test.py`.
- Методы и функции тестирования должны быть названы `test_<something>`.
- Тестовые классы должны быть названы `Test<Something>`.

Поскольку наши тестовые файлы и функции начинаются с `test_`, то у нас всё в порядке. Есть способы изменить эти правила обнаружения, если у вас есть куча тестов с разными именами.

Я расскажу об этом в главе 6, "конфигурация", на странице 113.

Давайте более подробно рассмотрим результат запуска только одного файла:

```
$ cd /path/to/code/ch1/tasks
$ pytest test_three.py
===== test session starts =====
platform darwin -- Python 3.6.2, pytest-3.2.1, py-1.4.34, pluggy-0.4.0
rootdir: /path/to/code/ch1/tasks, inifile:
collected 2 items
test_three.py ..
===== 2 passed in 0.01 seconds =====
```

Результат говорит нам совсем немного.

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

`pytest` предоставляет изящный разделитель для начала тестового сеанса. Сеанс-это один вызов `pytest`, включая все тесты, выполняемые в нескольких каталогах. Это определение сеанса становится важным, когда я говорю о области сеанса по отношению к фикстурам `pytest` в определении области фикстур, на странице 56.

Платформа `darwin` — у меня на Mac. На компьютере с ОС Windows платформа отличается. Далее перечислены версии Python и `pytest`, а также зависимости от пакетов `pytest`. И `py`, и `pluggy` — это пакеты, разработанные командой `pytest`, чтобы помочь с реализацией `pytest`.

rootdir: /path/to/code/ch1/tasks, inifile:

`rootdir` — это самый верхний общий каталог для всех каталогов в которых ищется тестовый код. В `inifile` (здесь пустой) перечислены используемые файлы конфигурации. Конфигурационными файлами могут быть `pytest.ini`, `tox.ini` или `setup.cfg`. Более подробные сведения о конфигурационных файлах вы найдете в главе 6 «Конфигурация» на стр. 113.

collected 2 items

Это две тестовые функции в файле.

test_three.py ..

`test_three.py` показывает тестируемый файл. Для каждого тестового файла есть одна строка. Две точки означают, что тесты пройдены — по одной точке для каждой тестовой функции или метода. Точки предназначены только для прохождения тестов. Failures (сбоев), errors (ошибок), skips (пропусков), xfails, и xpasses обозначаются с F, E, s, x, и X, соответственно. Если вы хотите видеть больше точек для прохождения тестов, используйте опцию `-v` или `--verbose`.

== 2 passed in 0.01 seconds ==

Эта строка относится к числу пройденных тестов и времени, затраченному на весь сеанс тестирования. При наличии непроходных тестов здесь также будет указан номер каждой категории.

Результат теста-это основной способ, благодаря которому пользователь, выполняющий тест или просматривающий результаты, может понять, что произошло в ходе выполнения теста. В `pytest` тестовые функции могут иметь несколько различных результатов, а не просто пройти или не пройти. Вот возможные результаты тестовой функции:

- PASSED (.): Тест выполнен успешно.
- FAILED (F): Тест не выполнен успешно (или XPASS + strict).
- SKIPPED (s): Тест был пропущен. Вы можете заставить `pytest` пропустить тест, используя декораторы `@pytest.mark.skip()` или `pytest.mark.skipif()`, обсуждаемые в разделе пропуск тестов, на стр. 34.
- xfail (x): Тест не должен был пройти, был запущен и провалился. Вы можете принудительно указать `pytest`, что тест должен завершиться неудачей, используя декоратор `@pytest.mark.xfail()`, описанный в маркировках тестов как ожидающий неудачу, на стр. 37.
- XPASS (X): Тест не должен был пройти, был запущен и прошел!..
- ERROR (E): Исключение произошло за пределами функции тестирования, либо в фикстуре, обсуждается в главе 3, `pytest` Фикстуры, на стр. 49, или в `hook function`, обсуждается в главе 5, Плагины, на странице 95.

Выполнение Только Одного Теста

Пожалуй, первое, что вы захотите сделать, после того, как начали писать тесты, — это запустить только один. Укажите файл напрямую и добавьте имя `::test_name:`

```
$ cd /path/to/code/ch1
$ pytest -v tasks/test_four.py::test_asdict
===== test session starts =====
collected 3 items
tasks/test_four.py::test_asdict PASSED
===== 1 passed in 0.01 seconds =====
```

Теперь давайте рассмотрим некоторые варианты.

Использование Опций

Мы уже пару раз использовали опцию `verbose`, `-v` или `--verbose`, но есть еще много опций, о которых стоит знать. Мы не собираемся использовать их все в этой книге, только некоторые. Ознакомиться с полным списком вы можете с помощью опции `pytest --help`.

Ниже приведены несколько вариантов, которые весьма полезны при работе с `pytest`. Это далеко не полный список, но этих опций для начала вполне достаточно.

```
$ pytest --help

usage: pytest [options] [file_or_dir] [file_or_dir] [...]

... subset of the list ...

positional arguments:
  file_or_dir

general:
  -k EXPRESSION          only run tests which match the given substring
                        expression. An expression is a python evaluable
                        expression where all names are substring-matched
                        against test names and their parent classes. Example:
                        -k 'test_method or test_other' matches all test
                        functions and classes whose name contains
                        'test_method' or 'test_other', while -k 'not
                        test_method' matches those that don't contain
                        'test_method' in their names. Additionally keywords
                        are matched to classes and functions containing extra
                        names in their 'extra_keyword_matches' set, as well as
                        functions which have names assigned directly to them.
```

```
-m MARKEXPRESS    only run tests matching given mark expression.
                   example: -m 'mark1 and not mark2'.

--markers          show markers (builtin, plugin and per-project ones).
-x, --exitfirst    exit instantly on first error or failed test.
--maxfail=num      exit after first num failures or errors.

...

--capture=method   per-test capturing method: one of fd|sys|no.
-s                shortcut for --capture=no.

...

--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

...

reporting:
-v, --verbose      increase verbosity.
-q, --quiet        decrease verbosity.
--verbosity=VERBOSE set verbosity

...

-l, --showlocals   show locals in tracebacks (disabled by default).
--tb=style         traceback print mode (auto/long/short/line/native/no).

...

--durations=N      show N slowest setup/test durations (N=0 for all).

...

collection:
--collect-only      only collect tests, don't execute them.

...

test session debugging and configuration:
--basetemp=dir      base temporary directory for this test run. (warning:
                   this directory is removed if it exists)
--version           display pytest lib version and import information.
-h, --help          show help message and configuration info
```


--collect-only

Параметр `--collect-only` показывает, какие тесты будут выполняться с заданными параметрами и конфигурацией. Этот параметр удобно сначала показать, чтобы выходные данные можно было использовать в качестве ссылки для остальных примеров. Если вы начинаете в каталоге `ch1`, вы должны увидеть все тестовые функции, которые вы смотрели до сих пор в этой главе:

```
$ cd /path/to/code/ch1
$ pytest --collect-only

===== test session starts =====
collected 6 items
<Module 'test_one.py'>
  <Function 'test_passing'>
<Module 'test_two.py'>
  <Function 'test_failing'>
<Module 'tasks/test_four.py'>
  <Function 'test_asdict'>
  <Function 'test_replace'>
<Module 'tasks/test_three.py'>
  <Function 'test_defaults'>
  <Function 'test_member_access'>

===== no tests ran in 0.03 seconds =====
```

Параметр `--collect-only` полезен для проверки правильности выбора других опций, которые выбирают тесты перед запуском тестов. Мы будем использовать его снова с `-k`, чтобы показать, как это работает.

-k EXPRESSION

Параметр `-k` позволяет использовать выражение для определения функций тестирования.

Весьма мощная опция! Её можно использовать как ярлык для запуска отдельного теста, если имя уникально, или запустить набора тестов, которые имеют общий префикс или суффикс в их именах. Предположим, вы хотите запустить тесты `test_asdict()` и `test_defaults()`. Вы можете проверить фильтр с помощью: `--collect-only:`

```
$ cd /path/to/code/ch1
$ pytest -k "asdict or defaults" --collect-only

===== test session starts =====
collected 6 items
<Module 'tasks/test_four.py'>
  <Function 'test_asdict'>
<Module 'tasks/test_three.py'>
  <Function 'test_defaults'>

===== 4 tests deselected =====
===== 4 deselected in 0.03 seconds =====
```

Ага! Это похоже на то, что нам надо. Теперь вы можете запустить их, удалив `--collect-only`:

```
$ pytest -k "asdict or defaults"
===== test session starts =====
collected 6 items
tasks/test_four.py .
tasks/test_three.py .

===== 4 tests deselected =====
===== 2 passed, 4 deselected in 0.03 seconds =====
```

Упс! Просто точки. Так значит они прошли. Но были ли они правильными тестами? Один из способов узнать — использовать `-v` или `--verbose`:

```
$ pytest -v -k "asdict or defaults"

===== test session starts =====
collected 6 items
tasks/test_four.py::test_asdict PASSED
tasks/test_three.py::test_defaults PASSED

===== 4 tests deselected =====
===== 2 passed, 4 deselected in 0.02 seconds =====
```

Ага! Это были правильные тесты.

-m MARKEXPR

Маркеры-один из лучших способов пометить подмножество тестовых функций для совместного запуска. В качестве примера, один из способов запустить `test_replace()` и `test_member_access()`, даже если они находятся в отдельных файлах, это пометить их. Можно использовать любое имя маркера. Допустим, вы хотите использовать `run_these_please`. Отметим тесты, используя декоратор `@pytest.mark.run_these_please`, вот так:

```
import pytest

...
@pytest.mark.run_these_please
def test_member_access():
    ...
```

Теперь то же самое для `test_replace()`. Затем вы можете запустить все тесты с тем же маркером с помощью `pytest -m run_these_please`:

```
$ cd /path/to/code/ch1/tasks
$ pytest -v -m run_these_please

===== test session starts =====
collected 4 items
test_four.py::test_replace PASSED
test_three.py::test_member_access PASSED

===== 2 tests deselected =====
===== 2 passed, 2 deselected in 0.02 seconds =====
```

Выражение маркера не обязательно должно быть одним маркером. Вы можете использовать такие варианты, как `-m "mark1 and mark2"` для тестов с обоими маркерами, `-m "mark1 and not mark2"` для тестов, которые имеют метку 1, но не метку 2, `-m "mark1 or mark2"` для тестов с одним из и т. д., Я более подробно обсужу маркеры в Методах проверки маркировки, на стр. 31.

-x, --exitfirst

Нормальным поведением *pytest* является запустить все тесты, которые он найдет. Если тестовая функция обнаружит сбой *assert* или *exception*, выполнение этого теста останавливается, и тест завершается ошибкой. И тогда *pytest* запускает следующий тест. По большей части, это то что надо. Однако, особенно при отладке проблемы, мешает

сразу всей тестовой сессии, когда тест не является правильным. Вот что делает `-x` опция. Давайте попробуем это на шести тестах, которые у нас есть в настоящий момент:

```
$ cd /path/to/code/ch1
$ pytest -x

===== test session starts =====
collected 6 items
test_one.py .
test_two.py F

===== FAILURES =====
_____ test_failing _____
      def test_failing():
>         assert (1, 2, 3) == (3, 2, 1)
E         assert (1, 2, 3) == (3, 2, 1)
E             At index 0 diff: 1 != 3
E             Use -v to get the full diff

test_two.py:2: AssertionError
===== 1 failed, 1 passed in 0.38 seconds =====
```

В верхней части выходных данных вы видите, что все шесть тестов (или “элементов (items)”) были собраны, а в нижней строке вы видите, что один тест не прошел и один прошел, и *pytest* отобразил строку “прервано (Interrupted)”, чтобы сообщить нам, что он остановлен. Без `-x` все шесть тестов были бы запущены. Давайте повторим еще раз без `-x`. А также используем `--tb=no`, чтобы отключить трассировку стека, так как вы ее уже видели и не обязательно видеть её снова:

```
$ cd /path/to/code/ch1
$ pytest --tb=no

===== test session starts =====
collected 6 items

test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..

===== 1 failed, 5 passed in 0.09 seconds =====
```

Этот пример демонстрирует, что без `-x`, `pytest` отмечает сбой в `test_two.py` и продолжает дальнейшее тестирование.

`--maxfail=num`

Параметр `-x` приводит к остановке после первого отказа теста. Если вы хотите, чтобы некоторые число сбоев было допущено, но не целая тонна, используйте параметр `--maxfail`, чтобы указать, сколько ошибок допускается получить. Пока трудно показать это только с одним неудачным тестом в нашей системе, но давайте посмотрим в любом случае. Поскольку есть только один сбой, если мы установили `--maxfail = 2`, все тесты должны выполняться, а `--maxfail = 1` должен действовать так же, как `-x`:

```
$ cd /path/to/code/ch1
$ pytest --maxfail=2 --tb=no

===== test session starts =====
collected 6 items
test_one.py .
test_two.py F
tasks/test_four.py ..
tasks/test_three.py ..

===== 1 failed, 5 passed in 0.08 seconds =====

$ pytest --maxfail=1 --tb=no

===== test session starts =====
collected 6 items
test_one.py .
test_two.py F
!!!!!!!!!! Interrupted: stopping after 1 failures !!!!!!!!!!!

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

Еще раз мы использовали `--tb=no`, чтобы отключить трассировку.

`-s` and `--capture=method`

Флаг `-s` позволяет печатать операторы — или любой другой вывод, который обычно печатается в `stdout`, чтобы фактически быть напечатанным в стандартном выводе во время выполнения тестов. Это сокращенный вариант для `--capture=no`. Смысл в том, что обычно выходные данные захватываются во всех тестах. Неудачные тесты будут выводиться после того, как тест будет протекать в предположении, что выход поможет вам

понять, что что то пошло не так. Параметр `-s` или `--capture=no` отключает захват выходных данных. При разработке тестов я обычно добавляю несколько операторов `print()`, чтобы можно было следить за ходом теста.

Другой вариант, который может помочь вам обойтись без операторов печати в вашем коде, `-l/--showlocals`, который распечатывает локальные переменные в тесте, если тест терпит неудачу.

Другие опции метода захвата `--capture=fd` и `--capture=sys`. — Опция `--capture=sys` заменяет `sys.stdout/stderr` в тем-файлах. Опция `--capture=fd` указывает файловые дескрипторы 1 и 2 на временный файл.

Я включаю описания `sys` и `fd` для полноты. Но, честно говоря, я никогда не нуждался и не использовал их. Я часто использую `-s`. И чтобы полностью описать, как работает `-s`, мне нужно было коснуться методов захвата.

У нас пока нет никаких операторов печати в наших тестах; демонстрация была бы бессмысленной. Тем не менее, я предлагаю вам немного поиграть с ними, чтобы вы увидели это в действии.

-lf, --last-failed

При сбое одного или нескольких тестов способ выполнения только неудачных тестов полезен для отладки. Просто используйте `--lf` и вы готовы к отладке:

Это удобно, если вы используете опцию `--tb`, которая скрывает некоторую информацию, и вы хотите повторить сбой с другой опцией отслеживания.

```
$ cd /path/to/code/ch1
$ pytest --lf

===== test session starts =====
run-last-failure: rerun last 1 failures
collected 6 items
test_two.py F

===== FAILURES =====
_____ test_failing _____
      def test_failing():
>         assert (1, 2, 3) == (3, 2, 1)
E         assert (1, 2, 3) == (3, 2, 1)
E             At index 0 diff: 1 != 3
E             Use -v to get the full diff
```

```
test_two.py:2: AssertionError
===== 5 tests deselected =====
===== 1 failed, 5 deselected in 0.08 seconds =====
```

-ff, --failed-first

Параметр `--ff/--failed-first` будет делать то же самое, что и `--last-failed`, а затем выполнять остальные тесты, прошедшие в прошлый раз:

```
$ cd /path/to/code/ch1
$ pytest --ff --tb=no

===== test session starts =====
run-last-failure: rerun last 1 failures first

collected 6 items

test_two.py F
test_one.py .
tasks/test_four.py ..
tasks/test_three.py ..

===== 1 failed, 5 passed in 0.09 seconds =====
```

Обычно `test_failing()` из `test_two.py` запускается после `test_one.py`. Однако, поскольку `test_failing()` не удалось в последний раз, `--ff` заставляет его запускаться в первую очередь

-v, --verbose

Опция `-v/--verbose` предоставляет более развернутую информацию по итогам. Наиболее очевидным отличием является то, что каждый тест получает свою собственную строку, а имя теста и результат прописываются вместо точки.

Мы уже использовали его пару раз, но давайте запустим его снова для удовольствия в сочетании с `--ff` и `--tb=no`:

```
$ cd /path/to/code/ch1
$ pytest -v --ff --tb=no

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

```

run-last-failure: rerun last 1 failures first
collected 6 items

test_two.py::test_failing FAILED
test_one.py::test_passing PASSED
tasks/test_four.py::test_asdict PASSED
tasks/test_four.py::test_replace PASSED
tasks/test_three.py::test_defaults PASSED
tasks/test_three.py::test_member_access PASSED

===== 1 failed, 5 passed in 0.07 seconds =====

```

```

(venv33) C:\BOOK\bopytest-code\code\ch1>pytest -v --ff --tb=no
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0 -- c:\venv33\scripts\python.exe
cachedir: .cache
rootdir: C:\BOOK\bopytest-code\code\ch1, inifile:
collected 6 items
run-last-failure: rerun previous 1 failure first

test_two.py::test_failing FAILED
test_one.py::test_passing PASSED
tasks/test_four.py::test_asdict PASSED
tasks/test_four.py::test_replace PASSED
tasks/test_three.py::test_defaults PASSED
tasks/test_three.py::test_member_access PASSED

===== 1 failed, 5 passed in 0.28 seconds =====

```

На цветном терминале в отчете вы также увидите красный результат FAILED и зеленые PASSED.

-q, --quiet

Опция -q/--quiet противоположна -v/--verbose; она сокращает объем информации в отчете. Мне нравится использовать его в сочетании с --tb=line, в этом случае выводятся только неудавшиеся строки любых неудачных тестов.

Попробуем -q самостоятельно:

```

$ cd /path/to/code/ch1
$ pytest -q
.F....
===== FAILURES =====
_____ test_failing _____
      def test_failing():
>         assert (1, 2, 3) == (3, 2, 1)
E         assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3

```



```

E             Full diff:
E             - (1, 2, 3)
E             ? ^      ^
E             + (3, 2, 1)
E             ? ^      ^

test_two.py:2: AssertionError
1 failed, 5 passed in 0.08 seconds

```

Опция `-q` делает вывод довольно кратким, но обычно этого достаточно. Мы будем часто использовать опцию `-q` в остальной части книги (а также `--tb=no`), чтобы ограничить вывод тем, что мы конкретно пытаемся понять в то время.

`-l, --showlocals`

При использовании параметра `-l/--showlocals` локальные переменные и их значения отображаются вместе с `tracebacks` для неудачных тестов.

До сих пор у нас не было неудачных тестов с локальными переменными. Если я возьму тест `test_replace()` и изменю

```
t_expected = Task('finish book', 'brian', True, 10)
```

на

```
t_expected = Task('finish book', 'brian', True, 11)
```

10 и 11 должны вызвать сбой. Любое изменение ожидаемого значения приведет к сбою. Но этого достаточно для демонстрации опции командной строки `--l/--showlocals`:

```

$ cd /path/to/code/ch1
$ pytest -l tasks

===== test session starts =====
collected 4 items
tasks/test_four.py .F
tasks/test_three.py ..

===== FAILURES =====
_____ test_replace _____

```

```

@pytest.mark.run_these_please
def test_replace():
    """replace() should change passed in fields."""
    t_before = Task('finish book', 'brian', False)
    t_after = t_before._replace(id=10, done=True)
    t_expected = Task('finish book', 'brian', True, 11)
> assert t_after == t_expected
E     AssertionError: assert Task(summary=...e=True, id=10) == Task(summary='
E         At index 3 diff: 10 != 11
E         Use -v to get the full diff

t_after      = Task(summary='finish book', owner='brian', done=True, id=10)
t_before     = Task(summary='finish book', owner='brian', done=False, id=None)
t_expected   = Task(summary='finish book', owner='brian', done=True, id=11)

tasks\test_four.py:28: AssertionError
===== 1 failed, 3 passed in 0.08 seconds =====

```

```

(venv33) C:\BOOK\bopytest-code\code\chl>pytest -l tasks
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: C:\BOOK\bopytest-code\code\chl, inifile:
collected 4 items

tasks\test_four.py .F
tasks\test_three.py ..

===== FAILURES =====
_____ test_replace _____

@pytest.mark.run_these_please
def test_replace():
    """replace() should change passed in fields."""
    t_before = Task('finish book', 'brian', False)
    t_after = t_before._replace(id=10, done=True)
    t_expected = Task('finish book', 'brian', True, 11)
> assert t_after == t_expected
E     AssertionError: assert Task(summary=...e=True, id=10) == Task(summary='...e=True, id=11)
E         At index 3 diff: 10 != 11
E         Use -v to get the full diff

t_after      = Task(summary='finish book', owner='brian', done=True, id=10)
t_before     = Task(summary='finish book', owner='brian', done=False, id=None)
t_expected   = Task(summary='finish book', owner='brian', done=True, id=11)

tasks\test_four.py:28: AssertionError
===== 1 failed, 3 passed in 0.20 seconds =====

(venv33) C:\BOOK\bopytest-code\code\chl>

```

Локальные переменные `t_after`, `t_before` и `t_expected` отображаются после фрагмента кода со значением, которое они содержали во время неудавшегося `assert`-а.

--tb=style

Параметр `--tb=style` изменяет способ вывода пакетов трассировки для сбоев. При сбое теста `pytest` отображает список сбоев и так называемую обратную трассировку, которая показывает точную строку, в которой произошел сбой. Хотя *tracebacks* полезны большую часть времени, бывают случаи, когда они раздражают. Вот где опция `--tb=style` пригодится. Стили, которые я считаю полезными, это `short`, `line` и `no`. `short` печатает только строку *assert* и символ **E** без контекста; `line` сохраняет ошибку в одной строке; `no` полностью удаляет трассировку.

Давайте оставим пока модификацию `test_replace()`, чтобы она завершилась неудачей, и запустим ее с разными стилями трассировки. `--tb=no` полностью удаляет трассировку

```
$ cd /path/to/code/ch1
$ pytest --tb=no tasks
===== test session starts =====
collected 4 items
tasks/test_four.py .F
tasks/test_three.py ..
===== 1 failed, 3 passed in 0.04 seconds =====
```

`--tb=line` in many cases is enough to tell what's wrong. If you have a ton of failing tests, this option can help to show a pattern in the failures:

`--tb=line` во многих случаях достаточно, чтобы показать, что не так. Если у вас гора неудачных тестов, этот параметр может помочь отобразить шаблон в сбоях:

```
$ pytest --tb=line tasks
===== test session starts =====
collected 4 items
tasks/test_four.py .F
tasks/test_three.py ..
===== FAILURES =====
/path/to/code/ch1/tasks/test_four.py:20:
AssertionError: assert Task(summary=...e=True, id=10) == Task(
summary='...e=True, id=11)
===== 1 failed, 3 passed in 0.05 seconds =====
```

Следующий шаг в verbose tracebacks `--tb=short`:

```
$ pytest --tb=short tasks
===== test session starts =====
collected 4 items
tasks/test_four.py .F
tasks/test_three.py ..
===== FAILURES =====
_____ test_replace _____
tasks/test_four.py:20: in test_replace
assert t_after == t_expected
E AssertionError: assert Task(summary=...e=True, id=10) == Task(
summary='...e=True, id=11)
E At index 3 diff: 10 != 11
E Use -v to get the full diff
===== 1 failed, 3 passed in 0.04 seconds =====
```

Это определенно достаточно, чтобы рассказать вам, что происходит.

Есть три оставшихся варианта трассировки, которые мы пока не рассмотрели.

`pytest --tb=long` покажет вам наиболее исчерпывающий и информативный `traceback`.
`pytest --tb=auto` покажет вам длинную версию для первого и последнего *tracebacks*, если у вас есть несколько сбоев. Это поведение по умолчанию. `pytest --tb=native` покажет вам стандартную библиотеку *traceback* без дополнительной информации.

--durations=N

Опция `--durations=N` невероятно полезна, когда вы пытаетесь ускорить свой набор тестов. Она не меняет ваши тесты; сообщает самый медленный N номер `tests/setups/teardowns` по окончании тестов. Если вы передадите `--durations=0`, он сообщит обо всем в порядке от самого медленного к самому быстрому.

Поскольку наши тесты не достаточно длинные, я добавлю `time.sleep(0.1)` к одному из тестов. Угадайте, какой:

```
$ cd /path/to/code/ch1
$ pytest --durations=3 tasks
===== test session starts =====
collected 4 items
tasks/test_four.py ..
tasks/test_three.py ..
===== slowest 3 test durations =====
0.10s call tasks/test_four.py::test_replace
```

```
0.00s setup tasks/test_three.py::test_defaults
0.00s teardown tasks/test_three.py::test_member_access
===== 4 passed in 0.13 seconds
```

Медленный тест с дополнительным *sleep* появляется сразу же после вызова метки, за которым следует установка и опровержение. Каждый тест по существу состоит из трех этапов: *call*(вызов), *настройки*(*setup*) и *опровержения*(*teardown*). Установка и опровержение также являются фикстурой, и вы можете добавить код для получения данных или тестируемой системы программного обеспечения в состояние предварительного условия до запуска теста, а также, при необходимости, очистить их. Я подробно освещаю фикстуры в главе 3, *pytest Fixtures*, на стр. 49.

--version

Опция `--version` показывает версию *pytest* и каталог, в котором он установлен:

```
$ pytest --version
This is pytest version 3.0.7, imported from
/path/to/venv/lib/python3.5/site-packages/pytest.py
```

Поскольку мы установили *pytest* в виртуальную среду, *pytest* будет находиться в каталоге *site-packages* этой виртуальной среды.

-h, --help

Опция `-h/--help` весьма полезна, даже после того, как вы привыкнете к *pytest*. Она не только показывает вам, как использовать *stock-овый* *pytest*, но также расширяется по мере установки плагинов, чтобы показать параметры и переменные конфигурации, добавленные плагинами.

Опция `-h` показывает:

- Использование: `pytest [опции] [file_or_dir] [file_or_dir] [...]`
- Параметры командной строки и краткое описание, включая добавленные параметры через плагины
- Список опций, доступных для *ini* файлов конфигурации стиля, которые я буду обсуждать подробнее в главе 6, *Конфигурация*, на стр. 113
- Список переменных среды, которые могут влиять на поведение *pytest* (также обсуждается в главе 6, *Конфигурация*, на стр. 113)

- Напоминание о том, что `pytest --markers` можно использовать для просмотра доступных маркеров, обсуждаемых в главе 2, Написание тестовых функций, на стр. 23
- Напоминание о том, что `pytest --fixtures` могут быть использованы для просмотра доступных фикстур, обсуждаемых в главе 3, Фикстуры pytest, на стр. 49

Последний часть информации текста справки отображает это примечание:

```
(shown according to specified file_or_dir or current dir if not specified)
```

Это примечание важно, поскольку параметры, маркеры и фикстуры могут изменяться в зависимости от каталога или тестового файла. Это происходит потому, что по пути к указанному файлу или каталогу `pytest` может найти файлы `conftest.py`, которые могут включать функции-ловушки (hook functions), создающие новые параметры, определения фикстур и определения маркеров.

Возможность настраивать поведение `pytest` в файлах `conftest.py` и тестовых файлах позволяет настраивать поведение локально для проекта или даже подмножество тестов для проекта. Вы узнаете о файлах `conftest.py` и `ini`, таких как `pytest.ini` в главе 6 «Конфигурация», на странице 113.

Упражнения

1. Создайте новую виртуальную среду, используя `python -m virtualenv` или `python -m venv`. Даже если вы знаете, что вам не нужны виртуальные среды для проекта, над которым вы работаете, пораруйте меня и достаточно узнайте о них, чтобы создать виртуалку, дабы опробовать вещи из этой книги. Я сопротивлялся их использованию очень долго, и теперь я всегда использую их. Прочтите Приложение 1, Виртуальная среда, на стр. 155, если у вас возникли трудности.
2. Практикуйте активацию и деактивацию виртуальной среды несколько раз.

```
- $ source venv/bin/activate  
- $ deactivate
```

On Windows:

```
- C:\Users\okken\sandbox>venv\scripts\activate.bat  
- C:\Users\okken\sandbox>deactivate
```

3. Установите pytest в новой виртуальной среде. См. Приложение 2, `pip`, на странице 159, если у вас возникли проблемы. Даже если вы думали, что у вас уже установлен pytest, вам нужно установить его в виртуальную среду, которую вы только что создали.
4. Создайте несколько тестовых файлов. Вы можете использовать те, которые мы использовали в этой главе или сделать свои собственные. Попрактикуйте pytest на этих файлах.
5. Измените операторы `assert`. Не просто используйте `assert something == something_else`; попробуйте такие вещи, как:
 - `assert 1 in [2, 3, 4]`
 - `assert a < b`
 - `assert 'fizz' not in 'fizzbuzz'`

Что дальше

В этой главе мы рассмотрели, где получить pytest и различные способы его запуска. Однако мы не обсуждали, что входит в тестовые функции. В следующей главе мы рассмотрим написание тестовых функций, параметризацию их так, чтобы они вызывались с различными данными, и группировку тестов в классы, модули и пакеты.



+13

1,7k

41



29

Карма

149,7

Рейтинг

Александр Драгункин @AlekSandrDr

Пользователь

90 подписчиков

Поделиться публикацией



 Комментировать

ПОХОЖИЕ ПУБЛИКАЦИИ

4 сентября 2017

Тестируем асинхронный код с помощью PyTest (перевод)

29 октября 2015

PyTest

10 ноября 2014

Как в Яндексе используют PyTest и другие фреймворки для функционального тестирования

ПОПУЛЯРНОЕ ЗА СУТКИ

вчера в 20:22

Как я хакера ловил

вчера в 09:39

Из жизни с Kubernetes: Как HTTP-сервер испанцев не жаловал

вчера в 12:00

Дроны и роботы, помогавшие спасти парижский собор Нотр-Дам

вчера в 11:09

Как консольные войны стимулировали прогресс консолей и видеоигр

вчера в 11:03

Кто есть кто в open source: биографии гиков



Настройка языка

Полная версия

© 2006–2019 «ТМ»

