pytest

unknown

июн. 03, 2021

Оглавление

Глава 1

Установка и запуск

Pythons: Python 3.5, 3.6, 3.7, PyPy3

Платформы: Linux и Windows

Имя пакета PyPI: pytest

Документация в PDF: download latest

`pytest` - это фреймворк, который позволяет легко создавать как простые, так и расширяемые тесты. Тесты выразительны и легко читаются — не нужно никаких шаблонов. Начните работу в считанные минуты с небольшого модульного или сложного функционального теста для вашего приложения или библиотеки.

1.1 Установка `pytest`

1. Выполните в командной строке:

```
pip install -U pytest
```

2. Убедитесь, что установили нужную версию:

```
$ pytest --version
This is pytest version 5.x.y, imported from $PYTHON_PREFIX/lib/python3.6/site-packages/pytest/__
↪init__.py
```

1.2 Создание первого теста

Создайте простой тест всего лишь из четырех строк кода:

```
# content of test_sample.py
def func(x):
    return x + 1


def test_answer():
    assert func(3) == 5
```

Готово. Теперь можно запустить тест:

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_sample.py F                                                  [100%]

================================ FAILURES ================================
_____________________________ test_answer _____________________________

    def test_answer():
>       assert func(3) == 5
E       assert 4 == 5
E        +  where 4 = func(3)

test_sample.py:6: AssertionError
========================== 1 failed in 0.12s ===========================
```

Поскольку `func(3)` возвращает вовсе не `5`, тест вернул отчет о падении.

Примечание: Для проверки ожидаемого результата можно использовать простой оператор `assert`. Встроенный в `pytest` подробный анализ assert распишет вам промежуточные значения выражений `assert`, так что вам не нужно использовать множество имен унаследованных от JUnit методов.

1.3 Запуск множества тестов

Команда `pytest` запускает все файлы с именами в формате `test_*.py` или `\*_test.py` в текущей папке и ее подпапках. В целом, тесты ищутся по *стандартным правилам поиска тестов*.

1.4 Проверка брошенного исключения

Используйте *проверку ожидаемых исключений*, чтобы убедиться, что код бросает ожидаемое исключение:

```
# content of test_sysexit.py
import pytest


def f():
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    raise SystemExit(1)


def test_mytest():
    with pytest.raises(SystemExit):
        f()
```

Запустите этот тест с опцией “quiet” (`-q` - указывает, что нужно опустить имя файла в отчете):

```
$ pytest -q test_sysexit.py
.                                                                    [100%]
1 passed in 0.12s
```

1.5 Группировка тестовых функций в класс

Если тестов много, вы можете захотеть создать тестовый класс. `pytest` облегчает создание классов с множеством тестов:

```
# content of test_class.py
class TestClass:
    def test_one(self):
        x = "this"
        assert "h" in x

    def test_two(self):
        x = "hello"
        assert hasattr(x, "check")
```

`pytest` ищет тесты для запуска по *правилам Python по поиску тестов*, так что найдет обе функции с префиксом `test_`. Никаких подклассов создавать не нужно, просто убедитесь, что имя вашего класса начинается с `Test`, иначе класс будет пропущен. Мы можем запустить именно этот модуль, просто указав его имя:

```
$ pytest -q test_class.py
.F                                                                   [100%]
================================= FAILURES =================================
____________________________ TestClass.test_two ____________________________

self = <test_class.TestClass object at 0xdeadbeef>

    def test_two(self):
        x = "hello"
>       assert hasattr(x, "check")
E       AssertionError: assert False
E        +  where False = hasattr('hello', 'check')

test_class.py:8: AssertionError
1 failed, 1 passed in 0.12s
```

Первый тест пройдет (`pass`), а второй упадет (`fail`). Поскольку `pytest` выводит промежуточные значения `assert`, вам будет проще понять причину сбоя.

1.6 Запрос временного каталога для функциональных тестов

pytest представляет встроенные фикстуры для запроса произвольных ресурсов, например, для создания уникального каталога временных файлов:

```
# content of test_tmpdir.py
def test_needsfiles(tmpdir):
    print(tmpdir)
    assert 0
```

Укажите tmpdir в описании тестовой функции, и pytest найдет и запустит соответствующую фикстуру для создания нужного ресурса до вызова самой функции. В данном случае перед запуском теста pytest создаст уникальный (для каждого вызова тестовой фунции) каталог временных файлов:

```
$ pytest -q test_tmpdir.py
F                                                                    [100%]
================================ FAILURES ================================
_____________________________ test_needsfiles ______________________________

tmpdir = local('PYTEST_TMPDIR/test_needsfiles0')

    def test_needsfiles(tmpdir):
        print(tmpdir)
>       assert 0
E       assert 0

test_tmpdir.py:3: AssertionError
--------------------------- Captured stdout call ---------------------------
PYTEST_TMPDIR/test_needsfiles0
1 failed in 0.12s
```

Подробнее об обработке параметра tmpdir см.: Временные каталоги и файлы.

Получить список встроенных в pytest *фикстур* можно, выполнив команду:

```
pytest --fixtures   # выводит список встроенных и пользовательских фикстур
```

Обратите внимание, что если не добавить опцию -v , фикстуры с префиксом _ не попадут в список.

1.7 Читайте дальше

Посмотрите дополнительные ресурсы по pytest, которые помогут вам настроить тесты для вашего уникального рабочего процесса:

- «*Вызов pytest с помощью python -m pytest*» - примеры вызова из командной строки
- «*Использование pytest с существующими наборами тестов*» - работа с уже существующими тестами
- «*Маркировка тестов*» - маркировка тестов (pytest.mark)
- «*Фикстуры pytest: явные, модальные, расширяемые*» - обеспечение функциональной основы тестов
- «plugins» - написание плагинов и управление ими
- «*Рекомендации по интеграции*» - виртуальное окружение и оформление тестов

Глава 2

Использование и вызов

2.1 Вызов pytest с помощью python -m pytest

Тестирование можно запустить из командной строки интерпретатора `Python`, используя команду:

```
python -m pytest [...]
```

В отличие от запуска напрямую командой `pytest [...]`, запуск через `Python` добавит текущий каталог в `sys.path`.

2.2 Статусы завершения

Выполнение `pytest` может генерировать один из следующих статусов завершения:

Exit code 0 Все тесты были собраны и успешно прошли

Exit code 1 Тесты были собраны и запущены, но некоторые из них упали

Exit code 2 Выполнение тестов было прервано пользователем

Exit code 3 Во время выполнения тестов произошла внутренняя ошибка

Exit code 4 Ошибка запуска `pytest` из командной строки

Exit code 5 Не удалось собрать тесты (тесты не найдены)

Коллекция статусов представлена перечислением _pytest.config.ExitCode. Статусы завершения являются частью публичного API, их можно импортировать и использовать непосредственно:

```
from pytest import ExitCode
```

Примечание: Для настройки кода завершения сценария, особенно когда тесты не удалось собрать, можно использовать плагин pytest-custom_exit_code.

2.3 Получение помощи по версии, параметрам, переменным окружения

```
pytest --version   # показывает версию и место, откуда импортирован ``pytest``
pytest --fixtures  # показывает доступные встроенные функции
pytest -h | --help # показывает помощь по командной строке и параметры конфигруационного файла
```

2.4 Остановка после первых N падений

Чтобы остановить процесс тестирования после первых N падений, используются параметры:

```
pytest -x           # остановка после первого упавшего теста
pytest --maxfail=2  # остановка после первых двух упавших тестов
```

2.5 Выбор выполняемых тестов

`pytest` поддерживает несколько способов выбора и запуска тестов из командной строки.

Запуск тестов модуля

```
pytest test_mod.py
```

Запуск тестов из директории

```
pytest testing/
```

Запуск тестов, удовлетворяющих ключевому выражению

```
pytest -k "MyClass and not method"
```

Эта команда запустит тесты, имена которых удовлетворяют заданному строковому выражению (без учета регистра). Строковые выражения могут включать операторы `Python`, которые используют имена файлов, классов и функций в качестве переменных. В приведенном выше примере будет запущен тест `MyClass.test_something`, но не будет запущен тест `TestMyClass.test_method_simple`.

Запуск тестов по идентификаторам узлов

Каждому собранному тесту присваивается уникальный идентификатор `nodeid`, который состоит из имени файла модуля, за которым следуют спецификаторы, такие как имена классов, имена функций и параметры из параметризации, разделенные символами `::`:

Чтобы запустить конкретный тест из модуля, выполните:

```
pytest test_mod.py::test_func
```

Еще один пример спецификации тестового метода в командной строке:

```
pytest test_mod.py::TestClass::test_method
```

Запуск маркированных тестов

```
pytest -m slow
```

Будут запущены тесты, помеченные декоратором `@pytest.mark.slow`.

Подробнее см. *marks*.

Запуск тестов из пакетов

```
pytest --pyargs pkg.testing
```

Будет импортирован пакет `pkg.testing`, и его расположение в файловой системе будет использовано для поиска и запуска тестов.

2.6 Изменение вывода сообщений трассировки

Примеры вывода:

```
pytest --showlocals # показывать локальные переменные в сообщениях
pytest -l           # показывать локальные переменные в сообщениях (краткий вариант)
pytest --tb=auto    # (по умолчанию) "расширенный" вывод для первого и
                    # последнего сообщений, и "короткий" для остальных
pytest --tb=long    # исчерпывающий, подробный формат сообщений
pytest --tb=short   # сокращенный формат сообщений
pytest --tb=line    # только одна строка на падение
pytest --tb=native  # стандартный формат библиотеки Python
pytest --tb=no      # никаких сообщений
```

Использование `--full-trace` приводит к тому, что при ошибке печатаются очень длинные трассировки (длиннее, чем при `--tb=long`). Параметр также гарантирует, что сообщения трассировки будут напечатаны при **прерывании выполнения с клавиатуры** с помощью Ctrl+C. Это очень полезно, если тесты занимают слишком много времени, и вы прерываете их с клавиатуры с помощью Ctrl+C, чтобы узнать, где они зависли. По умолчанию при прерывании вывод не будет показан (поскольку исключение `KeyboardInterrupt` будет поймано `pytest`). Используя этот параметр, вы можете быть уверены, что увидите трассировку.

2.7 Детализация сводного отчета

Флаг `-r` можно использовать для отображения «краткой сводной информации по тестированию» в конце тестового сеанса, что упрощает получение четкой картины всех сбоев, пропусков, xfails и т. д.

По умолчанию для списка сбоев и ошибок используется добавочная комбинация `fE`.

Пример:

```
# content of test_example.py
import pytest


@pytest.fixture
def error_fixture():
    assert 0
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def test_ok():
    print("ok")


def test_fail():
    assert 0


def test_error(error_fixture):
    pass


def test_skip():
    pytest.skip("skipping this test")


def test_xfail():
    pytest.xfail("xfailing this test")


@pytest.mark.xfail(reason="always xfail")
def test_xpass():
    pass
```

```
$ pytest -ra
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 6 items

test_example.py .FEsxX                                                 [100%]

================================== ERRORS ==================================
_____________________ ERROR at setup of test_error ______________________

    @pytest.fixture
    def error_fixture():
>       assert 0
E       assert 0

test_example.py:6: AssertionError
================================= FAILURES =================================
_______________________________ test_fail ________________________________

    def test_fail():
>       assert 0
E       assert 0

test_example.py:14: AssertionError
======================== short test summary info ========================
SKIPPED [1] $REGENDOC_TMPDIR/test_example.py:22: skipping this test
XFAIL test_example.py::test_xfail
  reason: xfailing this test
XPASS test_example.py::test_xpass always xfail
ERROR test_example.py::test_error - assert 0
```

(continues on next page)

(продолжение с предыдущей страницы)

```
FAILED test_example.py::test_fail - assert 0
== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.12s ===
```

Параметр `-r` принимает ряд символов после себя. Использованный выше символ `a` означает "все, кроме успешных».

Вот полный список доступных символов, которые можно использовать:

- `f` - упавшие (добавляет раздел FAILED)
- `E` - ошибки (добавляет раздел ERROR)
- `s` - пропущенные (добавляет раздел SKIPPED)
- `x` - тесты XFAIL (добавляет раздел XFAIL)
- `X` - тесты XPASS (добавляет раздел XPASS)
- `p` - успешные (passed)
- `P` - успешные (passed) с выводом

Есть и специальные символы для пропуска отдельных групп:

- `a` - выводить все, кроме `pP`
- `A` - выводить все
- `N` - ничего не выводить (может быть полезным, поскольку по умолчанию используется комбинация `fE`).

Можно использовать более одного символа. Например, для того, чтобы увидеть только упавшие и пропущенные тесты, можно выполнить:

```
$ pytest -rfs
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 6 items

test_example.py .FEsxX                                                 [100%]

================================== ERRORS ==================================
_____________________ ERROR at setup of test_error _____________________

    @pytest.fixture
    def error_fixture():
>       assert 0
E       assert 0

test_example.py:6: AssertionError
================================= FAILURES =================================
_______________________________ test_fail ________________________________

    def test_fail():
>       assert 0
E       assert 0

test_example.py:14: AssertionError
======================== short test summary info ========================
```

(continues on next page)

(продолжение с предыдущей страницы)

```
FAILED test_example.py::test_fail - assert 0
SKIPPED [1] $REGENDOC_TMPDIR/test_example.py:22: skipping this test
== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.12s ===
```

Использование `p` добавляет в сводный отчет успешные тесты, а `P` добавляет дополнительный раздел «пройдены" (PASSED) для тестов, которые прошли, но перехватили вывод:

```
$ pytest -rpP
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 6 items

test_example.py .FEsxX                                                 [100%]

================================== ERRORS ==================================
_____________________ ERROR at setup of test_error _____________________

    @pytest.fixture
    def error_fixture():
>       assert 0
E       assert 0

test_example.py:6: AssertionError
================================= FAILURES =================================
_______________________________ test_fail ________________________________

    def test_fail():
>       assert 0
E       assert 0

test_example.py:14: AssertionError
================================= PASSES ===================================
_______________________________ test_ok ____________________________________
--------------------------- Captured stdout call ---------------------------
ok
======================== short test summary info ========================
PASSED test_example.py::test_ok
== 1 failed, 1 passed, 1 skipped, 1 xfailed, 1 xpassed, 1 error in 0.12s ===
```

2.8 Запуск отладчика PDB (Python Debugger) при падении тестов

`python` содержит встроенный отладчик PDB (Python Debugger). `pytest` позволяет запустить отладчик с помощью параметра командной строки:

```
pytest --pdb
```

Использование параметра позволяет запускать отладчик при каждом падении теста (или прерывании его с клавиатуры). Часто хочется сделать это для первого же упавшего теста, чтобы понять причину его падения:

```
pytest -x --pdb   # вызывает отладчик при первом падении и завершает тестовую сессию
pytest --pdb --maxfail=3  # вызывает отладчик для первых трех падений
```

Обратите внимание, что при любом падении информация об исключении сохраняется в `sys.last_value`, `sys.last_type` и `sys.last_traceback`. При интерактивном использовании это позволяет перейти к отладке после падения с помощью любого инструмента отладки. Можно также вручную получить доступ к информации об исключениях, например:

```
>>> import sys
>>> sys.last_traceback.tb_lineno
42
>>> sys.last_value
AssertionError('assert result == "ok"',)
```

2.9 Запуск отладчика PDB (Python Debugger) в начале теста

`pytest` позволяет запустить отладчик сразу же при старте каждого теста. Для этого нужно передать следующий параметр:

```
pytest --trace
```

В этом случае отладчик будет вызываться при запуске каждого теста.

2.10 Установка точек останова

Чтобы установить точку останова, вызовите в коде `import pdb;pdb.set_trace()`, и `pytest` автоматически отключит перехват вывода для этого теста, при этом:

- на перехват вывода в других тестах это не повлияет;
- весь перехваченный ранее вывод будет обработан как есть;
- перехват вывода возобновится после завершения отладочной сессии (с помощью команды `continue`).

2.11 Использование встроенной функции breakpoint

`Python 3.7` содержит встроенную функцию `breakpoint()`. `pytest` поддерживает использование `breakpoint()` следующим образом:

- если вызывается `breakpoint()`, и при этом переменная `PYTHONBREAKPOINT` установлена в значение по умолчанию, `pytest` использует расширяемый отладчик PDB вместо системного;
- когда тестирование будет завершено, система снова будет использовать отладчик `Pdb` по умолчанию;
- если `pytest` вызывается с опцией `--pdb` то расширяемый отладчик PDB используется как для функции `breakpoint()`, так и для упавших тестов/необработанных исключений;
- для определения пользовательского класса отладчика можно использовать `--pdbcls`.

2.12 Профилирование продолжительности выполнения теста

Чтобы получить список 10 самых медленных тестов, выполните:

```
pytest --durations=10
```

По умолчанию, `pytest` не покажет тесты со слишком маленькой (менее одной сотой секунды) длительностью выполнения, если в командной строке не будет передан параметр `-vv`.

2.13 Модуль `faulthandler`

Стандартный модуль faulthandler можно использовать для сброса трассировок `Python` при ошибке или по истечении времени ожидания.

При запуске `pytest` модуль автоматически подключается, если только в командной строке не используется опция `-p no:faulthandler`.

Кроме того, для сброса трассировок всех потоков в случае, когда тест длится более `X` секунд, можно использовать опцию faulthandler_timeout=X (для Windows неприменима).

Примечание: Эта функциональность была интегрирована из внешнего плагина pytest-faulthandler с двумя небольшими изменениями:

- чтобы ее отключить, используйте `-p no:faulthandler` вместо `--no-faulthandler`;
- опция командной строки `--faulthandler-timeout` превратилась в конфигурационную опцию faulthandler_timeout. Ее по-прежнему можно настроить из команндной строки, используя `-o faulthandler_timeout=X`.

2.14 Создание файлов формата JUnit

Чтобы создать результирующие файлы в формате, понятном Jenkins или другому серверу непрерывной интеграции, используйте вызов:

```
pytest --junitxml=path
```

Команда создает xml-файл по указанному пути.

Чтобы задать имя корневого xml-элемента для набора тестов, можно настроить параметр `junit_suite_name` в конфигурационном файле:

```
[pytest]
junit_suite_name = my_suite
```

Спецификация JUnit XML, по-видимому, указывает, что атрибут `"time"` должен сообщать об общем времени выполнения теста, включая выполнение setup- и teardown- методов (1, 2). Это поведение `pytest` по умолчанию. Чтобы вместо этого сообщать только о длительности вызовов, настройте параметр `junit_duration_report` следующим образом:

```
[pytest]
junit_duration_report = call
```

2.14.1 record_property

Чтобы записать дополнительную информацию для теста, используйте фикстуру `record_property`:

```
def test_function(record_property):
    record_property("example_key", 1)
    assert True
```

Такая запись добавит дополнительное свойство `example_key="1"` к сгенерированному тегу `testcase`:

```
<testcase classname="test_function" file="test_function.py" line="0" name="test_function" time="0.
↪0009">
  <properties>
    <property name="example_key" value="1" />
  </properties>
</testcase>
```

Эту функциональность также можно использовать совместно с пользовательскими маркерами:

```
# content of conftest.py


def pytest_collection_modifyitems(session, config, items):
    for item in items:
        for marker in item.iter_markers(name="test_id"):
            test_id = marker.args[0]
            item.user_properties.append(("test_id", test_id))
```

И в тесте:

```
# content of test_function.py
import pytest


@pytest.mark.test_id(1501)
def test_function():
    assert True
```

В файле получим:

```
<testcase classname="test_function" file="test_function.py" line="0" name="test_function" time="0.
↪0009">
  <properties>
    <property name="test_id" value="1501" />
  </properties>
</testcase>
```

Предупреждение: Пожалуйста, обратите внимание, что использование этой возможности приведет к записи некорректного с точки зрения JUnitXML-схем последних версий файла и может вызывать проблемы при работе с некоторыми серверами непрерывной интеграции.

2.14.2 record_xml_attribute

Чтобы добавить дополнительный атрибут в элемент `testcase`, можно использовать фикстуру `record_xml_attribute`. Ее также можно использовать для переопределения существующих значений:

```
def test_function(record_xml_attribute):
    record_xml_attribute("assertions", "REQ-1234")
    record_xml_attribute("classname", "custom_classname")
    print("hello world")
    assert True
```

В отличие от `record_property`, дочерний элемент в данном случае не добавляется. Вместо этого в элемент `testcase` будет добавлен атрибут `assertions="REQ-1234"`, а значение атрибута `classname` по умолчанию будет заменено на `"classname=custom_classname"`:

```
<testcase classname="custom_classname" file="test_function.py" line="0" name="test_function" time=
↪"0.003" assertions="REQ-1234">
    <system-out>
        hello world
    </system-out>
</testcase>
```

Предупреждение: `record_xml_attribute` пока используется в режиме эксперимента, и в будущем может быть заменен чем-то более мощным и/или общим. Однако сама функциональность как таковая будет сохранена.

Использование `record_xml_attribute` поверх `record_xml_property` может быть полезным при парсинге xml-отчетов средствами непрерывной интеграции. Однако некоторые парсеры допускают не любые элементы и атрибуты. Многие инструменты (как в примере ниже), используют xsd-схему для валидации входящих xml. Поэтому убедитесь, что имена атрибутов, которые вы используете, являются допустимыми для вашего парсера.

Ниже представлена схема, которую использует `Jenkins` для валидации xml-отчетов:

```
<xs:element name="testcase">
    <xs:complexType>
        <xs:sequence>
            <xs:element ref="skipped" minOccurs="0" maxOccurs="1"/>
            <xs:element ref="error" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="failure" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="system-out" minOccurs="0" maxOccurs="unbounded"/>
            <xs:element ref="system-err" minOccurs="0" maxOccurs="unbounded"/>
        </xs:sequence>
        <xs:attribute name="name" type="xs:string" use="required"/>
        <xs:attribute name="assertions" type="xs:string" use="optional"/>
        <xs:attribute name="time" type="xs:string" use="optional"/>
        <xs:attribute name="classname" type="xs:string" use="optional"/>
        <xs:attribute name="status" type="xs:string" use="optional"/>
    </xs:complexType>
</xs:element>
```

Предупреждение: Пожалуйста, обратите внимание, что использование этой возможности приведет к записи некорректного с точки зрения JUnitXML-схем последних версий файла и может вызывать проблемы при работе с некоторыми серверами непрерывной интеграции.

Чтобы добавить свойства каждому тесту из набора, можно использовать фикстуру `record_testsuite_property` с параметром `scope="session"` (в этом случае она будет применяться ко всем тестам тестовой сессии).

```
import pytest


@pytest.fixture(scope="session", autouse=True)
def log_global_env_facts(record_testsuite_property):
    record_testsuite_property("ARCH", "PPC")
    record_testsuite_property("STORAGE_TYPE", "CEPH")


class TestMe:
    def test_foo(self):
        assert True
```

Этой фикстуре передаются имя (`name`) и значение (`value`) тэга `<property>`, который добавляется на уровне тестового набора для генерируемого xml-файла:

```
<testsuite errors="0" failures="0" name="pytest" skipped="0" tests="1" time="0.006">
  <properties>
    <property name="ARCH" value="PPC"/>
    <property name="STORAGE_TYPE" value="CEPH"/>
  </properties>
  <testcase classname="test_me.TestMe" file="test_me.py" line="16" name="test_foo" time="0.
↪000243663787842"/>
</testsuite>
```

`name` должно быть строкой, а `value` будет преобразовано в строку и корректно экранировано.

В отличие от случаев использования *record_property* и *record_xml_attribute* созданный xml-файл будет совместим с последним стандартом `xunit`.

2.15 Создание файлов в формате resultlog

Для создания машиночитаемых логов в формате `plain-text` можно выполнить

```
pytest --resultlog=path
```

и просмотреть содержимое по указанному пути `path`. Эти файлы также используются ресурсом PyPy-test для отображения результатов тестов после ревизий.

Предупреждение: Поскольку эта возможность редко используется, она запланирована к удалению в `pytest 6.0`.

Если вы пользуетесь ею, рассмотрите возможность использования нового плагина pytest-reportlog .

Подробнее см.: the deprecation docs.

2.16 Отправка отчетов на сервис pastebin

Создание ссылки для каждого упавшего теста:

```
pytest --pastebin=failed
```

Эта команда отправит информацию о прохождении теста на удаленный сервис регистрации и сгенерирует ссылку для каждого падения. Тесты можно отбирать как обычно, или, например, добавить `-x`, если вы хотите отправить данные по конкретному упавшему тесту.

Создание ссылки для лога тестовой сессии:

```
pytest --pastebin=all
```

В настоящее время реализовна регистрация только в сервисе http://bpaste.net.

Изменено в вресии 5.2

Если по каким-то причинам не удалось создать ссылку, вместо падения всего тестового набора генерируется предупреждение.

2.17 Подключение плагинов

В командной строке можно явно подгрузить какой-либо внутренний или внешний плагин, используя опцию `-p`:

```
pytest -p mypluginmodule
```

Опция принимает параметр `name`, который может быть:

- Полным именем модуля, записанным через точку, например `myproject.plugins`. Имя должно быть импортируемым.
- «Входным» именем плагина, которое передается в `setuptools` при регистрации плагина. К примеру, чтобы подгрузить pytest-cov , нужно использовать:

  ```
  pytest -p pytest_cov
  ```

2.18 Отключение плагинов

Чтобы отключить загрузку определенных плагинов во время вызова, используйте опцию `-p` с префиксом `no:`.

Пример: чтобы отключить загрузку плагина `doctest`, который отвечает за выполнение тестов из строк «docstring», вызовите `pytest` следующим образом:

```
pytest -p no:doctest
```

2.19 Вызов `pytest` из кода Python

`pytest` можно вызвать прямо в коде Python:

```
pytest.main()
```

Такой способ эквивалентен вызову «pytest» из командной строки. В этом случае вместо исключения `SystemExit` возвращается статус завершения. Можно также передавать параметры и опции:

```
pytest.main(["-x", "mytestdir"])
```

Дополнительные плагины можно указать в `pytest.main`:

```
# content of myinvoke.py
import pytest


class MyPlugin:
    def pytest_sessionfinish(self):
        print("*** test run reporting finishing")


pytest.main(["-qq"], plugins=[MyPlugin()])
```

Выполнив этот код, увидим, что `MyPlugin` был загружен и применен:

```
$ python myinvoke.py
.FEsxX.                                                              [100%]*** test run reporting␣
↪finishing

==================================== ERRORS ====================================
_______________________ ERROR at setup of test_error _______________________

    @pytest.fixture
    def error_fixture():
>       assert 0
E       assert 0

test_example.py:6: AssertionError
================================= FAILURES =================================
______________________________ test_fail _______________________________

    def test_fail():
>       assert 0
E       assert 0

test_example.py:14: AssertionError
```

Примечание: Вызов `pytest.main()` приводит к тому, что импортируются не только тесты, но и все модули, которые они используют. Из-за механизма кэширования импорта `Python` последующие вызовы `pytest.main()` из того же процесса не будут учитывать изменения в файлах, внесенные между вызовами. Поэтому не рекомендуется многократное использование `pytest.main()` в одном и том же процессе (например, при перезапуске тестов).

Глава 3

Использование pytest с существующими наборами тестов

pytest можно использовать с большинством существующих наборов тестов, однако его поведение отличается от поведения других фреймворков, таких как nose или встроенный в Python unittest.

3.1 Запуск существующих наборов тестов с помощью pytest

Допустим, вы хотите присоединиться к какому-либо существующему репозитарию. После того, как вы спомощью какой-то системы контроля версий получите локальную копию кода и установите виртуальное окружение, возможно, вам захочется запустить в корне проекта:

```
cd <repository>
pip install -e .  # альтернативная виртуальная среда, включающая
                  # 'python setup.py develop' и 'conda develop'
```

Это даст возможность создать символическую ссылку на ваш код, которая позволит его редактировать, в то время как ваши тесты будут запускаться на нем, как на установленном пакете.

Установка вашего проекта в режиме разработки позволит вам избежать переустановки пакета каждый раз, когда вы хотите запустить тесты. Это удобнее, чем возиться с sys.path, чтобы указать путь к вашим тестам в локальном коде.

Советуем также рассмотреть возможность использования *tox*.

Глава 4

Оператор assert и вывод информации о проверках

4.1 Проверка с помощью оператора assert

pytest позволяет использовать стандартный оператор языка Python - assert - для проверки соответствия ожидаемых результатов фактическим. Например, такую конструкцию

```
# content of test_assert1.py
def f():
    return 3


def test_function():
    assert f() == 4
```

можно использовать, чтобы убедиться что ваша функция вернет определенное значение. Если assert упадет, вы сможете увидеть значение, возвращаемое вызванной функцией:

```
$ pytest test_assert1.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_assert1.py F                                                  [100%]

================================ FAILURES ================================
_____________________________ test_function _____________________________

    def test_function():
>       assert f() == 4
E       assert 3 == 4
E        +  where 3 = f()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
test_assert1.py:6: AssertionError
============================ 1 failed in 0.12s ==============================
```

pytest поддерживает отображение значений наиболее распространенных операций, включая вызовы, параметры, сравнения, бинарные и унарные операции (см. *Python: примеры отчетов об ошибках pytest*). Это позволяет использовать стандартные конструкции python без шаблонного кода, не теряя при этом информацию.

Однако, если вы укажете в assert текст сообщения об ошибке, например, вот так,

```
assert a % 2 == 0, "value was odd, should be even"
```

то никакая аналитическая информация выводиться не будет, и в трейсбэке вы увидите просто указанное сообщение об ошибке.

См. *Детальный анализ неудачных проверок (assertion introspection)* для получения дополнительной информации об анализе операторов assert.

4.2 Проверка ожидаемых исключений

Чтобы убедиться в том, что вызвано ожидаемое исключение, нужно использовать assert в контексте pytest.raises. Например, так:

```
import pytest


def test_zero_division():
    with pytest.raises(ZeroDivisionError):
        1 / 0
```

А если нужно получить доступ к фактической информации об исключении, можно использовать:

```
def test_recursion_depth():
    with pytest.raises(RuntimeError) as excinfo:

        def f():
            f()

        f()
    assert "maximum recursion" in str(excinfo.value)
```

excinfo - это экземпляр класса ExceptionInfo, которым обертывается вызванное исключение. Наиболее интереесными его атрибутами являются .type, .value и .traceback.

Чтобы проверить, что регулярное выражение соответствует строковому представлению исключения (аналогично методу TestCase.assertRaisesRegexp в unittest), конекст-менеджеру можно передать параметр match:

```
import pytest


def myfunc():
    raise ValueError("Exception 123 raised")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def test_match():
    with pytest.raises(ValueError, match=r".* 123 .*"):
        myfunc()
```

Регулярное выражение из параметра `match` сопоставляется с функцией `re.search`, так что в приведенном выше примере `match='123'` также сработает.

Есть и альтернативный вариант использования `pytest.raises`, когда вы передаете функцию, которая которая должна выполняться с заданными `*args` и `**kwargs` и проверять, что вызвано указанное исключение:

```
pytest.raises(ExpectedException, func, *args, **kwargs)
```

В случае падения теста `pytest` выведет вам полезную информацию, например, о том, что исключение не вызвано (*no exception*) или вызвано неверное исключение (*wrong exception*).

Обратите внимание, что параметр `raises` можно также указать в декораторе `@pytest.mark.xfail`, который особым образом проверяет само падение теста, а не просто возникновение какого-то исключения:

```
@pytest.mark.xfail(raises=IndexError)
def test_f():
    f()
```

Использование `pytest.raises` скорре всего пригодится, когда вы тестируете исключения, генерируемые собственным кодом, а вот маркировка тестовой функции маркером `@pytest.mark.xfail`, наверное, лучше подойдет для документирования незафиксированных (когда тест описывает то, что «должно бы» происходить) или зависимых от чего-либо багов.

4.3 Проверка ожидаемых предупреждений

Проверить, что код генерирует ожидаемое предупреждение можно с помощью pytest.warns.

4.4 Использование контекстно-зависимых сравнений

pytest выводит подробный анализ контекстно-зависимой информации, когда сталкивается со сравнениями. Например, в результате исполнения этого модуля

```
# content of test_assert2.py


def test_set_comparison():
    set1 = set("1308")
    set2 = set("8035")
    assert set1 == set2
```

будет выведен следующий отчет:

```
$ pytest test_assert2.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
```

(continues on next page)

(продолжение с предыдущей страницы)

```
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_assert2.py F                                                    [100%]

================================= FAILURES =================================
___________________________ test_set_comparison ____________________________

    def test_set_comparison():
        set1 = set("1308")
        set2 = set("8035")
>       assert set1 == set2
E       AssertionError: assert {'0', '1', '3', '8'} == {'0', '3', '5', '8'}
E         Extra items in the left set:
E         '1'
E         Extra items in the right set:
E         '5'
E         Use -v to get the full diff

test_assert2.py:6: AssertionError
============================ 1 failed in 0.12s =============================
```

Вывод результатов сравнения для отдельных случаев:

- сравнение длинных строк: будут показаны различия
- сравнение длинных последовательностей: будет показан индекс первого несоответствия
- сравнение словарей: будут показаны различающиеся элементы

Больше примеров: *reporting demo*.

4.5 Определение собственных сообщений к упавшим `assert`

Можно добавить свое подробное объяснение, реализовав хук (hook) `pytest_assertrepr_compare` (см. pytest_assertrepr_compare).

Для примера рассмотрим добавление в файл *conftest.py* хука, который устанавливает наше сообщение для сравниваемых объектов `Foo`:

```
# content of conftest.py
from test_foocompare import Foo


def pytest_assertrepr_compare(op, left, right):
    if isinstance(left, Foo) and isinstance(right, Foo) and op == "==":
        return [
            "Comparing Foo instances:",
            "   vals: {} != {}".format(left.val, right.val),
        ]
```

Теперь напишем тестовый модуль:

```
# content of test_foocompare.py
class Foo:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def __init__(self, val):
        self.val = val

    def __eq__(self, other):
        return self.val == other.val


def test_compare():
    f1 = Foo(1)
    f2 = Foo(2)
    assert f1 == f2
```

Запустив тестовый модуль, получим сообщение, которое мы определили в файле `conftest.py`:

```
$ pytest -q test_foocompare.py
F                                                                    [100%]
================================= FAILURES =================================
_______________________________ test_compare _______________________________

    def test_compare():
        f1 = Foo(1)
        f2 = Foo(2)
>       assert f1 == f2
E       assert Comparing Foo instances:
E            vals: 1 != 2

test_foocompare.py:12: AssertionError
1 failed in 0.12s
```

4.6 Детальный анализ неудачных проверок (assertion introspection)

Детальный анализ упавших проверок достигается переопределением операторов `assert` перед запуском. Переопределенные `assert` помещают аналитическую информацию в сообщение о неудачной проверке. `pytest` переопределяет только тестовые модули, обнаруженные им в процессе сборки (collecting) тестов, поэтому **‘‘assert‘‘-ы в поддерживающих модулях, которые сами по себе не являются тестами, переопределены не будут**.

Можно вручную включить возможность переопределения `assert` для импортируемого модуля, вызвав register-assert-rewrite перед его импортом (лучше это сделать в корневом файле‘‘conftest.py‘‘).

Дополнительную информацию можно найти в статье Бенджамина Петерсона: Behind the scenes of pytest’s new assertion rewriting.

4.6.1 Кэширование переопределенных файлов

`pytest` кэширует переопределенные модули на диск. Можно отключить такое поведение (например, чтобы избежать устаревших `.pyc` файлов в проектах, которые задействуют множество файлов), добавив в ваш корневой файл `conftest.py`:

```
import sys

sys.dont_write_bytecode = True
```

Обратите внимание, что это не влияет на анализ упавших проверок, единственное отличие заключается в том, что `.pyc`-файлы не будут кэшироваться на диск.

Кроме того, кэширование при переопределении будет автоматически отключаться, если не получается записать новые `.pyc`- файлы, т. е. для read-only файлов или zip-архивов.

4.6.2 Отключение переопределения `assert`

При импорте `pytest` перезаписывает тестовые модули, используя хук импорта для записи новых `.pyc`-файлов. В большинстве случаев это работает. Тем не менее, при работе с механизмом импорта, такой способ может создавать проблемы.

На этот случай есть 2 опции:

- Отключите переопределение для отдельного модуля, добавив строку `PYTEST_DONT_REWRITE` в docstring (строковую переменную для документирования модуля).
- Отключите переопределение для всех модулей с помощью `--assert=plain`.

Глава 5

Фикстуры pytest: явные, модальные, расширяемые

Тестовые фикстуры инициализируют тестовые функции. Они обеспечивают надежность тестов, согласованнность и повторямость их результатов. При инициализации можно настраивать сервисы, состояния, переменные окружения. Доступ к ним осуществляется через аргументы тестовых функций; для каждой фикстуры, используемой тестовой функцией, в самой функции, как правило, существует соответствующий аргумент, имя которого совпадает с наименованием фикстуры.

Фикстуры `pytest` значительно удобнее классических `setup/teardown`-функций `xUnit`, поскольку:

- фикстуры имеют явные имена и активируются путем их объявления в тестовых функциях, модулях, классах и проектах.
- фикстуры реализованы модально: каждый вызов фикстуры инициализирует *функцию-фикстуру*, которая в свою очередь может использовать другие фикстуры.
- управление фикстурами расширяется от простого модуля до комплексного функционального тестирования, позволяя параметризовать фикстуры и тесты в соответствии с конфигурацией и опциями компонентов, или повторно использовать фикстуры внутри функции, класса, модуля или тестовой сесси в целом.

При этом `pytest` продолжает поддерживать *Классический «setup» в стиле xunit*. Можно смешивать оба стиля, постепенно переходя от классического стиля к новому, если вам так нравится. Можно начинать с существующего стиля unittest.TestCase или с проектов nose.

Фикстуры определяются с использованием декоратора @pytest.fixture, *описанного ниже*. В `pytest` есть полезные встроенные фикстуры, см. список встроенных фикстур.

5.1 Фикстуры как аргументы функций

Тестовые функции принимают фикстуры как входящий аргумент с тем же именем. Для каждого такого аргумента функция-фикстура предоставляет объект фикстуры. Для того, чтобы зарегистировать функцию как фикстуру, нужно использовать декоратор `@pytest.fixture`. Давайте рассмотрим простой тестовый модуль, содержащий фикстуру и использующую ее тестовую функцию:

```
# content of ./test_smtpsimple.py
import pytest


@pytest.fixture
def smtp_connection():
    import smtplib

    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)


def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
    assert 0  # в демонстрационных целях
```

Здесь `test_ehlo` использует значение фикстуры `smtp_connection`. При передаче аргумента `pytest` найдет и вызовет маркированную функцию-фикстуру `smtp_connection`. Запустив тест, увидим следующее:

```
$ pytest test_smtpsimple.py
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_smtpsimple.py F                                                 [100%]

================================= FAILURES =================================
_______________________________ test_ehlo ________________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
>       assert 0  # for demo purposes
E       assert 0

test_smtpsimple.py:14: AssertionError
========================== 1 failed in 0.12s ===============================
```

В трейсбеке мы видим, что тестовая функция была вызвана с аргументом `smtp_connection` - объектом `smtplib.SMTP()`, который был создан фикстурой. Тестовая функция упала на проверке `assert 0`. В данном случае `pytest` использует следующий алгоритм для вызова тестовой функции:

1. `pytest` *находит* функцию `test_ehlo` по ее префиксу `test_`. Ей передается аргумент с именем `smtp_connection`, поэтому `pytest` ищет и находит функцию с именем `smtp_connection`, помеченную как фикстура.
2. Фикстура `smtp_connection()` вызывается для создания объекта-функции.
3. Затем вызывается функция `test_ehlo(<объект smtp_connection>)`, выполняется и падает на последней строчке.

Обратите внимание, что если вы неправильно напишите имя аргумента-функции или попытаетесь использовать недоступную функцию, то получите ошибку со списком доступных аргументов-функций.

Примечание: Чтобы посмотреть список доступных фикстур, можно использовать опцию `--fixtures`:

```
pytest --fixtures test_simplefactory.py
```

При этом фикстуры с ведущим символом «_» будут выведены в список, только если вы используете опцию `-v`.

5.2 Фикстуры: яркий пример внедрения зависимостей

Фикстуры позволяют тестовым функциям легко получать предварительно инициализированные объекты и работать с ними, не заботясь об импорте/установке/очистке.

Вот яркий пример внедрения зависимостей, где фикстуры играют роль *внедренного объекта*, а тестовые функции являются *потребителями* объектов-фикстур.

5.3 `conftest.py`: расширение фикстур

Если вы планируете использовать фикстуру в нескольких тестах, то можно объявить ее в файле `conftest.py`. При этом импортировать ее не нужно - `pytest` найдет ее автоматически. Поиск фикстур начинается с тестовых классов, затем они иущется в тестовых модулях и в файлах `conftest.py`, и, в последнюю очередь, во встроенных и сторонних плагинах.

В `conftest.py` также можно встраивать плагины для подкаталогов.

5.4 Расширение тестовых данных

Хороший способ для того, чтобы сделать тестовые данные из файлов доступными для ваших тестов - загрузка этих данных в фикстуру. При этом используются механизмы кэширования `pytest`.

Еще один хороший подход заключается в добавлении файлов с данными в папку `tests`. Существуют также плагины, которые помогают управлять этим аспектом тестирования, например, pytest-datadir, или pytest-datafiles.

5.5 Область действия (уровень) фикстуры: расширение фикстуры на все тесты класса, модуля, сессии

Фикстуры, требующие доступа к сети, зависят от подключения и обычно требуют больших временных затрат на их создание. Расширяя предыдущий пример, мы можем добавить параметр `scope="module"` в декоратор фикстуры `@pytest.fixture` , чтобы функция-фикстура `smtp_connection` вызывалась только один раз для тестового модуля (по умолчанию параметр `scope` установлен в значение `function`). Таким образом, каждая тестовая функция модуля получит тот же самый объект `smtp_connection`, что позволит сэкономить время на создание подключения. Возможными значениями параметра `scope` являются `function`, `class`, `module`, `package` или `session`.

В следующем примере мы помещаем фикстуру в файл `conftest.py`, чтобы доступ к ней могли иметь разные тесты из разных тестовых модулей нашего каталога:

```
# content of conftest.py
import pytest
import smtplib


@pytest.fixture(scope="module")
def smtp_connection():
    return smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
```

Наша фикстура по-прежнему называется `smtp_connection`, и получить к ней доступ из любой тестовой функции или другой фикстуры (в пределах директории, в которой расположен наш файл `conftest.py` и ее поддиректорий) можно, передав параметр `smtp_connection` в объявлении нашей функции/фикстуры:

```
# content of test_module.py


def test_ehlo(smtp_connection):
    response, msg = smtp_connection.ehlo()
    assert response == 250
    assert b"smtp.gmail.com" in msg
    assert 0  # в демонстрационных целях


def test_noop(smtp_connection):
    response, msg = smtp_connection.noop()
    assert response == 250
    assert 0  # в демонстрационных целях
```

Здесь мы специально вставляем обреченные на неудачу операторы `assert 0`, чтобы посмотреть, что происходит при запуске тестов:

```
$ pytest test_module.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py FF                                                  [100%]

================================ FAILURES ================================
_____________________________ test_ehlo _____________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
        assert b"smtp.gmail.com" in msg
>       assert 0  # for demo purposes
E       assert 0

test_module.py:7: AssertionError
_____________________________ test_noop _____________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
        assert response == 250
>       assert 0  # for demo purposes
E       assert 0

test_module.py:13: AssertionError
============================ 2 failed in 0.12s =============================
```

Мы видим, что оба оператора `assert 0` упали. И, поскольку `pytest` показывает значения входящих параметров, мы также можем увидеть, что в обе тестовые функции был передан один и тот же объект `smtp_connection` (с уровнем модуля). В итоге, мы выполнили только одно smtp-подключение для обеих тестовых функций вместо двух.

Если вы предпочитаете создавать одно smtp-подключение на сессию, можно просто задать параметру `scope` значение `session`:

```
@pytest.fixture(scope="session")
def smtp_connection():
    # the returned fixture value will be shared for
    # all tests needing it
    ...
```

Соответственно, применив область действия `class`, получим один вызов фикстуры для класса.

Примечание: `pytest` кэширует только один экземпляр фикстуры одновременно. Это значит, что при использовании параметризованных фикстур `pytest` может вызывать фикстуру для заданного уровня больше одного раза.

5.5.1 Область действия `package` (экспериментальная возможность)

В `pytest 3.7` была введена область действия `package`. Фикстура с такой областью действия работает, пока не будет выполнен последний тест *пакета (package)*.

Предупреждение: Эта возможность рассматривается как **экспериментальная** и в последующих версиях может быть удалена, если при ее использовании будут обнаружены подводные камни или серьезные проблемы. Поэтому, пожалуйста, используйте ее с осторожностью и не забывайте сообщать об обнаруженных проблемах.

5.5.2 Динамическая область действия

В некоторых случаях можно изменять область действия фикстуры без изменения кода. Чтобы этого добиться, сделайте фикстуру *вызываемым объектом* (callable). Этот объект должен возвращать строку с допустимым значением области действия, и выполнен он будет только один раз - во время определения фикстуры. Такой объект вызывается с двумя ключевыми параметрами: строкой `fixture_name` и конфигурируемым объектом `config`.

В ситуациях, когда требуется значительное время для инициализации фикстуры (например, при создании процессов docker-контейнеров), такая возможность может быть очень полезна. Например,

можно использовать опцию командной строки для определения области действия процессов docker-контейнеров в разных виртуальных средах:

```
def determine_scope(fixture_name, config):
    if config.getoption("--keep-containers", None):
        return "session"
    return "function"


@pytest.fixture(scope=determine_scope)
def docker_container():
    yield spawn_container()
```

5.6 Порядок создания фикстур

При запросе фикстуры функцией сначала инициализиурются фикстуры с самой широкой областью действия - `session` и `module`, а затем - фикстуры более низкого уровня с областями `class` или `function`. В рамках одной тестовой функции порядок создания фикстур с одинаковой областью действия зависит от очередности вызова этих фикстур и установленных между ними зависимостей. При этом фикстуры с параметром `autouse = True` инициализируются прежде явно объявленных фикстур того же уровня.

Рассмотрим следующий код:

```
import pytest

# fixtures documentation order example
order = []


@pytest.fixture(scope="session")
def s1():
    order.append("s1")


@pytest.fixture(scope="module")
def m1():
    order.append("m1")


@pytest.fixture
def f1(f3):
    order.append("f1")


@pytest.fixture
def f3():
    order.append("f3")


@pytest.fixture(autouse=True)
def a1():
    order.append("a1")


@pytest.fixture
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def f2():
    order.append("f2")


def test_order(f1, m1, f2, s1):
    assert order == ["s1", "m1", "a1", "f3", "f1", "f2"]
```

Фикстуры, запрошенные функцией `test_order`, будут инициализированы в следующем порядке:

1. `s1`: фикстура с самой широкой областью действия (`session`).
2. `m1`: фикстура второго уровня (`module`).
3. `a1`: фикстура с областью действия `function` (`function-scoped` fixture) и параметром `autouse = True`: экземпляр этой фикстуры будет создан до создания остальных `function-scoped` фикстур.
4. `f3`: `function-scoped` фикстура, которую запрашивает функция `f1`: ее нужно создать в момент запроса
5. `f1`: первая `function-scoped` фикстура в списке аргументов функции `test_order`.
6. `f2`: последняя `function-scoped` фикстура в списке аргументов функции `test_order`.

5.7 Финализаторы в фикстуре / выполнение завершающего кода

`pytest` поддерживает выполнение фикстурами специфического завершающего кода при выходе из области действия. Если вы используете оператор `yield` вместо `return`, то весь код после `yield` выполняет роль «уборщика»:

```
# content of conftest.py

import smtplib
import pytest


@pytest.fixture(scope="module")
def smtp_connection():
    smtp_connection = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)
    yield smtp_connection  # возвращает значение фикстуры
    print("teardown smtp")
    smtp_connection.close()
```

Операторы `print` и `smtp.close()` будут выполнены после завершения последнего теста модуля независимо от того, было ли вызвано исключение или нет.

Давайте запустим:

```
$ pytest -s -q --tb=no
FFteardown smtp

========================= short test summary info ==========================
FAILED test_module.py::test_ehlo - assert 0
FAILED test_module.py::test_noop - assert 0
2 failed in 0.12s
```

Мы видим, что подключение `smtp_connection` было закрыто после выполнения двух тестов. Однако если вы зададите для фикстуры область действия `function`, то установка и разрыв соединения будут

производиться для каждого запущенного теста. В любом случае, нет нужды обрабатывать инициализацию и демонтаж соединения в самом модуле.

Обратите внимание, что можно использовать синтаксис `yield` с оператором `with`:

```
# content of test_yield2.py

import smtplib
import pytest


@pytest.fixture(scope="module")
def smtp_connection():
    with smtplib.SMTP("smtp.gmail.com", 587, timeout=5) as smtp_connection:
        yield smtp_connection  # возвращает значение фикстуры
```

После выполнения теста соединение `smtp_connection` будет разорвано, поскольку объект `smtp_connection` автоматически закрывается после завершения выполнения оператора `with`.

Использование финализатора менеджера контекста `contextlib.ExitStack()` гарантирует корректное закрытие соединений, вне зависимости от того, вызвала ли *установочная* часть кода фикстуры исключение. Это удобно, поскольку позволяет корректно очищать все ресурсы, созданные фикстурой, даже если один из них не удастся создать или получить:

```
# content of test_yield3.py

import contextlib

import pytest


@contextlib.contextmanager
def connect(port):
    ...  # устанавливаем соединение
    yield
    ...  # разрываем соединение


@pytest.fixture
def equipments():
    with contextlib.ExitStack() as stack:
        yield [stack.enter_context(connect(port)) for port in ("C1", "C3", "C28")]
```

Если в приведенном примере попытка установить соединение `"C28"` будет неудачной, `"C1"` и `"C3"` все равно будут корректно разорваны.

Обратите внимание: если исключение было вызвано во время выполнения *установочной* части (до оператора `yield`), *завершающий* код (после `yield`) выполнен не будет.

Альтернативным способом добиться выполнения *завершающего* кода является использование метода `addfinalizer` объекта *request-context* для регистрации финализатора.

Вот пример использования `addfinalizer` для разрыва соединения в фикстуре `smtp_connection`:

```
# content of conftest.py
import smtplib
import pytest
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@pytest.fixture(scope="module")
def smtp_connection(request):
    smtp_connection = smtplib.SMTP("smtp.gmail.com", 587, timeout=5)

    def fin():
        print("teardown smtp_connection")
        smtp_connection.close()

    request.addfinalizer(fin)
    return smtp_connection  # возвращает значение фикстуры
```

А вот пример фикстуры `equipments` с использованием `addfinalizer`:

```
# content of test_yield3.py

import contextlib
import functools

import pytest


@contextlib.contextmanager
def connect(port):
    ...  # устанавливаем соединение
    yield
    ...  # разрываем соединение


@pytest.fixture
def equipments(request):
    r = []
    for port in ("C1", "C3", "C28"):
        cm = connect(port)
        equip = cm.__enter__()
        request.addfinalizer(functools.partial(cm.__exit__, None, None, None))
        r.append(equip)
    return r
```

Оба метода - `yield` и `addfinalizer` - работают похоже, выполняя свою часть кода по завершении тестов. Конечно, если исключение будет вызвано **до** инициализации финализатора, ее код выполняться не будет.

5.8 Фикстуры могут анализировать запрашивающий контекст

Фикстура может принимать объект `request` для анализа контекста запрашивающей тестовой функции, класса или модуля. В продолжением предыдущего примера, давайте прочтем URL сервера из тестового модуля, который использует нашу фикстуру.

```
# content of conftest.py
import pytest
import smtplib


@pytest.fixture(scope="module")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def smtp_connection(request):
    server = getattr(request.module, "smtpserver", "smtp.gmail.com")
    smtp_connection = smtplib.SMTP(server, 587, timeout=5)
    yield smtp_connection
    print("finalizing {} ({})".format(smtp_connection, server))
    smtp_connection.close()
```

Здесь мы используем параметр `request.module` чтобы получить переменную `smtpserver` из модуля. Если мы просто запустим `pytest`, ничего особо не изменится:

```
$ pytest -s -q --tb=no
FFfinalizing <smtplib.SMTP object at 0xdeadbeef> (smtp.gmail.com)

======================== short test summary info =========================
FAILED test_module.py::test_ehlo - assert 0
FAILED test_module.py::test_noop - assert 0
2 failed in 0.12s
```

Давайте быстренько создадим еще один тестовый модуль, который задает URL сервера в простарнстве имен модуля:

```
# content of test_anothersmtp.py

smtpserver = "mail.python.org"  # будет прочитан фикстурой smtp_connection


def test_showhelo(smtp_connection):
    assert 0, smtp_connection.helo()
```

Запустим:

```
$ pytest -qq --tb=short test_anothersmtp.py
F                                                                    [100%]
================================ FAILURES ================================
______________________________ test_showhelo _______________________________
test_anothersmtp.py:6: in test_showhelo
    assert 0, smtp_connection.helo()
E   AssertionError: (250, b'mail.python.org')
E   assert 0
------------------------- Captured stdout teardown -------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef> (mail.python.org)
```

Вуаля! Фикстура `smtp_connection` взяла имя нашего почтового сервера из пространства имен использующего ее модуля.

5.9 Фикстура как фабрика данных

Шаблон «фабрика-фикстура» может помочь в ситуациях, когда результат, возвращаемый фикстурой используется много раз в отдельном тесте. Суть в том, что вместо того, чтобы напрямую возвращать данные, фикстура возвращает функцию, которая генерирует данные. И затем эта функция может быть неоднократно вызвана в тесте.

Если нужно, фабрики-фикстуры могут принимать параметры:

```
@pytest.fixture
def make_customer_record():
    def _make_customer_record(name):
        return {"name": name, "orders": []}

    return _make_customer_record


def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

Если нужно управлять данными, созданными фабриками, фикстура позаботится и об этом (в нашем случае, будут очищены созданные записи):

```
@pytest.fixture
def make_customer_record():

    created_records = []

    def _make_customer_record(name):
        record = models.Customer(name=name, orders=[])
        created_records.append(record)
        return record

    yield _make_customer_record

    for record in created_records:
        record.destroy()


def test_customer_records(make_customer_record):
    customer_1 = make_customer_record("Lisa")
    customer_2 = make_customer_record("Mike")
    customer_3 = make_customer_record("Meredith")
```

5.10 Параметризация фикстур

Фикстуры могут быть параметризованы, если их нужно вызывать неоднократно, выполняя несколько одинаковых, использующих эти фикстуры, тестов. Обычно повторно запускаемые тестовые функции не зависят друг от друга. И в этом случае параметризация фикстур помогает писать исчерпывающие функциональные тесты для компонентов, которые сами по себе могут быть сконфигурированы разными способами.

Расширяя предыдущий пример, мы можем указать фикстуре, что нужно создавать два объекта `smtp_connection`, тем самым заставив все тесты, ее использующие, выполняться дважды:

```
# content of conftest.py
import pytest
import smtplib


@pytest.fixture(scope="module", params=["smtp.gmail.com", "mail.python.org"])
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def smtp_connection(request):
    smtp_connection = smtplib.SMTP(request.param, 587, timeout=5)
    yield smtp_connection
    print("finalizing {}".format(smtp_connection))
    smtp_connection.close()
```

Главным внесенным изменением является объявление списка параметров `params` в декораторе `@pytest.fixture <_pytest.python.fixture>`, для каждого из которых фикстура будет выполняться и получать значение `request.param`. Менять код в тестовой функции не нужно. Давайте запустим на ш тестовый модуль «test_module.py»:

```
$ pytest -q test_module.py
FFFF                                                                 [100%]
================================ FAILURES ================================
_________________________ test_ehlo[smtp.gmail.com] _________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
        assert b"smtp.gmail.com" in msg
>       assert 0  # for demo purposes
E       assert 0

test_module.py:7: AssertionError
_________________________ test_noop[smtp.gmail.com] _________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
        assert response == 250
>       assert 0  # for demo purposes
E       assert 0

test_module.py:13: AssertionError
_________________________ test_ehlo[mail.python.org] _________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>

    def test_ehlo(smtp_connection):
        response, msg = smtp_connection.ehlo()
        assert response == 250
>       assert b"smtp.gmail.com" in msg
E       AssertionError: assert b'smtp.gmail.com' in b'mail.python.org\nPIPELINING\nSIZE␣
↪51200000\nETRN\nSTARTTLS\nAUTH DIGEST-MD5 NTLM CRAM-
↪MD5\nENHANCEDSTATUSCODES\n8BITMIME\nDSN\nSMTPUTF8\nCHUNKING'

test_module.py:6: AssertionError
-------------------------- Captured stdout setup ---------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef>
_________________________ test_noop[mail.python.org] _________________________

smtp_connection = <smtplib.SMTP object at 0xdeadbeef>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def test_noop(smtp_connection):
        response, msg = smtp_connection.noop()
        assert response == 250
>       assert 0  # for demo purposes
E       assert 0

test_module.py:13: AssertionError
------------------------- Captured stdout teardown -------------------------
finalizing <smtplib.SMTP object at 0xdeadbeef>
4 failed in 0.12s
```

Мы видим, что каждая из пары наших тестовых функций была выполнена дважды, сначала с одним, потом с другим объектом `smtp_connection`. Обратите внимание, что с соединением `mail.python.org` второй тест упал на функции `test_ehlo`, поскольку мы ожидали найти в сообщении строку с другим названием сервера.

Для каждого значения параметра параметризованной фикстуры `pytest` сгенерирует ID - строку, которая его идентифицирует (например, строки `test_ehlo[smtp.gmail.com]` и `test_ehlo[mail.python.org]` для нашего кода). Можно использовать эти ID вместе с опцией `-k` для выбора отдельных вариантов теста для запуска. По этим же ID можно понять, какой именно параметр использовался в упавшем тесте. Если вы запустите `pytest` с опцией `--collect-only`, то сможете увидеть все сгенерированные ID.

Числа, строки, логические значения и значение None имеют свои строковые представления, которые используются в ID теста. Для остальных объектов `pytest` создает строку, основываясь на имени аргумента. С помощью ключевого слова `ids` можно самостоятельно определить строку, которая будет использоваться в ID теста для определенного значения фикстуры:

```
# content of test_ids.py
import pytest


@pytest.fixture(params=[0, 1], ids=["spam", "ham"])
def a(request):
    return request.param


def test_a(a):
    pass


def idfn(fixture_value):
    if fixture_value == 0:
        return "eggs"
    else:
        return None


@pytest.fixture(params=[0, 1], ids=idfn)
def b(request):
    return request.param


def test_b(b):
    pass
```

Пример выше показывает, что `ids` можно определять как списком строк, так и функцией, которая

будет вызвана со значением фикстуры и вернет строку. В последнем случае, если функция вернет `None`, то `pytest` сгенерирует ID автоматически.

При запуске тестов из примеров выше будут сгенерированы следующие ID:

```
$ pytest --collect-only
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 10 items
<Module test_anothersmtp.py>
  <Function test_showhelo[smtp.gmail.com]>
  <Function test_showhelo[mail.python.org]>
<Module test_ids.py>
  <Function test_a[spam]>
  <Function test_a[ham]>
  <Function test_b[eggs]>
  <Function test_b[1]>
<Module test_module.py>
  <Function test_ehlo[smtp.gmail.com]>
  <Function test_noop[smtp.gmail.com]>
  <Function test_ehlo[mail.python.org]>
  <Function test_noop[mail.python.org]>

========================== no tests ran in 0.12s ==========================
```

5.11 Использование маркировки с параметризованными фикстурами

pytest.param можно использовать для маркировки значений параметров параметризованных фикстур, точно так же, как и с *@pytest.mark.parametrize*.

Пример:

```
# content of test_fixture_marks.py
import pytest


@pytest.fixture(params=[0, 1, pytest.param(2, marks=pytest.mark.skip)])
def data_set(request):
    return request.param


def test_data(data_set):
    pass
```

При выполнении этого теста вызов `data_set` со значением 2 будет пропущен (*skipped*).

```
$ pytest test_fixture_marks.py -v
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 3 items
```

(continues on next page)

(продолжение с предыдущей страницы)

```
test_fixture_marks.py::test_data[0] PASSED                          [ 33%]
test_fixture_marks.py::test_data[1] PASSED                          [ 66%]
test_fixture_marks.py::test_data[2] SKIPPED                         [100%]

======================= 2 passed, 1 skipped in 0.12s =======================
```

5.12 Модальность: использование фикстур фикстурами

Фикстуры могут использоваться не только тестовыми функциями, но и другими фикстурами. Это помогает делать ваши тесты модальными и дает возможность повторного использования фреймворк-зависимых фикстур во множестве проектов. Чтобы продемонстрировать это, давайте расширим предыдущий пример и инициализируем объект `app`, в котором будем использовать уже объявленный ресурс `smtp_connection`:

```
# content of test_appsetup.py

import pytest


class App:
    def __init__(self, smtp_connection):
        self.smtp_connection = smtp_connection


@pytest.fixture(scope="module")
def app(smtp_connection):
    return App(smtp_connection)


def test_smtp_connection_exists(app):
    assert app.smtp_connection
```

Здесь мы объявляем фикстуру `app`, которая принимает ранее объявленную фикстуру `smtp_connection` и создает с ее помощью объект `App`:

```
$ pytest -v test_appsetup.py
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 2 items

test_appsetup.py::test_smtp_connection_exists[smtp.gmail.com] PASSED [ 50%]
test_appsetup.py::test_smtp_connection_exists[mail.python.org] PASSED [100%]

============================ 2 passed in 0.12s =============================
```

Поскольку фикстура `smtp_connection` параметризована, тест запустится дважды с разными экземплярами приложения `App` и соответствующими им серверами smtp. Нам не нужно параметризовать `smtp_connection` в фикстуре `app`, так как `pytest` самостоятельно анализирует граф зависимостей.

Обратите внимание, что фикстура `app` имеет уровень модуля и использует фикстуру `smtp_connection` того же уровня. Пример будет работать и в том случае, если `smtp_connection` будет кэшироваться на

уровне сессии: для фикстур нормально использовать другие фикстуры с более обширной областью действия, но не наоборот - фикстура уровня сессии не может полноценно использовать фикстуру уровня модуля.

5.13 Автоматическая группировка тестов экземплярами фикстур

pytest минимизирует число активных фикстур во время выполнения теста. Если у вас есть параметризованная фикстура, то каждый экземпляр теста, ее использующий, сначала запускается с очередным параметром, а затем вызывает финализатор прежде, чем слудующий объект фикстуры будет инициализирован. Это, как и другие предоставляемые возможности, облегчает тестирование приложений, которые создают и используют глобальные состояния.

Следующий пример использует две параметризованные фикстуры, одна из которых имеет уровень модуля, и обе функции вызывают print, чтобы продемонстрировать поток инициализации/завершения.

```
# content of test_module.py
import pytest


@pytest.fixture(scope="module", params=["mod1", "mod2"])
def modarg(request):
    param = request.param
    print("  SETUP modarg", param)
    yield param
    print("  TEARDOWN modarg", param)


@pytest.fixture(scope="function", params=[1, 2])
def otherarg(request):
    param = request.param
    print("  SETUP otherarg", param)
    yield param
    print("  TEARDOWN otherarg", param)


def test_0(otherarg):
    print("  RUN test0 with otherarg", otherarg)


def test_1(modarg):
    print("  RUN test1 with modarg", modarg)


def test_2(otherarg, modarg):
    print("  RUN test2 with otherarg {} and modarg {}".format(otherarg, modarg))
```

Давайте запустим код в режиме подробных сообщений (с опцией -v) и посмотрим на вывод:

```
$ pytest -v -s test_module.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 8 items

```

(continues on next page)

(продолжение с предыдущей страницы)

```
test_module.py::test_0[1]   SETUP otherarg 1
  RUN test0 with otherarg 1
PASSED  TEARDOWN otherarg 1

test_module.py::test_0[2]   SETUP otherarg 2
  RUN test0 with otherarg 2
PASSED  TEARDOWN otherarg 2

test_module.py::test_1[mod1]   SETUP modarg mod1
  RUN test1 with modarg mod1
PASSED
test_module.py::test_2[mod1-1]   SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod1
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[mod1-2]   SETUP otherarg 2
  RUN test2 with otherarg 2 and modarg mod1
PASSED  TEARDOWN otherarg 2
test_module.py::test_1[mod2]   TEARDOWN modarg mod1
  SETUP modarg mod2
  RUN test1 with modarg mod2
PASSED

test_module.py::test_2[mod2-1]   SETUP otherarg 1
  RUN test2 with otherarg 1 and modarg mod2
PASSED  TEARDOWN otherarg 1

test_module.py::test_2[mod2-2]   SETUP otherarg 2
  RUN test2 with otherarg 2 and modarg mod2
PASSED  TEARDOWN otherarg 2
  TEARDOWN modarg mod2


========================== 8 passed in 0.12s ===========================
```

Вы можете увидеть, что параметризация фикстуры `modarg` на уровне модуля привела к выполнению тестов в порядке, позволяющем минимизировать «активные» ресурсы. Финализатор фикстуры с параметром `mod1` был вызван до инициализация фикстуры с параметром `mod2`.

Заметьте, что `test_0` полностью независим и поэтому был завершен первым. `test_1` был выполнен с параметром `mod1`, потом с тем же параметром был запущен `test_2`, после этого - `test_1` с параметром `mod2`, последним был запущен `test_2``с параметром ``mod2`.

При этом параметризованная фикстура `otherarg` с уровнем функции инициализировалась и демонтировалась для каждого теста, который ее использовал.

5.14 Использование фикстур в классах, модулях и проектах

Иногда тестовым функциям не нужно напрямую обращаться к объекту фикстуры. Например, для тестов может потребоваться пустой рабочий каталог, но нам не важно, какой именно каталог это будет. Вот здесь можно посмотреть, как для этого использовать встроенную фикстуру `pytest` tempfile. Мы же опишем создание такой фикстуры в файле `conftest.py`:

```
# content of conftest.py

import os
import shutil
import tempfile

import pytest


@pytest.fixture
def cleandir():
    old_cwd = os.getcwd()
    newpath = tempfile.mkdtemp()
    os.chdir(newpath)
    yield
    os.chdir(old_cwd)
    shutil.rmtree(newpath)
```

Затем объявим ее использование в тестовом модуле с помощью декортатора `usefixtures`:

```
# content of test_setenv.py
import os
import pytest


@pytest.mark.usefixtures("cleandir")
class TestDirectoryInit:
    def test_cwd_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
        with open("myfile", "w") as f:
            f.write("hello")

    def test_cwd_again_starts_empty(self):
        assert os.listdir(os.getcwd()) == []
```

Фикстура `cleandir` будет инициализироваться для выполнения каждого тестового метода. Давайте запустим код и убедимся, что наша фикстура инициализируется и тесты проходят:

```
$ pytest -q
..                                                                   [100%]
2 passed in 0.12s
```

Также можно «прицепить» несколько фикстур сразу

```
@pytest.mark.usefixtures("cleandir", "anotherfixture")
def test():
    ...
```

и определять использование фикстуры на уровне модуля, используя возможности механизма маркировки:

```
pytestmark = pytest.mark.usefixtures("cleandir")
```

Обратите внимание, что переменная **должна** называться именно `pytestmark`; если вы назовете ее, например, `foomark`, ваша фикстура инициализироваться не будет.

Можно также затребовать вашу фикстуру для всех тестов проекта, указав в «ini»-файле:

```
# content of pytest.ini
[pytest]
usefixtures = cleandir
```

Предупреждение: Внимание! Такая маркировка неэффективна для **функций-фикстур**! Нижеприведенный код **не будет работать так, как должен**:

```
@pytest.mark.usefixtures("my_other_fixture")
@pytest.fixture
def my_fixture_that_sadly_wont_use_my_other_fixture():
    ...
```

На данный момент подобный код не генерирует ошибок или предупреждений, но это планируется исправить, см. #3664.

5.15 Фикстуры autouse (автоматическое использование фикстур)

Иногда хочется, чтобы фикстуры вызывались автоматически, без явного указания их в качестве аргумента и без использования декоратора *usefixtures* . Предположим, у нас есть фикстура, имитирующая базу данных с архитектурой «begin/rollback/commit» и мы хотим автоматически обернуть каждый тестовый метод транзакцией и откатом к начальному состоянию. Вот макет реализации этой идеи:

```
# content of test_db_transact.py

import pytest


class DB:
    def __init__(self):
        self.intransaction = []

    def begin(self, name):
        self.intransaction.append(name)

    def rollback(self):
        self.intransaction.pop()


@pytest.fixture(scope="module")
def db():
    return DB()


class TestClass:
    @pytest.fixture(autouse=True)
    def transact(self, request, db):
        db.begin(request.function.__name__)
        yield
        db.rollback()

    def test_method1(self, db):
        assert db.intransaction == ["test_method1"]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def test_method2(self, db):
        assert db.intransaction == ["test_method2"]
```

Фикстура `transact` уровня класса промаркирована *autouse=true*, и это означает, что все тестовые методы класса будут использовать эту фикстуру без необходимости указывать ее в сигнатуре тестовой функции или использовать на уровне класса декоратор `usefixtures`.

Запустив, получим два успешно пройденных теста:

```
$ pytest -q
..                                                                   [100%]
2 passed in 0.12s
```

Вот как работают фикстуры на разных уровнях:

- «autouse»-фикстуры соблюдают область действия, определенную с помощью параметра `scope=`: если для фикстуры установлен уровень `scope='session'` - она будет инициализирована только один раз, при этом неважно, где она определена. `scope='class'` означает инициализацию один раз для класса и т. д.
- если «autouse»-фикстура определена в тестовом модуле, то ее будут автоматически использовать все тесты модуля.
- если «autouse»-фикстура определена в файле `conftest.py`, то вызывать фикстуру будут все тесты во всех тестовых модулях соответствующей директории.
- и, наконец (**пожалуйста, используйте эту возможность с осторожностью**): если вы определяете «autouse»-фикстуру в плагине, она будет вызываться для всех тестов во всех проектах, где установлен этот плагин. Это может быть полезно, если фикстура работает только при определенных настройках (указанных, например, в «ini»-файлах). Такая глобальная фикстура всегда должна быстро определять, нужно ли ей что-либо делать, чтобы избежать дорогостоящего импорта и вычислений.

Что касается приведенной выше фикстуры `transact`, вы можете захотеть, чтобы она была доступна в вашем проекте, не будучи при этом активной. Классический способ сделать это - поместить ее в файл `conftest.py`, **не применяяя «autouse»**:

```
# content of conftest.py
@pytest.fixture
def transact(request, db):
    db.begin()
    yield
    db.rollback()
```

И затем, если понадобится, создать тестовый класс, объявив ее использование:

```
@pytest.mark.usefixtures("transact")
class TestClass:
    def test_method1(self):
        ...
```

В этом случае фикстуру `transact` будут использовать все тестовые методы класса `TestClass`; остальные тесты не будут к ней обращаться, пока вы так же явно не укажете необходимость ее использования.

5.16 Переопределение фикстур разного уровня

В больших тестовых наборах вам может понадобиться переопределять глобальные (`global`) или корневые (`root`) фкстуры локальными (`locally`), сохраняя тестовый код читабельным и поддерживаемым.

5.16.1 Переопределение фикстур на уровне каталога (conftest.py)

Допустим, наш проект имеет такую файловую структуру:

```
tests/
    __init__.py

    conftest.py
        # content of tests/conftest.py
        import pytest

        @pytest.fixture
        def username():
            return 'username'

    test_something.py
        # content of tests/test_something.py
        def test_username(username):
            assert username == 'username'

    subfolder/
        __init__.py

        conftest.py
            # content of tests/subfolder/conftest.py
            import pytest

            @pytest.fixture
            def username(username):
                return 'overridden-' + username

        test_something.py
            # content of tests/subfolder/test_something.py
            def test_username(username):
                assert username == 'overridden-username'
```

Как видите, фикстура с одним и тем же именем может быть переопределена на уровне конкретного подкаталога. При этом «базовая» фикстура доступна в переопределенной (см. пример выше).

5.16.2 Переопределение фикстуры на уровне тестового модуля

Рассмотрим еще одну файловую структуру:

```
tests/
    __init__.py

    conftest.py
        # content of tests/conftest.py
        import pytest
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        @pytest.fixture
        def username():
            return 'username'

    test_something.py
        # content of tests/test_something.py
        import pytest

        @pytest.fixture
        def username(username):
            return 'overridden-' + username

        def test_username(username):
            assert username == 'overridden-username'

    test_something_else.py
        # content of tests/test_something_else.py
        import pytest

        @pytest.fixture
        def username(username):
            return 'overridden-else-' + username

        def test_username(username):
            assert username == 'overridden-else-username'
```

Пример показывает, как можно переопределить фикстуру с одним и тем же именем в конкретном тестовом модуле.

5.16.3 Переопределению фикстуры с помощью параметризации

Возьмем следующую структуру тестов:

```
tests/
    __init__.py

    conftest.py
        # content of tests/conftest.py
        import pytest

        @pytest.fixture
        def username():
            return 'username'

        @pytest.fixture
        def other_username(username):
            return 'other-' + username

    test_something.py
        # content of tests/test_something.py
        import pytest

        @pytest.mark.parametrize('username', ['directly-overridden-username'])
        def test_username(username):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            assert username == 'directly-overridden-username'

        @pytest.mark.parametrize('username', ['directly-overridden-username-other'])
        def test_username_other(other_username):
            assert other_username == 'other-directly-overridden-username-other'
```

В этом варианте значение фикстуры переопределяется значением параметра. Обратите внимание, что значение фикстуры может быть переопределено таким способом, деже если тесты не используют фикстуру напрямую (т. е. она не упоминается в самих функциях).

5.16.4 Замена параметризованной фикстуры непараметризованной и наоборот

Расммотрим такую структуру:

```
tests/
    __init__.py

    conftest.py
        # content of tests/conftest.py
        import pytest

        @pytest.fixture(params=['one', 'two', 'three'])
        def parametrized_username(request):
            return request.param

        @pytest.fixture
        def non_parametrized_username(request):
            return 'username'

    test_something.py
        # content of tests/test_something.py
        import pytest

        @pytest.fixture
        def parametrized_username():
            return 'overridden-username'

        @pytest.fixture(params=['one', 'two', 'three'])
        def non_parametrized_username(request):
            return request.param

        def test_username(parametrized_username):
            assert parametrized_username == 'overridden-username'

        def test_parametrized_username(non_parametrized_username):
            assert non_parametrized_username in ['one', 'two', 'three']

    test_something_else.py
        # content of tests/test_something_else.py
        def test_username(parametrized_username):
            assert parametrized_username in ['one', 'two', 'three']

        def test_username(non_parametrized_username):
            assert non_parametrized_username == 'username'
```

Здесь параметризованная фикстура заменяется непараметризованной и наоборот в рамках конкретного

тестового модуля. То же самое можно проделывать и для тестовых каталогов/подкаталогов.

Глава 6

Маркировка тестов

Благодаря `pytest.mark` можно передавать метаданные в ваши тесты. Ниже приведены примеры некоторых встроенных маркеров:

- *skip* - пропускает (skip) тестовую функцию
- *skipif* - пропускает (skip) тестовую функцию при соблюдении определенных условий
- *xfail* - помечает тест как «ожидаемо падающий» (xfail) при соблюдении каких-то условий
- *parametrize* - используется для многократного вызова одной и той же тестовой функции с разными параметрами

Можно с легкостью создавать маркеры и применять их ко всему тестовому классу или модулю. Маркеры можно использовать в плагинах и для *отбора тестов* при запуске `pytest` из командной строки с опцией `-m`.

Примеры и документацию см. *Работа с пользовательской маркировкой*.

Примечание: Маркеры можно применять только к самим тестам, на *фикстуры* они никак не влияют.

6.1 Регистрация маркера

Свой маркер можно зарегистрировать в файле `pytest.ini`:

```
[pytest]
markers =
    slow: помечает тесты как "медленные" (не будут выбраны при запуске с '-m "not slow"')
    serial
```

При этом все, что после `:` - это описание маркера (опционально).

Можно также зарегистрировать новые маркеры программно в хуке (hook) pytest_configure.

```
def pytest_configure(config):
    config.addinivalue_line(
        "markers", "env(name): mark test to run only on named environment"
    )
```

Если маркер зарегистрирован, то запуск `pytest` не генерирует предупреждений (см. ниже). Поэтому рекомендуется всегда регистрировать свои маркеры для сторонних плагинов.

6.2 Генерация ошибок при обнаружении неопознанного маркера

Незарегистрированные маркеры, применяемые с помощью декоратора `@pytest.mark.name_of_the_mark`, будут всегда генерировать предупреждения. Это сделано, чтобы избежать сюрпризов из-за опечаток. В предыдущем разделе рассказано, как отключить генерацию предупреждений для настроенных маркеров с помощью регистрации их в `pytest.ini` или хуке `pytest_configure`.

Если применить опцию командной строки `--strict-markers`, то все неопознанные маркеры, используемые декоратором `@pytest.mark.name_of_the_mark`, будут генерировать ошибку. Для своего проекта эту опцию можно применить по умолчанию, добавив `--strict-markers` в `addopts` в `pytest.ini`.

```
[pytest]
addopts = --strict-markers
markers =
    slow: marks tests as slow (deselect with '-m "not slow"')
    serial
```

Глава 7

Skip и xfail: работа с тестами, которые не могут быть пройдены

Тестовые функции, которые не могут быть запущены на определенных платформах или от которых вы ожидаете сбоя, можно пометить так, чтобы `pytest` работал с ними соответствующим образом и представлял сводку сеанса тестирования, считая набор тестов пройденным и помечая его *зеленым*.

skip (пропуск) используется в случае, когда вы ожидаете, что ваш тест пройдет только при соблюдении некоторых условий, в противном случае `pytest` должен полностью пропустить выполнение теста. Распространенными примерами являются пропуск тестов только для windows на платформах, отличных от windows, или пропуск тестов, зависящих от внешнего ресурса, который в данный момент недоступен (например, базы данных).

xfail применяется, когда вы ожидаете, что тест по каким-то причинам должен упасть. Обычный пример - это тест на еще не реализованную функцию или еще не исправленную ошибку. Когда тест, помеченный `pytest.mark.xfail`, проходит, несмотря на ожидаемое падение, в сводке результатов он будет помечен как **xpass**.

`pytest` подсчитывает и перечисляет тесты, помеченные `skip` и `xfail`, отдельно. Подробная информация о пропущенных / упавших тестах по умолчанию не отображается, чтобы не загромождать выходные данные. Чтобы увидеть детали, соответствующие «коротким» буквам, показанным в ходе выполнения теста, можно использовать параметр `-r`, как показано ниже:

```
pytest -rxXs  # показывать дополнительную информацию о тестах xfailed, xpassed, и skipped
```

Больше информации о параметре `-r` можно получить, выполнив команду `pytest -h` (вызов справки).

(См. *как изменить опции командной строки по умолчанию*)

7.1 Пропуск тестовых функций

Простейший способ пропустить тестовую функцию - пометить ее декоратором `skip`, которому может быть передана в качестве параметра `reason` причина пропуска:

```
@pytest.mark.skip(reason="no way of currently testing this")
def test_the_unknown():
    ...
```

Также можно пропустить тест непосредственно во время выполнения, вызвав функцию `pytest.skip(reason)`:

```
def test_function():
    if not valid_config():
        pytest.skip("unsupported configuration")
```

Такой способ может быть полезен, когда невозможно определить условие пропуска во время импорта.

Можно также пропустить выполнение всего тестового модуля - для этого на уровне модуля используется метод `pytest.skip(reason, allow_module_level = True)`:

```
import sys
import pytest

if not sys.platform.startswith("win"):
    pytest.skip("skipping windows-only tests", allow_module_level=True)
```

7.1.1 skipif

Этот декоратор используется, если вы хотите пропускать или не пропускать тесты в зависимости от выполнения какого-либо условия. Ниже - пример тестовой функции, которую следует пропустить при запуске интерпретатора Python ниже версии 3.6:

```
import sys


@pytest.mark.skipif(sys.version_info < (3, 6), reason="requires python3.6 or higher")
def test_function():
    ...
```

Если во время сбора данных условие выполняется (принимает значение `True`) - тестовая функция будет пропущена, а указанная причина при использовании параметра `-rs` отобразится в отчете.

Маркер `skipif` можно использовать совместно для нескольких модулей. Рассмотрим следующий тестовый модуль:

```
# content of test_mymodule.py
import mymodule

minversion = pytest.mark.skipif(
    mymodule.__versioninfo__ < (1, 1), reason="at least mymodule-1.1 required"
)


@minversion
def test_function():
    ...
```

Можно импортировать маркер и использовать его в другом тестовом модуле:

```
# test_myothermodule.py
from test_mymodule import minversion


@minversion
def test_anotherfunction():
    ...
```

Для больших наборов тестов обычно рекомендуется иметь один файл, в котором определяются маркеры, которые затем последовательно применяются во всем наборе тестов.

Кроме того, можно использовать строки условий (см. Conditions as strings instead of booleans) вместо логических значений, но их нельзя легко переносить между модулями, поэтому они поддерживаются главным образом из соображений обратной совместимости.

7.1.2 Пропуск всех тестовых функций класса или модуля

Маркер `skipif` (так же, как и остальные маркеры) можно использовать для класса:

```
@pytest.mark.skipif(sys.platform == "win32", reason="does not run on windows")
class TestPosixCalls:
    def test_function(self):
        "will not be setup or run under 'win32' platform"
```

Если условие выполняется, маркер будет применен для каждого тестового метода класса.

Если вы хотите пропустить все тестовые функции модуля, вы можете использовать `pytestmark` на глобальном уровне:

```
# test_module.py
pytestmark = pytest.mark.skipif(...)
```

Когда к тестовой функции применяется несколько декораторов `skipif`, она будет пропущена, если верно любое из условий пропуска.

7.1.3 Пропуск файлов и директорий

Иногда может потребоваться пропустить весь файл или каталог, например, если тесты основаны на специфических для версии Python функциях или содержат код, который вы не хотите запускать с помощью `pytest`. В этом случае необходимо исключить файлы и каталоги из коллекции. Дополнительную информацию смотрите в разделе *Настройка поиска тестов*.

7.1.4 Пропуск тестов в зависимости от успешности импорта

Вы можете пропустить тесты в случае неудачного импорта, применив `pytest.importorskip` на уровне модуля, в рамках теста или «setup»-фикстуры:

```
docutils = pytest.importorskip("docutils")
```

В данном случае, если `docutils` не будет импортирован, то тест будет пропущен. Также можно пропустить тест в зависимости от версии импортируемой библиотеки:

```
docutils = pytest.importorskip("docutils", minversion="0.3")
```

При этом версия считывается из специального атрибута модуля `__version__`.

7.1.5 Краткая сводка

Вот краткая шпаргалка, как пропускать тесты в различных ситуациях:

1. Пропустить все тесты модуля:

```
pytestmark = pytest.mark.skip("all tests still WIP")
```

2. Пропустить все тесты модуля при выполнении какого-то условия:

```
pytestmark = pytest.mark.skipif(sys.platform == "win32", reason="tests for linux only")
```

3. Пропустить все тесты модуля при неудачном импорте:

```
pexpect = pytest.importorskip("pexpect")
```

7.2 XFail: маркируем тесты, которые должны упасть

Маркер `xfail` используется для пометки ожидаемо падающих тестов:

```
@pytest.mark.xfail
def test_function():
    ...
```

Такой тест будет запущен, но при падении не вызовет сообщения об ошибке. В отчете он будет помещен в раздел ожидаемых сбоев (`XFAIL`) или неожиданно прошедших (`XPASS`).

Маркировку `xfail` можно установить непосредственно в функции (или в «setup»-фикстуре):

```
def test_function():
    if not valid_config():
        pytest.xfail("failing configuration (but should work)")
```

```
def test_function2():
    import slow_module

    if slow_module.slow_function():
        pytest.xfail("slow_module taking too long")
```

Эти два примера иллюстрируют ситуации, в которых вы не хотите проверять условие на уровне модуля.

Обратите внимание, что в случае вызова `pytest.xfail` (в отличие от маркировки с помощью `@pytest.mark.xfail`) код, расположенный после этого вызова, выполняться не будет. Это происходит потому, что внутренне метод реализуется путем создания определенного исключения.

7.2.1 Параметр strict

Ни `XFAIL`, ни `XPASS` по умолчанию не приводят к падению всего набора тестов. Но это можно изменить, установив параметру `strict` значение `True`:

```
@pytest.mark.xfail(strict=True)
def test_function():
    ...
```

В этом случае, если тест будет неожиданно пройден (`XPASS`), то это приведет к падению всего тестового набора.

Значение по умолчанию параметра `strict` можно изменить в настройках (в файле“pytest.ini“), используя опцию `xfail_strict`:

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

7.2.2 Параметр `reason`

Так же, как и при использовании *skipif*, можно установить зависимость маркировки `xfail` от определенного условия:

```
@pytest.mark.xfail(sys.version_info >= (3, 6), reason="python3.6 api changes")
def test_function():
    ...
```

7.2.3 Параметр `raises`

Если вы хотите уточнить причину сбоя теста, вы можете указать одно исключение или кортеж исключений в параметре `raises`:

```
@pytest.mark.xfail(raises=RuntimeError)
def test_function():
    ...
```

В этом случае тест будет объявлен в отчете, как обычный сбой, если он не выполняется с исключением, упомянутом в параметре `raises`.

7.2.4 Параметр `run`

Если тест должен быть помечен и учитываться в отчете как маркированный `xfail`, но при этом даже не должен выполняться, можно установить параметр `run` в значение `False`:

```
@pytest.mark.xfail(run=False)
def test_function():
    ...
```

Это особенно полезно для тестов `xfail`, которые приводят к сбою интерпретатора и должны быть исследованы позже.

7.2.5 Игнорирование `xfail`

Используя параметр `--runxfail`, можно принудительно запускать и выполнять тесты, помеченные `xfail`, как обычные непомеченные тесты:

```
pytest --runxfail
```

В этом случае метод `pytest.xfail` также будет игнорироваться.

7.2.6 Примеры

Простой тест с несколькими примерами:

```
import pytest

xfail = pytest.mark.xfail


@xfail
def test_hello():
    assert 0


@xfail(run=False)
def test_hello2():
    assert 0


@xfail("hasattr(os, 'sep')")
def test_hello3():
    assert 0


@xfail(reason="bug 110")
def test_hello4():
    assert 0


@xfail('pytest.__version__[0] != "17"')
def test_hello5():
    assert 0


def test_hello6():
    pytest.xfail("reason")


@xfail(raises=IndexError)
def test_hello7():
    x = []
    x[1] = 1
```

Запустив его с параметром `-rx` (report-on-xfail), получим следующий отчет:

```
example $ pytest -rx xfail_demo.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/example
collected 7 items

```

(continues on next page)

(продолжение с предыдущей страницы)

```
xfail_demo.py xxxxxxx                                                    [100%]

======================== short test summary info ==========================
XFAIL xfail_demo.py::test_hello
XFAIL xfail_demo.py::test_hello2
  reason: [NOTRUN]
XFAIL xfail_demo.py::test_hello3
  condition: hasattr(os, 'sep')
XFAIL xfail_demo.py::test_hello4
  bug 110
XFAIL xfail_demo.py::test_hello5
  condition: pytest.__version__[0] != "17"
XFAIL xfail_demo.py::test_hello6
  reason: reason
XFAIL xfail_demo.py::test_hello7
============================ 7 xfailed in 0.12s ============================
```

7.3 Skip/xfail с параметризацией

При использовании параметризации можно маркировать `skip/xfail` отдельные экземпляры тестов:

```python
import pytest


@pytest.mark.parametrize(
    ("n", "expected"),
    [
        (1, 2),
        pytest.param(1, 0, marks=pytest.mark.xfail),
        pytest.param(1, 3, marks=pytest.mark.xfail(reason="some bug")),
        (2, 3),
        (3, 4),
        (4, 5),
        pytest.param(
            10, 11, marks=pytest.mark.skipif(sys.version_info >= (3, 0), reason="py2k")
        ),
    ],
)
def test_increment(n, expected):
    assert n + 1 == expected
```

Глава 8

Параметризация фикстур и тестовых функций

pytest обеспечивает параметризацию тестовых функций на нескольких уровнях:

- pytest.fixture позволяет *параметризовать фикстуры*;
- *@pytest.mark.parametrize* позволяет определить множество аргументов и фикстур для тестовой функции или класса;
- *pytest_generate_tests* позволяет определять пользовательские расширения и схемы параметризации.

8.1 @pytest.mark.parametrize: параметризация тестовых функций

Встроенный декоратор pytest.mark.parametrize позволяет параметризовать аргументы тестовых функций. Ниже приведен типичный пример тестовой функции, реализующей проверку того, что определенный ввод приводит к ожидаемому выводу:

```
# content of test_expectation.py
import pytest


@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Здесь декоратор `@parametrize` определяет три различных кортежа `(test_input, expected)`, так что функция `test_eval` будет работать три раза, используя их по очереди:

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 3 items
```

(continues on next page)

(продолжение с предыдущей страницы)

```
test_expectation.py ..F                                                  [100%]

================================= FAILURES =================================
____________________________ test_eval[6*9-42] _____________________________

test_input = '6*9', expected = 42

    @pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
    def test_eval(test_input, expected):
>       assert eval(test_input) == expected
E       AssertionError: assert 54 == 42
E        +  where 54 = eval('6*9')

test_expectation.py:6: AssertionError
======================== 1 failed, 2 passed in 0.12s ========================
```

Примечание: По умолчанию `pytest` экранирует любые не ASCII-символы, которые используются в строках unicode для параметризации. Если вы хотите использовать строки unicode в параметризации и видеть их в терминале как есть (без экранирования), пропишите в файле `pytest.ini` следующее:

```
[pytest]
disable_test_id_escaping_and_forfeit_all_rights_to_community_support = True
```

При этом имейте в виду, что в некоторых ОС и при установке некоторых плагинов такое использование может приводить к неожиданным побочным эффектам и даже ошибкам.

В примере выше только одна пара параметров приводит к падению теста. И, как обычно, в трассировке можно увидеть входные (`input`) и выходные (`output`) значения аргументов функции.

Обратите внимание, что маркер `parametrize` можно использовать также и для классов и модулей (см. *Маркировка тестов*) и это также приведет к вызову нескольких функций с разным набором аргументов.

Можно также помечать отдельные экземпляры теста внутри маркера параметризации. Вот пример использования параметризации совместно со встроенным `mark.xfail`:

```
# content of test_expectation.py
import pytest


@pytest.mark.parametrize(
    "test_input,expected",
    [("3+5", 8), ("2+4", 6), pytest.param("6*9", 42, marks=pytest.mark.xfail)],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Давайте запустим:

```
$ pytest
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
```

(continues on next page)

(продолжение с предыдущей страницы)

```
collected 3 items

test_expectation.py ..x                                                  [100%]

======================== 2 passed, 1 xfailed in 0.12s ========================
```

Тот набор параметров, который раньше вызывал сбой, теперь помечается как `xfailed` (ожидаемое падение).

Когда значения, передаваемые при параметризации, оказываются пустым списком - например, если они динамически генерируются некоторой функцией, - поведение `pytest` определяется опцией empty_parameter_set_mark.

Чтобы запустить тест со всеми комбинациями различных параметров, можно применить несколько маркеров параметризации:

```
import pytest


@pytest.mark.parametrize("x", [0, 1])
@pytest.mark.parametrize("y", [2, 3])
def test_foo(x, y):
    pass
```

Такая запись позволит выполнить тест со всеми комбинациями x и y: `x=0/y=2`, `x=1/y=2`, `x=0/y=3` и `x=1/y=3`; комбинации будут формироваться в порядке следования маркеров параметризации.

8.2 Базовый пример: pytest_generate_tests

Иногда вам может потребоваться реализовать собственную схему параметризации или некоторый динамизм для определения параметров или области применения фикстуры. Для этого можно использовать hook-функцию `pytest_generate_tests`, которая вызывается при сборке тестовой функции. Через переданный `metafunc`-объект можно запросить требуемый контекст тестов и, самое главное, можно вызвать `metafunc.parametrize()` для параметризации.

Давайте предположим, что мы хотим запустить тест с использованием строковых входных данных, которые нужно устанавливать с помощью новой опции командной строки `pytest`. Давайте сначала напишем простой тест, который принимает фикстуру `stringinput` в качестве аргумента:

```
# content of test_strings.py


def test_valid_string(stringinput):
    assert stringinput.isalpha()
```

Затем мы пропишем в файл `conftest.py` добавление опции командной строки и параметризацию нашей тестовой функции:

```
# content of conftest.py


def pytest_addoption(parser):
    parser.addoption(
        "--stringinput",
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        action="append",
        default=[],
        help="list of stringinputs to pass to test functions",
    )


def pytest_generate_tests(metafunc):
    if "stringinput" in metafunc.fixturenames:
        metafunc.parametrize("stringinput", metafunc.config.getoption("stringinput"))
```

Теперь, если мы передадим две входных строки, наш тест будет выполнен дважды:

```
$ pytest -q --stringinput="hello" --stringinput="world" test_strings.py
..                                                                   [100%]
2 passed in 0.12s
```

Давайте запустим тест со значением, которое должно привести к падению:

```
$ pytest -q --stringinput="!" test_strings.py
F                                                                    [100%]
================================ FAILURES ================================
___________________________ test_valid_string[!] ___________________________

stringinput = '!'

    def test_valid_string(stringinput):
>       assert stringinput.isalpha()
E       AssertionError: assert False
E        +  where False = <built-in method isalpha of str object at 0xdeadbeef>()
E        +    where <built-in method isalpha of str object at 0xdeadbeef> = '!'.isalpha

test_strings.py:4: AssertionError
1 failed in 0.12s
```

Как и ожидалось, наш тест упал.

Если при запуске вы не укажете строковое значение, то тест будет пропущен, поскольку функция `metafunc.parametrize()` будет вызвана с пустым списком параметров:

```
$ pytest -q -rs test_strings.py
s                                                                    [100%]
======================== short test summary info =========================
SKIPPED [1] test_strings.py: got empty parameter set ['stringinput'], function test_valid_string␣
↪at $REGENDOC_TMPDIR/test_strings.py:2
1 skipped in 0.12s
```

Обратите внимание, что при многократном вызове `metafunc.parametrize` с различными множествами параметров, имена параметров в множестве не должны дублироваться, иначе возникнет ошибка.

8.3 Еще примеры

Больше примеров можно увидеть здесь: *параметризация тестов*.

Глава 9

Классический «setup» в стиле xunit

Этот раздел описывает популярный классический способ реализации фикстур («setup/teardown» методов) на уровнях модулей/классов/функций.

Примечание: Поскольку эти методы просты и хорошо знакомы тем, кто уже использовал `unittest` or `nose background`, вы также можете рассмотреть применение более мощного *механизма фикстур*, который использует концепцию внедрения зависимостей, позволяя создавать более модальные и масштабируемые тесты, что особенно важно в больших проектах или при функциональном тестировании. В одном и том же файле можно использовать оба механизма, однако нужно иметь в виду, что подклассы `unittest.TestCase` не способны принимать аргументы-фикстуры.

9.1 «setup/teardown» для модуля

Если у вас есть несколько тестовых функций и классов в отдельном модуле, вы можете реализовать следующие методы-фикстуры, которые будут вызываться только один раз для всех функций модуля:

```
def setup_module(module):
    """настройка любых состояний, специфичных для выполнения этого модуля """


def teardown_module(module):
    """ очистка любых состояний, настроенных ранее с помощью метода
        setup_module
    """
```

Начиная с `pytest-3.0`, параметр `module` можно не указывать

9.2 «setup/teardown» для класса

Аналогичные методы существуют и на уровне класса - они будут вызваны до и после всех тестовых методов класса:

```
@classmethod
def setup_class(cls):
    """ настройка любых состояний, специфичных для выполнения этого класса (который
    обычно содержит тесты).
    """


@classmethod
def teardown_class(cls):
   """ очистка любых состояний, настроенных ранее с помощью метода
       setup_class.
   """
```

9.3 «setup/teardown» для функций и методов

Аналогично, следующими методами можно «обернуть» каждый вызов метода класса:

```
def setup_method(self, method):
    """ настройка любого состояния, связанного с выполнением этого метода класса.
    setup_method вызывается для каждого метода класса.
    """


def teardown_method(self, method):
    """ очистка любого состояния, настроенного ранее с помощью вызова setup_method
    """
```

Начиная с `pytest-3.0`, параметр `method` указывать не обязательно.

Если же вы предпрочитаете определять тестовые функции на уровне модуля (без разнесения по классам), для них тоже можно реализовать фикстуры:

```
def setup_function(function):
    """ настройка любого состояния, связанного с выполнением данной функции.
    Вызывается для каждой тестовой функции модуля.
    """


def teardown_function(function):
    """ очистка любого состояния, настроенного ранее с помощью вызова setup_function
    """
```

Начиная с `pytest-3.0`, параметр `function` указывать не обязательно.

Замечания:

- Пары «setup/teardown» во время тестирования могут вызываться многократно.
- Функция «teardown» не будет вызвана, если соответствующая «setup» функция существует, но была пропущена или выдала ошибку.

- Вплоть до `pytest-4.2`, для функций в стиле `xunit` не соблюдались правила задания области действия фикстур, поэтому, к примеру, могло случаться так, что `setup_method` вызывался до `autouse` фикстуры уровня сессии.

  Сейчас функции `xunit` интегрированы с механизмом фикстур, и для них применяются те же правила области действия, что и для вызова фикстур.

Глава 10

Рекомендации по интеграции

10.1 Установка пакета с помощью pip

При разработке мы рекомендуем использовать venv для создания виртуального окружения и pip для установки вашего приложения и любых других используемых пакетов (таких как сам `pytest`). Это гарантирует, что ваш код и окружение будут изолированы от вашей системной установки Python.

Затем поместите в корень вашего проекта файл `setup.py`, в котором ,как минимум, должен быть следующий код:

```
from setuptools import setup, find_packages

setup(name="PACKAGENAME", packages=find_packages())
```

Здесь `PACKAGENAME` - имя вашего пакета. Затем вы можете установить ваш пакет в режиме редактирования, запустив в той же директории:

```
pip install -e .
```

Это позволит вам менять код исходных файлов (как тестов, так и самого приложения) и перезапускать тесты по вашему желанию. Вы выполняете аналог `python setup.py develop` или `conda develop` и устанавливаете ваш пакет, используя символическую ссылку на разрабатываемый код.

10.2 Соглашения Python по поиску тестов

Стандартный поиск тестов `pytest` выполняется по следующим правилам:

- Если не переданы никакие параметры, поиск начинается в testpaths (если они настроены) или в текущей директории. Кроме того, можно использовать параметры командной строки с любой комбинацией директорий, имен файлов и идентификаторов узлов.
- Проводится рекурсивный поиск в директориях, если они не соответствуют шаблону norecursedirs.

- Ищутся файлы с именами «test_*.py» or «*_test.py», импортированные по *имени для импорта*.
- Из этих файлов выбираются:
  - функции и методы с префиксом «test», расположенные вне классов
  - тестовые функции и методы с префиксом «test» внутри классов с префиксом «Test» (без метода «__init__»)

Пример настройки поиска тестов: *Изменение стандартных правил поиска тестов Python*.

В модулях `Python` `pytest` для поиска тестов также использует стандартный механизм подклассов unittest.TestCase.

10.3 Выбор шаблона размещения и правила импорта тестов

`pytest` поддерживает два способа размещения тестов:

10.3.1 Тесты вне кода самого приложения

Размещение тестов в отдельном каталоге вне фактического кода приложения может быть полезно, если у вас много функциональных тестов или же по каким-то причинам вы хотите держать тесты отдельно от фактического кода приложения (зачастую это - хорошая идея):

```
setup.py
mypkg/
    __init__.py
    app.py
    view.py
tests/
    test_app.py
    test_view.py
    ...
```

Такой способ дает вам следующие преимущества:

- Вы можете тестировать установленную версию приложения после выполнения `pip install`.
- Вы можете тестировать локальную копию приложения в режиме редактирования после выполнения `pip install --editable`.
- Если у вас нет `setup.py`-файла и вы пользуетесь тем, что `Python` по умолчанию помещает текущий каталог в `sys.path` для импорта вашего пакета, вы можете выполнить `python -m pytest`, чтобы запустить тестирование локальной копии напрямую, без использования `pip`.

Примечание: См. pytest vs python -m pytest чтобы узнать больше о разнице между вызовом `pytest` и `python -m pytest`.

Обратите внимание, что при использовании такой схемы ваши тестовые файлы должны иметь **уникальные имена**, поскольку `pytest` просто добавляет директорию `tests/` в `sys.path` и полное имя файла будет представлять собой просто имя вашего модуля.

Если все же нужно запускать модули с одинаковыми именами, можно добавить файл `__init__.py` в папку `tests` и ее подпапки, разбив ваши тесты на пакеты:

```
setup.py
mypkg/
    ...
tests/
    __init__.py
    foo/
        __init__.py
        test_view.py
    bar/
        __init__.py
        test_view.py
```

В этом случае `pytest` позволяет использовать одинаковые имена для файлов, поскольку на самом деле будут запускаться файлы с именами `tests.foo.test_view` и `tests.bar.test_view`. Однако здесь есть одна тонкость: чтобы загрузить тестовые модули из директории `tests`, `pytest` прописывает корень репозитария в переменную `sys.path`, что может иметь некоторые побочные эффекты, так как директория `mypkg` тоже импортируется. Это может создавать проблемы, если вы используете для тестирования вашего пакета в виртуальной среде такие инструменты, как tox, поскольку вам нужно тестировать *установленную* версию пакета, а не локальный код репозитария.

В таких случаях **настоятельно** рекомендуется использовать шаблон `src`, когда корневой пакет приложения находится в поддиректории корневой директории:

```
setup.py
src/
    mypkg/
        __init__.py
        app.py
        view.py
tests/
    __init__.py
    foo/
        __init__.py
        test_view.py
    bar/
        __init__.py
        test_view.py
```

Использование такого шаблона помогает обойти множество подводных камней и имеет массу преимуществ, которые прекрасно освещены в замечательном блоге: blog post by Ionel Cristian Mărieș.

10.3.2 Тесты как часть кода приложения

Встроенные в пакет приложения тесты имеют смысл, если у вас есть прямая связь между тестами и модулями приложения, и если вы хотите поставлять тесты вместе с приложением.

```
setup.py
mypkg/
    __init__.py
    app.py
    view.py
    test/
        __init__.py
        test_app.py
        test_view.py
        ...
```

В этом случае проще всего запустить тесты, используя опцию `--pyargs`:

```
pytest --pyargs mypkg
```

`pytest` найдет, где установлен `mypkg` и соберет тесты оттуда.

Обратите внимание, что такой способ тоже работает с шаблоном `src`, упомянутом в предыдущем разделе.

Примечание: Вы можете использовать пространство имен Python3 (PEP420) для своего приложения, но `pytest` все равно будет искать *имя для импорта*, основываясь на наличии файлов «__init__.py». Если вы используете один из рекумендуемых способов формирования файловой системы, но не хотите использовать файлы «__init__.py» в ваших директориях, эта схема будет работать на версиях `Python3.3` и выше. Однако при «встроенных» тестах вам придется использовать абсолютный импорт, чтобы добраться до кода вашего приложения.

Примечание: Когда `pytest` во время рекурсивного обхода файловой системы находит файл вида «a/b/test_module.py», он определяет имя для импорта следующим образом:

- определяется `basedir`: это первый, самый высоко расположенный в корне каталог, в котором нет файла `__init__.py`. Т. е., если и `a`, и `b` содержат файлы `__init__.py`, то родительская директория `a` станет `basedir`.
- выполняется `sys.path.insert(0, basedir)` чтобы задать полное имя для импорта
- выполняется `import a.b.test_module`, где путь определяется преобразованием разделителей `/` в символы `.`. Это означает, что нужно сопоставлять именя файлов и директорий импортируемым именам напрямую.

Причина такой эволюции метода импорта заключается в том, что в крупных проектах тестовые модули могут импортировать друг друга, и получение канонического имени импорта помогает избежать сюрпризов (например, двойного импорта тестового модуля).

10.4 tox

Если работа выполнена и вы хотите убедиться, что готовый пакет проходит все тесты, можно обратить внимание на tox - инструмент автоматизации тестирования и его поддержку pytest. `tox` помогает настроить виртуальное окружение с заранее заданными зависимостями и затем запускать предварительно настроенную команду тестирования с различными опциями. `tox` тестирует установленный пакет, а не источник кода, тем самым помогая найти погрешности сборки.

Глава 11

Конфигурирование

11.1 Опции командной строки и настройки конфигурационного файла

Чтобы получить помощь по командной строке и настройкам конфигурационных файлов, используйте стандартную опцию `-h`:

```
pytest -h   # справка по опциям и настройкам конфигурационного файла
```

Будут отображены опции командной строки и конфигурационных файлов, которые зарегистрированы установленными плагинами.

11.2 Инициализация: определение корневой директории и файла инициализации

`pytest` определяет корневую директорию `rootdir` при каждом прогоне тестов в зависимости от параметров командной строки (уточняющих пути и файлы) и существования *инициализирующих* файлов (*ini-files*). Определенные таким образом `rootdir` и *ini-file* печатаются в заголовочной части при запуске `pytest`.

Для чего `pytest` использует `rootdir`:

- Для генерации идентификаторов узлов во время сборки тестов; каждому тесту сопоставляется уникальный *nodeid* относительно `rootdir`, который учитывает полный путь, имена класса и функции, а также значение параметризации (если есть).
- Используется плагинами для хранения специфичной для проекта/запуска тестов информации; к примеру, внутренний плагин cache создает в `rootdir` поддиректорию `.pytest_cache` для хранения состояний выполнения тестов.

`rootdir` **НЕ** используется для модификации `sys.path`/`PYTHONPATH` и не оказывает влияния на импорт модулей. Подробнее см. pythonpath.

Опцию командной строки `--rootdir=path` указывают для принудительной установки определенного каталога. Передаваемый каталог может включать переменные окружения, если используется совместно с `addopts` из файла `pytest.ini`.

11.2.1 Алгоритм нахождения `rootdir`

Вот алгоритм поиска корневого каталога из `args`:

- определяем общий родительский каталог для переданных аргументов, которые распознаны в качестве существующих путей файловой системы. Если таких аргументов-путей нет, то текущей рабочей директорией становится общий родительский каталог;
- ищем файлы `pytest.ini`, `tox.ini` и `setup.cfg` в каталоге-предке и выше. Если находим - он становится ini-файлом, а его каталог становится `rootdir`.
- если ini-файлы не найдены, для определения `rootdir` ищем файл `setup.py` вверх от общего каталога-предка.
- если файл `setup.py` не найден, ищем файлы `pytest.ini`, `tox.ini` и `setup.cfg` в каждом указанном аргументе `args` и выше. Если нашли - он становится ini-файлом, а его директория становится корневой.
- если вообще никаких ini-файлов не найдено, то в качестве `rootdir` используется уже определенный общий каталог-предок. Это позволяет использовать `pytest` в стуктурах, которые не являются частью пакета и не имеют никакой конкретной конфигурации.

Если никакие `args` не указаны, `pytest` начинает определение `rootdir` из текущего рабочего каталога и оттуда же собирает тесты.

Предупреждение: В настраиваемых плагинах `pytest` аргументы командной строки могут включать путь, как в `pytest --log-output ../../test.log args`. В этом случае передавать `args` обязательно, в противном случае `pytest` использует для определения корневой директории папку test.log (также см. issue 1435). Для ссылки на текущий рабочий каталог можно использовать точку `"."`.

Обратите внимание, что существующий файл `pytest.ini` всегда будет считаться ini-файлом, тогда как `tox.ini` и `setup.cfg` файлы должны содержать секции `[pytest]` или `[tool:pytest]` соответственно. Опции из разных кандидатов в ini-файлы никогда не сливаются - побеждает первый найденный (при этом `pytest.ini` побеждает всегда, даже если в нем нет секции `[pytest]`).

Затем объект `config` приобретает следующие атрибуты:

- `config.rootdir`: определенная корневая директория, гарантированно существует.
- `config.inifile`: определенный ini-файл, может быть и `None`.

`rootdir` используется для генерации идентификаторов узлов тестов («nodeids»). Ее также могут использовать плагины для хранения установочной информации.

Пример:

```
pytest path/to/testdir path/other/
```

здесь определяется общий предок `path` и потом ini-файлы ищутся следующим образом:

```
# первый круг поиска файлов pytest.ini
path/pytest.ini
path/tox.ini    # должен содержать секцию [pytest]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
path/setup.cfg  # должен содержать секцию [tool:pytest]
pytest.ini
... # все пути вплоть до корня

# теперь ищем setup.py
path/setup.py
setup.py
... # все пути вплоть до корня
```

11.3 Как изменить опции командной строки по умолчанию

Печатать одни и те жа параметры командной строки каждый раз при запуске `pytest` может быть утомительно. К примеру, вы всегда хотели бы видеть подробную информацию и пропущенных и «xfail» тестах и лаконичную «точку» для прошедших. Тогда вы можете написать в конфигурационном файле:

```
# content of pytest.ini or tox.ini
[pytest]
addopts = -ra -q

# content of setup.cfg
[tool:pytest]
addopts = -ra -q
```

Кроме того, можно установить переменную окружения `PYTEST_ADDOPTS`, чтобы добавить параметры командной строки для использования в этой среде:

```
export PYTEST_ADDOPTS="-v"
```

Вот как конструируется командная строка при наличии `addopts` или переменных окружения:

```
<pytest.ini:addopts> $PYTEST_ADDOPTS <extra command-line arguments>
```

Так что если пользователь напишет в командной строке:

```
pytest -m slow
```

фактически будет выполнена команда:

```
pytest -ra -q -v -m slow
```

Обратите внимание, что как и в других интерпретируемых приложениях, если параметры конфликтуют друг с другом - применяется последний. Поэтому в примере выше будет показан подробный вывод, поскольку опция `-v` перезапишет опцию `-q`.

11.4 Встроенные параметры конфигурационного файла

Полный лист возможных настроек конфигурационного файла см. reference documentation.

Глава 12

Примеры и приемы настройки

Вот расширяемый список примеров. Свяжитесь с разработчиками, если вам нужны еще примеры или у вас есть вопросы. Можно также посмотреть *документацию*, в которой тоже много примеров. Ответы на вопросы и примеры кода можно также поискать на ресурсе pytest on stackoverflow.com.

Базовые примеры смотри здесь:

- *Установка и запуск* основные примеры
- *Проверка с помощью оператора assert* примеры использования `assert`
- *Фикстуры pytest: явные, модальные, расширяемые* примеры фикстур/ «setup»-функций
- *Параметризация фикстур и тестовых функций* примеры параметризации тестовых функций
- ../unittest примеры интеграции с «unittest»
- ../nose примеры интеграции с «nosetests»

Следующие примеры освещают различные варианты, которые вам могут пригодиться:

12.1 Python: примеры отчетов об ошибках `pytest`

Вот пример нескольких прогонов тестов и того, как `pytest` все это выводит:

```
assertion $ pytest failure_demo.py
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/assertion
collected 44 items

failure_demo.py FFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFFF         [100%]

================================= FAILURES =================================
___________________________ test_generative[3-6] ___________________________
```

(continues on next page)

(продолжение с предыдущей страницы)

```
param1 = 3, param2 = 6

    @pytest.mark.parametrize("param1, param2", [(3, 6)])
    def test_generative(param1, param2):
>       assert param1 * 2 < param2
E       assert (3 * 2) < 6

failure_demo.py:20: AssertionError
_________________________ TestFailing.test_simple __________________________

self = <failure_demo.TestFailing object at 0xdeadbeef>

    def test_simple(self):
        def f():
            return 42

        def g():
            return 43

>       assert f() == g()
E       assert 42 == 43
E        +  where 42 = <function TestFailing.test_simple.<locals>.f at 0xdeadbeef>()
E        +  and   43 = <function TestFailing.test_simple.<locals>.g at 0xdeadbeef>()

failure_demo.py:31: AssertionError
____________________ TestFailing.test_simple_multiline _____________________

self = <failure_demo.TestFailing object at 0xdeadbeef>

    def test_simple_multiline(self):
>       otherfunc_multi(42, 6 * 9)

failure_demo.py:34:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

a = 42, b = 54

    def otherfunc_multi(a, b):
>       assert a == b
E       assert 42 == 54

failure_demo.py:15: AssertionError
___________________________ TestFailing.test_not ___________________________

self = <failure_demo.TestFailing object at 0xdeadbeef>

    def test_not(self):
        def f():
            return 42

>       assert not f()
E       assert not 42
E        +  where 42 = <function TestFailing.test_not.<locals>.f at 0xdeadbeef>()

failure_demo.py:40: AssertionError
```

(continues on next page)

(продолжение с предыдущей страницы)

```
_____________________ TestSpecialisedExplanations.test_eq_text _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_text(self):
>       assert "spam" == "eggs"
E       AssertionError: assert 'spam' == 'eggs'
E         - eggs
E         + spam

failure_demo.py:45: AssertionError
_____________ TestSpecialisedExplanations.test_eq_similar_text _____________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_similar_text(self):
>       assert "foo 1 bar" == "foo 2 bar"
E       AssertionError: assert 'foo 1 bar' == 'foo 2 bar'
E         - foo 2 bar
E         ?     ^
E         + foo 1 bar
E         ?     ^

failure_demo.py:48: AssertionError
____________ TestSpecialisedExplanations.test_eq_multiline_text ____________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_multiline_text(self):
>       assert "foo\nspam\nbar" == "foo\neggs\nbar"
E       AssertionError: assert 'foo\nspam\nbar' == 'foo\neggs\nbar'
E           foo
E         - eggs
E         + spam
E           bar

failure_demo.py:51: AssertionError
______________ TestSpecialisedExplanations.test_eq_long_text _______________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_long_text(self):
        a = "1" * 100 + "a" + "2" * 100
        b = "1" * 100 + "b" + "2" * 100
>       assert a == b
E       AssertionError: assert '111111111111...2222222222222' == '111111111111...2222222222222'
E         Skipping 90 identical leading characters in diff, use -v to show
E         Skipping 91 identical trailing characters in diff, use -v to show
E         - 1111111111b222222222
E         ?           ^
E         + 1111111111a222222222
E         ?           ^

failure_demo.py:56: AssertionError
_________ TestSpecialisedExplanations.test_eq_long_text_multiline __________
```

(continues on next page)

(продолжение с предыдущей страницы)

```
self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_long_text_multiline(self):
        a = "1\n" * 100 + "a" + "2\n" * 100
        b = "1\n" * 100 + "b" + "2\n" * 100
>       assert a == b
E       AssertionError: assert '1\n1\n1\n1\n...n2\n2\n2\n2\n' == '1\n1\n1\n1\n...n2\n2\n2\n2\n'
E         Skipping 190 identical leading characters in diff, use -v to show
E         Skipping 191 identical trailing characters in diff, use -v to show
E           1
E           1
E           1
E           1
E           1...
E
E         ...Full output truncated (7 lines hidden), use '-vv' to show

failure_demo.py:61: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_list _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

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

failure_demo.py:64: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_list_long _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_list_long(self):
        a = [0] * 100 + [1] + [3] * 100
        b = [0] * 100 + [2] + [3] * 100
>       assert a == b
E       assert [0, 0, 0, 0, 0, 0, ...] == [0, 0, 0, 0, 0, 0, ...]
E         At index 100 diff: 1 != 2
E         Use -v to get the full diff

failure_demo.py:69: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_dict _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_dict(self):
>       assert {"a": 0, "b": 1, "c": 0} == {"a": 0, "b": 2, "d": 0}
E       AssertionError: assert {'a': 0, 'b': 1, 'c': 0} == {'a': 0, 'b': 2, 'd': 0}
E         Omitting 1 identical items, use -vv to show
E         Differing items:
E         {'b': 1} != {'b': 2}
E         Left contains 1 more item:
E         {'c': 0}
E         Right contains 1 more item:
E         {'d': 0}...
```

(continues on next page)

(продолжение с предыдущей страницы)

```
E
E         ...Full output truncated (2 lines hidden), use '-vv' to show

failure_demo.py:72: AssertionError
_____________________ TestSpecialisedExplanations.test_eq_set ____________________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_set(self):
>       assert {0, 10, 11, 12} == {0, 20, 21}
E       AssertionError: assert {0, 10, 11, 12} == {0, 20, 21}
E         Extra items in the left set:
E         10
E         11
E         12
E         Extra items in the right set:
E         20
E         21...
E
E         ...Full output truncated (2 lines hidden), use '-vv' to show

failure_demo.py:75: AssertionError
_____________ TestSpecialisedExplanations.test_eq_longer_list _____________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_longer_list(self):
>       assert [1, 2] == [1, 2, 3]
E       assert [1, 2] == [1, 2, 3]
E         Right contains one more item: 3
E         Use -v to get the full diff

failure_demo.py:78: AssertionError
_____________________ TestSpecialisedExplanations.test_in_list _____________________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_in_list(self):
>       assert 1 in [0, 2, 3, 4, 5]
E       assert 1 in [0, 2, 3, 4, 5]

failure_demo.py:81: AssertionError
__________ TestSpecialisedExplanations.test_not_in_text_multiline __________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_not_in_text_multiline(self):
        text = "some multiline\ntext\nwhich\nincludes foo\nand a\ntail"
>       assert "foo" not in text
E       AssertionError: assert 'foo' not in 'some multil...nand a\ntail'
E         'foo' is contained here:
E           some multiline
E           text
E           which
E           includes foo
E         ?          +++
```

(continues on next page)

(продолжение с предыдущей страницы)

```
E           and a...
E
E         ...Full output truncated (2 lines hidden), use '-vv' to show

failure_demo.py:85: AssertionError
___________ TestSpecialisedExplanations.test_not_in_text_single ____________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_not_in_text_single(self):
        text = "single foo line"
>       assert "foo" not in text
E       AssertionError: assert 'foo' not in 'single foo line'
E         'foo' is contained here:
E           single foo line
E         ?        +++

failure_demo.py:89: AssertionError
_________ TestSpecialisedExplanations.test_not_in_text_single_long _________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_not_in_text_single_long(self):
        text = "head " * 50 + "foo " + "tail " * 20
>       assert "foo" not in text
E       AssertionError: assert 'foo' not in 'head head h...l tail tail '
E         'foo' is contained here:
E           head head foo tail tail tail tail tail tail tail tail tail tail tail tail tail␣
↪tail tail tail tail tail tail
E         ?           +++

failure_demo.py:93: AssertionError
______ TestSpecialisedExplanations.test_not_in_text_single_long_term _______

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_not_in_text_single_long_term(self):
        text = "head " * 50 + "f" * 70 + "tail " * 20
>       assert "f" * 70 not in text
E       AssertionError: assert 'fffffffffff...fffffffffff' not in 'head head h...l tail tail '
E         'fffffffffffffffff...fffffffffffffffff' is contained here:
E           head head fffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffffftail␣
↪tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail tail
E         ?           ++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++++

failure_demo.py:97: AssertionError
_____________ TestSpecialisedExplanations.test_eq_dataclass _____________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_dataclass(self):
        from dataclasses import dataclass

        @dataclass
        class Foo:
            a: int
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            b: str

        left = Foo(1, "b")
        right = Foo(1, "c")
>       assert left == right
E       AssertionError: assert TestSpecialis...oo(a=1, b='b') == TestSpecialis...oo(a=1, b='c')
E         Omitting 1 identical items, use -vv to show
E         Differing attributes:
E         b: 'b' != 'c'

failure_demo.py:109: AssertionError
_____________ TestSpecialisedExplanations.test_eq_attrs _____________

self = <failure_demo.TestSpecialisedExplanations object at 0xdeadbeef>

    def test_eq_attrs(self):
        import attr

        @attr.s
        class Foo:
            a = attr.ib()
            b = attr.ib()

        left = Foo(1, "b")
        right = Foo(1, "c")
>       assert left == right
E       AssertionError: assert Foo(a=1, b='b') == Foo(a=1, b='c')
E         Omitting 1 identical items, use -vv to show
E         Differing attributes:
E         b: 'b' != 'c'

failure_demo.py:121: AssertionError
_____________________________ test_attribute ______________________________

    def test_attribute():
        class Foo:
            b = 1

        i = Foo()
>       assert i.b == 2
E       assert 1 == 2
E        +  where 1 = <failure_demo.test_attribute.<locals>.Foo object at 0xdeadbeef>.b

failure_demo.py:129: AssertionError
_________________________ test_attribute_instance __________________________

    def test_attribute_instance():
        class Foo:
            b = 1

>       assert Foo().b == 2
E       AssertionError: assert 1 == 2
E        +  where 1 = <failure_demo.test_attribute_instance.<locals>.Foo object at 0xdeadbeef>.b
E        +    where <failure_demo.test_attribute_instance.<locals>.Foo object at 0xdeadbeef> =
↪<class 'failure_demo.test_attribute_instance.<locals>.Foo'>()
```

(continues on next page)

(продолжение с предыдущей страницы)

```
failure_demo.py:136: AssertionError
___________________________ test_attribute_failure ___________________________

    def test_attribute_failure():
        class Foo:
            def _get_b(self):
                raise Exception("Failed to get attrib")

            b = property(_get_b)

        i = Foo()
>       assert i.b == 2

failure_demo.py:147:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

self = <failure_demo.test_attribute_failure.<locals>.Foo object at 0xdeadbeef>

    def _get_b(self):
>       raise Exception("Failed to get attrib")
E       Exception: Failed to get attrib

failure_demo.py:142: Exception
__________________________ test_attribute_multiple __________________________

    def test_attribute_multiple():
        class Foo:
            b = 1

        class Bar:
            b = 2

>       assert Foo().b == Bar().b
E       AssertionError: assert 1 == 2
E        +  where 1 = <failure_demo.test_attribute_multiple.<locals>.Foo object at 0xdeadbeef>.b
E        +    where <failure_demo.test_attribute_multiple.<locals>.Foo object at 0xdeadbeef> =
↪<class 'failure_demo.test_attribute_multiple.<locals>.Foo'>()
E        +  and   2 = <failure_demo.test_attribute_multiple.<locals>.Bar object at 0xdeadbeef>.b
E        +    where <failure_demo.test_attribute_multiple.<locals>.Bar object at 0xdeadbeef> =
↪<class 'failure_demo.test_attribute_multiple.<locals>.Bar'>()

failure_demo.py:157: AssertionError
___________________________ TestRaises.test_raises ___________________________

self = <failure_demo.TestRaises object at 0xdeadbeef>

    def test_raises(self):
        s = "qwe"
>       raises(TypeError, int, s)
E       ValueError: invalid literal for int() with base 10: 'qwe'

failure_demo.py:167: ValueError
______________________ TestRaises.test_raises_doesnt _______________________

self = <failure_demo.TestRaises object at 0xdeadbeef>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def test_raises_doesnt(self):
>       raises(IOError, int, "3")
E       Failed: DID NOT RAISE <class 'OSError'>

failure_demo.py:170: Failed
_________________________ TestRaises.test_raise __________________________

self = <failure_demo.TestRaises object at 0xdeadbeef>

    def test_raise(self):
>       raise ValueError("demo error")
E       ValueError: demo error

failure_demo.py:173: ValueError
________________________ TestRaises.test_tupleerror ________________________

self = <failure_demo.TestRaises object at 0xdeadbeef>

    def test_tupleerror(self):
>       a, b = [1]  # NOQA
E       ValueError: not enough values to unpack (expected 2, got 1)

failure_demo.py:176: ValueError
______ TestRaises.test_reinterpret_fails_with_print_for_the_fun_of_it ______

self = <failure_demo.TestRaises object at 0xdeadbeef>

    def test_reinterpret_fails_with_print_for_the_fun_of_it(self):
        items = [1, 2, 3]
        print("items is {!r}".format(items))
>       a, b = items.pop()
E       TypeError: 'int' object is not iterable

failure_demo.py:181: TypeError
--------------------------- Captured stdout call ---------------------------
items is [1, 2, 3]
________________________ TestRaises.test_some_error ________________________

self = <failure_demo.TestRaises object at 0xdeadbeef>

    def test_some_error(self):
>       if namenotexi:  # NOQA
E       NameError: name 'namenotexi' is not defined

failure_demo.py:184: NameError
____________________ test_dynamic_compile_shows_nicely _____________________

    def test_dynamic_compile_shows_nicely():
        import importlib.util
        import sys

        src = "def foo():\n assert 1 == 0\n"
        name = "abc-123"
        spec = importlib.util.spec_from_loader(name, loader=None)
        module = importlib.util.module_from_spec(spec)
        code = _pytest._code.compile(src, name, "exec")
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        exec(code, module.__dict__)
        sys.modules[name] = module
>       module.foo()

failure_demo.py:203:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

    def foo():
>    assert 1 == 0
E    AssertionError

<0-codegen 'abc-123' $REGENDOC_TMPDIR/assertion/failure_demo.py:200>:2: AssertionError
____________________ TestMoreErrors.test_complex_error ____________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_complex_error(self):
        def f():
            return 44

        def g():
            return 43

>       somefunc(f(), g())

failure_demo.py:214:
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -
failure_demo.py:11: in somefunc
    otherfunc(x, y)
- - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - - -

a = 44, b = 43

    def otherfunc(a, b):
>       assert a == b
E       assert 44 == 43

failure_demo.py:7: AssertionError
____________________ TestMoreErrors.test_z1_unpack_error ____________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_z1_unpack_error(self):
        items = []
>       a, b = items
E       ValueError: not enough values to unpack (expected 2, got 0)

failure_demo.py:218: ValueError
____________________ TestMoreErrors.test_z2_type_error ____________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_z2_type_error(self):
        items = 3
>       a, b = items
E       TypeError: 'int' object is not iterable
```

(continues on next page)

(продолжение с предыдущей страницы)

```
failure_demo.py:222: TypeError
_____________________ TestMoreErrors.test_startswith _____________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_startswith(self):
        s = "123"
        g = "456"
>       assert s.startswith(g)
E       AssertionError: assert False
E        +  where False = <built-in method startswith of str object at 0xdeadbeef>('456')
E        +    where <built-in method startswith of str object at 0xdeadbeef> = '123'.startswith

failure_demo.py:227: AssertionError
_____________________ TestMoreErrors.test_startswith_nested ______________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_startswith_nested(self):
        def f():
            return "123"

        def g():
            return "456"

>       assert f().startswith(g())
E       AssertionError: assert False
E        +  where False = <built-in method startswith of str object at 0xdeadbeef>('456')
E        +    where <built-in method startswith of str object at 0xdeadbeef> = '123'.startswith
E        +      where '123' = <function TestMoreErrors.test_startswith_nested.<locals>.f at␣
↪0xdeadbeef>()
E        +    and   '456' = <function TestMoreErrors.test_startswith_nested.<locals>.g at␣
↪0xdeadbeef>()

failure_demo.py:236: AssertionError
_____________________ TestMoreErrors.test_global_func ______________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_global_func(self):
>       assert isinstance(globf(42), float)
E       assert False
E        +  where False = isinstance(43, float)
E        +    where 43 = globf(42)

failure_demo.py:239: AssertionError
_____________________ TestMoreErrors.test_instance ______________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_instance(self):
        self.x = 6 * 7
>       assert self.x != 42
E       assert 42 != 42
E        +  where 42 = <failure_demo.TestMoreErrors object at 0xdeadbeef>.x
```

(continues on next page)

(продолжение с предыдущей страницы)

```
failure_demo.py:243: AssertionError
_____________________ TestMoreErrors.test_compare ____________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_compare(self):
>       assert globf(10) < 5
E       assert 11 < 5
E        +  where 11 = globf(10)

failure_demo.py:246: AssertionError
_____________________ TestMoreErrors.test_try_finally ____________________

self = <failure_demo.TestMoreErrors object at 0xdeadbeef>

    def test_try_finally(self):
        x = 1
        try:
>           assert x == 0
E           assert 1 == 0

failure_demo.py:251: AssertionError
___________________ TestCustomAssertMsg.test_single_line ___________________

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    def test_single_line(self):
        class A:
            a = 1

        b = 2
>       assert A.a == b, "A.a appears not to be b"
E       AssertionError: A.a appears not to be b
E       assert 1 == 2
E        +  where 1 = <class 'failure_demo.TestCustomAssertMsg.test_single_line.<locals>.A'>.a

failure_demo.py:262: AssertionError
___________________ TestCustomAssertMsg.test_multiline ____________________

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    def test_multiline(self):
        class A:
            a = 1

        b = 2
>       assert (
            A.a == b
        ), "A.a appears not to be b\nor does not appear to be b\none of those"
E       AssertionError: A.a appears not to be b
E         or does not appear to be b
E         one of those
E       assert 1 == 2
E        +  where 1 = <class 'failure_demo.TestCustomAssertMsg.test_multiline.<locals>.A'>.a
```

(continues on next page)

(продолжение с предыдущей страницы)

```
failure_demo.py:269: AssertionError
_____________________ TestCustomAssertMsg.test_custom_repr ____________________

self = <failure_demo.TestCustomAssertMsg object at 0xdeadbeef>

    def test_custom_repr(self):
        class JSON:
            a = 1

            def __repr__(self):
                return "This is JSON\n{\n  'foo': 'bar'\n}"

        a = JSON()
        b = 2
>       assert a.a == b, a
E       AssertionError: This is JSON
E         {
E           'foo': 'bar'
E         }
E       assert 1 == 2
E        +  where 1 = This is JSON\n{\n  'foo': 'bar'\n}.a

failure_demo.py:282: AssertionError
============================ 44 failed in 0.12s ============================
```

12.2 Основные шаблоны и примеры

12.2.1 Передача значений тестовой функции с помощью опций командной строки

Предположим, что мы хотим написать тест, который зависит от опции командной строки. Вот как это делается:

```
# content of test_sample.py
def test_answer(cmdopt):
    if cmdopt == "type1":
        print("first")
    elif cmdopt == "type2":
        print("second")
    assert 0  # чтобы увидеть вывод
```

Чтобы это работало, нам нужно добавить опцию командной строки и представить `cmdopt` через *фикстуру*:

```
# content of conftest.py
import pytest


def pytest_addoption(parser):
    parser.addoption(
        "--cmdopt", action="store", default="type1", help="my option: type1 or type2"
    )
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@pytest.fixture
def cmdopt(request):
    return request.config.getoption("--cmdopt")
```

Давайте запустим БЕЗ нашей новой опции:

```
$ pytest -q test_sample.py
F                                                                    [100%]
================================= FAILURES =================================
_______________________________ test_answer ________________________________

cmdopt = 'type1'

    def test_answer(cmdopt):
        if cmdopt == "type1":
            print("first")
        elif cmdopt == "type2":
            print("second")
>       assert 0  # to see what was printed
E       assert 0

test_sample.py:6: AssertionError
--------------------------- Captured stdout call ---------------------------
first
1 failed in 0.12s
```

А теперь применим `cmdopt`:

```
$ pytest -q --cmdopt=type2
F                                                                    [100%]
================================= FAILURES =================================
_______________________________ test_answer ________________________________

cmdopt = 'type2'

    def test_answer(cmdopt):
        if cmdopt == "type1":
            print("first")
        elif cmdopt == "type2":
            print("second")
>       assert 0  # to see what was printed
E       assert 0

test_sample.py:6: AssertionError
--------------------------- Captured stdout call ---------------------------
second
1 failed in 0.12s
```

Как видите, значение опции появилось в нашем тесте. Это основной шаблон. Однако скорее всего захочется обрабатывать опцию вне тестов, а так же передавать ее различным или более сложным объектам.

12.2.2 Динамическое добавлений опций командной строки

С помощью addopts ref можно статически добавить опцию командной строки для вашего проекта. Можно также динамически модифицировать аргументы командной строки перед их обработкой:

```
# setuptools plugin
import sys


def pytest_load_initial_conftests(args):
    if "xdist" in sys.modules:  # pytest-xdist plugin
        import multiprocessing

        num = max(multiprocessing.cpu_count() / 2, 1)
        args[:] = ["-n", str(num)] + args
```

Если у вас установлен xdist plugin, то теперь вы будете всегда прогонять тесты с использованием числа подпроцессов, близкого к параметрам вашего процессора. Запустим в пустой директории с нашим `conftest.py`:

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 0 items

========================== no tests ran in 0.12s ==========================
```

12.2.3 Конролируем пропуск тестов с помощью опции командной строки

Добавим в файл `conftest.py` опцию `--runslow`, чтобы контролировать пропуск тестов с пометкой `pytest.mark.slow`:

```
# content of conftest.py

import pytest


def pytest_addoption(parser):
    parser.addoption(
        "--runslow", action="store_true", default=False, help="run slow tests"
    )


def pytest_configure(config):
    config.addinivalue_line("markers", "slow: mark test as slow to run")


def pytest_collection_modifyitems(config, items):
    if config.getoption("--runslow"):
        #  опция --runslow запрошена в командной строке: медленные тесты не пропускаем
        return
    skip_slow = pytest.mark.skip(reason="need --runslow option to run")
    for item in items:
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        if "slow" in item.keywords:
            item.add_marker(skip_slow)
```

Можно теперь написать тестовый модуль:

```
# content of test_module.py
import pytest


def test_func_fast():
    pass


@pytest.mark.slow
def test_func_slow():
    pass
```

Если запустим его, то «медленный» тест будет пропущен:

```
$ pytest -rs    # "-rs" means report details on the little 's'
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py .s                                                   [100%]

========================= short test summary info ==========================
SKIPPED [1] test_module.py:8: need --runslow option to run
====================== 1 passed, 1 skipped in 0.12s =======================
```

А теперь запустим и медленные тесты, применив нашу опцию `--runslow`:

```
$ pytest --runslow
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py ..                                                   [100%]

=========================== 2 passed in 0.12s ===========================
```

12.2.4 Настройка `__tracebackhide__`

Если у вас есть вспомогательная функция, которую вы используете в тесте, то можно использовать маркер `pytest.fail`, чтобы «уронить» тест с определенным сообщением. Вспомогательная функция не будет отображаться в трейсбэке, если вы примените опцию `__tracebackhide__` где-нибудь в теле этой функции. Пример:

```
# content of test_checkconfig.py
import pytest
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def checkconfig(x):
    __tracebackhide__ = True
    if not hasattr(x, "config"):
        pytest.fail("not configured: {}".format(x))


def test_something():
    checkconfig(42)
```

Настройка `__tracebackhide__` влияет на то, ЧТО `pytest` выводит в трейсбэке: функция `checkconfig` не будет показана, пока в командной строке не будет применена опция `--full-trace`. Давайте запустим наш маленький тест:

```
$ pytest -q test_checkconfig.py
F                                                                    [100%]
================================= FAILURES =================================
______________________________ test_something ______________________________

    def test_something():
>       checkconfig(42)
E       Failed: not configured: 42

test_checkconfig.py:11: Failed
1 failed in 0.12s
```

Если вы хотите скрыть только определенные исключения, можно сопоставить `__tracebackhide__` объект, который, в свою очередь, вернет объект `ExceptionInfo`. Это можно использовать, к примеру, для того чтобы убедиться, что неожиданные исключения будут отображены:

```
import operator
import pytest


class ConfigException(Exception):
    pass


def checkconfig(x):
    __tracebackhide__ = operator.methodcaller("errisinstance", ConfigException)
    if not hasattr(x, "config"):
        raise ConfigException("not configured: {}".format(x))


def test_something():
    checkconfig(42)
```

Такое решение позволит избежать скрытия в трассировке неожиданных исключений.

12.2.5 Как определить, запущено ли приложение из `pytest`

Вообще-то, заставлять приложение вести себя по-другому при тестировании - плохая идея. Но если уж совершенно необходимо выяснить, запускается ли приложение из теста или нет, можно сделать как-то так:

```
# content of your_module.py


_called_from_test = False
```

```
# content of conftest.py


def pytest_configure(config):
    your_module._called_from_test = True
```

И потом проверять флажок `your_module._called_from_test`:

```
if your_module._called_from_test:
    # запущено для тестирования
    ...
else:
    # запущено "нормально"
    ...
```

в самом приложении

12.2.6 Добавление информации к заголовку отчета

Добавить дополнительную информацию к запуску `pytest` легко:

```
# content of conftest.py


def pytest_report_header(config):
    return "project deps: mylib-1.1"
```

При выводе эта строка отобразится в заголовке:

```
$ pytest
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
project deps: mylib-1.1
rootdir: $REGENDOC_TMPDIR
collected 0 items

========================== no tests ran in 0.12s ==========================
```

Можно возвращать список строк - для каждого элемента списка будет добавлена отдельная строка. Можно также рассмотреть `config.getoption('verbose')` для получения подробной информации:

```
# content of conftest.py


def pytest_report_header(config):
    if config.getoption("verbose") > 0:
        return ["info1: did you know that ...", "did you?"]
```

Эта строка будет добавлена только при использовании опции `--v`:

```
$ pytest -v
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
info1: did you know that ...
did you?
rootdir: $REGENDOC_TMPDIR
collecting ... collected 0 items

============================ no tests ran in 0.12s =============================
```

А без этой опции вывод не изменится:

```
$ pytest
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 0 items

============================ no tests ran in 0.12s =============================
```

12.2.7 Определение продолжительности выполнения тестов

Если у вас есть медленно выполняющийся огромный набор тестов, то может возникнуть желание выяснить, какие тесты самые медленные. Давайте создадим искусственный тестовый набор:

```
# content of test_some_are_slow.py
import time


def test_funcfast():
    time.sleep(0.1)


def test_funcslow1():
    time.sleep(0.2)


def test_funcslow2():
    time.sleep(0.3)
```

Теперь мы можем выяснить продолжительность трех самых медленных тестов с помощью `--durations=3`:

```
$ pytest --durations=3
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 3 items

test_some_are_slow.py ...                                            [100%]

========================= slowest 3 test durations =========================
```

(continues on next page)

(продолжение с предыдущей страницы)

```
0.30s call     test_some_are_slow.py::test_funcslow2
0.20s call     test_some_are_slow.py::test_funcslow1
0.11s call     test_some_are_slow.py::test_funcfast
============================ 3 passed in 0.12s =============================
```

12.2.8 Тестирование по шагам (incremental testing)

Иногда тесты могут состоять из нескольких серий, и выполнять их надо по шагам. Если на каком-то шаге тест упал, нет смысла выполнять следующие шаги этой серии, поскольку они в любом случае должны упасть и трейсбэк не пополнится никакой полезной информацией. Ниже - пример файла `conftest.py`, который вводдит маркер `incremental` для использования с классами:

```
# content of conftest.py

# сохраняем историю падений в разрезе имен классов и индексов в параметризации (если она␣
↪используется)
_test_failed_incremental: Dict[str, Dict[Tuple[int, ...], str]] = {}


def pytest_runtest_makereport(item, call):
    if "incremental" in item.keywords:
        # используется маркер incremental
        if call.excinfo is not None:
            # тест упал
            # извлекаем из теста имя класса
            cls_name = str(item.cls)
            # извлекаем индексы теста (если вместе с  incremental используется параметризация)
            parametrize_index = (
                tuple(item.callspec.indices.values())
                if hasattr(item, "callspec")
                else ()
            )
            # извлекаем имя тестовой функции
            test_name = item.originalname or item.name
            # сохраняем в _test_failed_incremental оригинальное имя упавшего теста
            _test_failed_incremental.setdefault(cls_name, {}).setdefault(
                parametrize_index, test_name
            )


def pytest_runtest_setup(item):
    if "incremental" in item.keywords:
        # извлекаем из теста имя класса
        cls_name = str(item.cls)
        # проверяем, падал ли предыдущий тест на этом классе
        if cls_name in _test_failed_incremental:
            # извлекаем индексы теста (если вместе с  incremental используется параметризация)
            parametrize_index = (
                tuple(item.callspec.indices.values())
                if hasattr(item, "callspec")
                else ()
            )
            # извлекаем имя первой тестовой функции, которая должна упасть для этого имени класса␣
↪и индекса
            test_name = _test_failed_incremental[cls_name].get(parametrize_index, None)
```

(continues on next page)

(продолжение с предыдущей страницы)

```
            # если нашли такое имя, значит, тест падал для такой комбинации класса & фукнкции
            if test_name is not None:
                pytest.xfail("previous test failed ({})".format(test_name))
```

Эти два хука совместно работают на прерывание маркированных `incremental` тестов в классе. Вот пример тестового модуля:

```
# content of test_step.py

import pytest


@pytest.mark.incremental
class TestUserHandling:
    def test_login(self):
        pass

    def test_modification(self):
        assert 0

    def test_deletion(self):
        pass


def test_normal():
    pass
```

Запустим:

```
$ pytest -rx
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items

test_step.py .Fx.                                                      [100%]

================================= FAILURES =================================
____________________ TestUserHandling.test_modification ____________________

self = <test_step.TestUserHandling object at 0xdeadbeef>

    def test_modification(self):
>       assert 0
E       assert 0

test_step.py:11: AssertionError
========================== short test summary info ==========================
XFAIL test_step.py::TestUserHandling::test_deletion
  reason: previous test failed (test_modification)
================== 1 failed, 2 passed, 1 xfailed in 0.12s ==================
```

Посокольку `test_modification` упал, `test_deletion` не выполнялся и попал в отчет как «ожидаемо падающий» (xfailed).

12.2.9 Фикстуры уровня пакета/каталога (setups)

Если в вашем дереве тестов есть вложенные каталоги, можно каждый из них рассматривать как область действия фикстур - для этого достаточно разместить фикстуры в файле `conftest.py` соответствующего каталога. При этом можно использовать все типы фикстур, включая фикстуры *autouse* - аналоги `setup/teardown` функций `xUnit` Однако имейте в виду, что рекомендуется явно ссылаться на фикстуры в тестах и классах, вместо того, чтобы полагаться на неявное выполнение `setup/teardown` функций, особенно если они расположены далеко от использующих их тестов.

Вот пример, как сделать фикстуру `db` доступной в каталоге:

```
# content of a/conftest.py
import pytest


class DB:
    pass


@pytest.fixture(scope="session")
def db():
    return DB()
```

И тестовый модуль в этой же директории:

```
# content of a/test_db.py
def test_a1(db):
    assert 0, db  # to show value
```

Еще один тестовый модуль:

```
# content of a/test_db2.py
def test_a2(db):
    assert 0, db  # to show value
```

А этот модуль расположен в соседней (сестринской) директории, и там фикстура `db` будет не видна:

```
# content of b/test_error.py
def test_root(db):  # no db here, will error out
    pass
```

Теперь запустим:

```
$ pytest
========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 7 items

test_step.py .Fx.                                                    [ 57%]
a/test_db.py F                                                       [ 71%]
a/test_db2.py F                                                      [ 85%]
b/test_error.py E                                                    [100%]

================================== ERRORS ==================================
_______________________ ERROR at setup of test_root ________________________
```

(continues on next page)

(продолжение с предыдущей страницы)

```
file $REGENDOC_TMPDIR/b/test_error.py, line 1
  def test_root(db):  # no db here, will error out
E       fixture 'db' not found
>       available fixtures: cache, capfd, capfdbinary, caplog, capsys, capsysbinary, doctest_
↪namespace, monkeypatch, pytestconfig, record_property, record_testsuite_property, record_xml_
↪attribute, recwarn, tmp_path, tmp_path_factory, tmpdir, tmpdir_factory
>       use 'pytest --fixtures [testpath]' for help on them.

$REGENDOC_TMPDIR/b/test_error.py:1
================================= FAILURES =================================
____________________ TestUserHandling.test_modification ____________________

self = <test_step.TestUserHandling object at 0xdeadbeef>

    def test_modification(self):
>       assert 0
E       assert 0

test_step.py:11: AssertionError
_________________________________ test_a1 __________________________________

db = <conftest.DB object at 0xdeadbeef>

    def test_a1(db):
>       assert 0, db  # to show value
E       AssertionError: <conftest.DB object at 0xdeadbeef>
E       assert 0

a/test_db.py:2: AssertionError
_________________________________ test_a2 __________________________________

db = <conftest.DB object at 0xdeadbeef>

    def test_a2(db):
>       assert 0, db  # to show value
E       AssertionError: <conftest.DB object at 0xdeadbeef>
E       assert 0

a/test_db2.py:2: AssertionError
============= 3 failed, 2 passed, 1 xfailed, 1 error in 0.12s ==============
```

Оба тестовых модуля из каталога `a` видят одну и ту же фикстуру `db`, а вот модуль из каталога `b` ее не видит. Конечно, мы можем так же определить фикстуру `db` в файле `b/conftest.py`. Обратите внимание, что каждая фикстура создается, только если требуется в тесте (кроме `autouse` фикстур - они всегда выполняются перед запуском тестов).

12.2.10 Обработка отчетов

Если нужно обрабатывать отчеты `pytest` или получать доступ к исполняющему тесты окружению, можно реализовать хук, который будет вызываться во время создания объекта «report». Ниже мы обрабатываем все упавшие тесты и получаем доступ к фикстуре (если она используется в тестах), которую хотим посмотреть во время обработки. Всю информацию мы запишем в файл `failures`:

```
# content of conftest.py
```

(continues on next page)

(продолжение с предыдущей страницы)

```
import pytest
import os.path


@pytest.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    # выполняем все остальные хуки, чтобы получить report object
    outcome = yield
    rep = outcome.get_result()

    # мы ищем только вызовы упавших тестов, а не setup/teardown
    if rep.when == "call" and rep.failed:
        mode = "a" if os.path.exists("failures") else "w"
        with open("failures", mode) as f:
            # давайте ради прикола посмотрим на фикстуру
            if "tmpdir" in item.fixturenames:
                extra = " ({})".format(item.funcargs["tmpdir"])
            else:
                extra = ""

            f.write(rep.nodeid + extra + "\n")
```

Допустим, у нас есть тесты:

```
# content of test_module.py
def test_fail1(tmpdir):
    assert 0


def test_fail2():
    assert 0
```

Запустим их:

```
$ pytest test_module.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py FF                                                  [100%]

================================ FAILURES ================================
_____________________________ test_fail1 _____________________________

tmpdir = local('PYTEST_TMPDIR/test_fail10')

    def test_fail1(tmpdir):
>       assert 0
E       assert 0

test_module.py:2: AssertionError
_____________________________ test_fail2 _____________________________

    def test_fail2():
```

(continues on next page)

(продолжение с предыдущей страницы)

```
>       assert 0
E       assert 0

test_module.py:6: AssertionError
============================ 2 failed in 0.12s =============================
```

Мы получили файл `failures` с идентификаторами упавших тестов:

```
$ cat failures
test_module.py::test_fail1 (PYTEST_TMPDIR/test_fail10)
test_module.py::test_fail2
```

12.2.11 Как сделать информацию о результатах тестов доступной для фикстуры

Если вы хотите, чтобы отчеты о результатах тестов были доступны в финализирующей фикстуре, можно реализовать следующий небольшой плагин:

```
# content of conftest.py

import pytest


@pytest.hookimpl(tryfirst=True, hookwrapper=True)
def pytest_runtest_makereport(item, call):
    # выполняем все остальные хуки до получения report object
    outcome = yield
    rep = outcome.get_result()

    # устанавливаем атрубут отчета на каждом этапе вызова:
    # "setup", "call", "teardown"

    setattr(item, "rep_" + rep.when, rep)


@pytest.fixture
def something(request):
    yield
    # "request.node" в данном случае "item", поскольку по мы используем уровень
    # по умолчанию - "function" scope
    if request.node.rep_setup.failed:
        print("setting up a test failed!", request.node.nodeid)
    elif request.node.rep_setup.passed:
        if request.node.rep_call.failed:
            print("executing test failed", request.node.nodeid)
```

Теперь, пусть у нас есть неудачные тесты:

```
# content of test_module.py

import pytest


@pytest.fixture
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def other():
    assert 0


def test_setup_fails(something, other):
    pass


def test_call_fails(something):
    assert 0


def test_fail2():
    assert 0
```

Запустим их:

```
$ pytest -s test_module.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 3 items

test_module.py Esetting up a test failed! test_module.py::test_setup_fails
Fexecuting test failed test_module.py::test_call_fails
F

================================== ERRORS ==================================
____________________ ERROR at setup of test_setup_fails ____________________

    @pytest.fixture
    def other():
>       assert 0
E       assert 0

test_module.py:7: AssertionError
================================= FAILURES =================================
_____________________________ test_call_fails ______________________________

something = None

    def test_call_fails(something):
>       assert 0
E       assert 0

test_module.py:15: AssertionError
_______________________________ test_fail2 ________________________________

    def test_fail2():
>       assert 0
E       assert 0

test_module.py:19: AssertionError
======================== 2 failed, 1 error in 0.12s ========================
```

Как видите, финализаторы могут использовать информацию из отчета.

12.2.12 Переменная окружения PYTEST_CURRENT_TEST

Иногда тестовая сессия может зависнуть, и бывает непросто выяснить, на каком именно тесте она «застряла» (например, `pytest` запущен в «тихом» (`-q`) режиме или нет доступа к консольному выводу). Это особенно неприятно, когда проблема возникает нерегулярно - получаем так называемые «мерцающие» (*flaky*) тесты.

При запуске тестов `pytest` задает переменную окружения `PYTEST_CURRENT_TEST`, которую можно проверять с помощью утилит мониторинга процессов или библиотек вроде psutil для того, чтобы выяснить, какой именно тест «застрял»:

```
import psutil

for pid in psutil.pids():
    environ = psutil.Process(pid).environ()
    if "PYTEST_CURRENT_TEST" in environ:
        print(f'pytest process {pid} running: {environ["PYTEST_CURRENT_TEST"]}')
```

Во время тестовой сессии `pytest` будет присваивать `PYTEST_CURRENT_TEST` текущий идентификатор узла (*nodeid*) и текущее состояние: `setup`, `call` или `teardown`.

К примеру, если мы запустим одну тестовую функцию `test_foo` из модуля `foo_module.py`, `PYTEST_CURRENT_TEST` будет принимать следующие значения:

1. `foo_module.py::test_foo (setup)`
2. `foo_module.py::test_foo (call)`
3. `foo_module.py::test_foo (teardown)`

Именно в таком порядке.

Примечание: Поскольку содержимое `PYTEST_CURRENT_TEST` должно быть читабельно, текущий формат от релиза к релизу может меняться (даже при фиксации багов), поэтому не стоит полагаться именно на такой вид при написании сценариев и автоматизации.

12.2.13 «Заморозка» pytest

Если вы «замораживаете» приложение с помощью инструмента вроде PyInstaller, чтобы распространить его среди конечных пользователей, хорошей идеей будет упаковать и ваш `pytest` и запускать тесты с «замороженным» приложением. Благодаря такому способу некоторые ошибки (например, отсутствие в исполняемом файле нужных зависимостей) могут быть обнаружены на раннем этапе; кроме того, это позволяет вам отправлять тестовые файлы пользователям, чтобы они сами могли запустить тесты на своих машинах, что может быть полезно для получения дополнительной информации о трудновоспроизводимой ошибке.

К счастью, в последних релизах `PyInstaller` уже есть хук для `pytest`, но если вы используете для «заморозки» другие инструменты, такие как `cx_freeze` или `py2exe`, можно использовать `pytest.freeze_includes()` для получения полного списка используемых `pytest` модулей. Однако конфигурирование инструмента для поиска внутренних модулей зависит от используемого инструмента.

Вместо того, чтобы «заморозить» `pytest` как отдельный исполняемый файл, можно заставить «замороженную» программу воспринимать `pytest` как некий хитрый аргумент, к которому она обращается во время запуска. Это позволит вам иметь один исполняемый файл - обычно так удобнее. Обратите внимание, что механизм поиска плагинов, используемый `pytest`, не работает с «замороженными» исполняемыми файлами, поэтому `pytest` не сможет найти сторонний плагин автоматически. Чтобы

подключить стороние плагины вроде pytest-timeout, их нужно явно импортировать и передать в pytest.main.

```
# contents of app_main.py
import sys
import pytest_timeout  # сторонний плагин

if len(sys.argv) > 1 and sys.argv[1] == "--pytest":
    import pytest

    sys.exit(pytest.main(sys.argv[2:], plugins=[pytest_timeout]))
else:
    # нормальное выполнение приложения: здесь можно проанализировать argv
    # как обычно
    ...
```

Такой шаблон позволит вам запускать тесты на «замороженном» приложении со стандартными опциями командной строки pytest:

```
./app_main --pytest --verbose --tb=long --junitxml=results.xml test-suite/
```

12.3 Параметризация тестов

pytest позволяет леко параметризовать тестовые функции. Основы параметризации см. *Параметризация фикстур и тестовых функций*.

Дальше приводятся некоторые примеры использования встроенного механизма.

12.3.1 Генарация комбинаций параметров, зависящих от опции командной строки

Давайте представим, что мы хотим проводить тесты с различными вычисляемыми параметрами и диапазон параметров должен определяться параметром командной строки. Для начала напишем ничего не делающий вычислительный тест:

```
# content of test_compute.py


def test_compute(param1):
    assert param1 < 4
```

Дальше добавим конфигурирование:

```
# content of conftest.py


def pytest_addoption(parser):
    parser.addoption("--all", action="store_true", help="run all combinations")


def pytest_generate_tests(metafunc):
    if "param1" in metafunc.fixturenames:
        if metafunc.config.getoption("all"):
            end = 5
```

(continues on next page)

(продолжение с предыдущей страницы)

```
        else:
            end = 2
        metafunc.parametrize("param1", range(end))
```

Это означает, что если мы не используем `--all`, будет запускаться 2 теста:

```
$ pytest -q test_compute.py
..                                                                   [100%]
2 passed in 0.12s
```

Мы произвели всего пару вычислений, поэтому видим 2 точки. Давайте воспользуемся нашей опцией:

```
$ pytest -q --all
....F                                                                [100%]
================================= FAILURES =================================
_____________________________ test_compute[4] ______________________________

param1 = 4

    def test_compute(param1):
>       assert param1 < 4
E       assert 4 < 4

test_compute.py:4: AssertionError
1 failed, 4 passed in 0.12s
```

Как и ожидалось, при выполнении тестов по всему диапазону значений `param1`, мы получили ошибку на последнем тесте.

12.3.2 Различные способы определения ID тестов

`pytest` конструирует строку, которая является идентификатором (ID) теста для каждого множества значений параметризованного теста. Эти индентификаторы можно использовать с опцией `-k`, чтобы отборать для выполнения определенные тесты, и они же идентифицируют конкретный упавший тест. Запустив `pytest --collect-only` , можно посмотреть сгенерированные ID.

У чисел, строк, логических значений и значения None есть свои строковые представления, которые используются в ID тестов. Для остальных объектов `pytest` генерирует ID на основании имен аргументов:

```
# content of test_time.py

import pytest

from datetime import datetime, timedelta

testdata = [
    (datetime(2001, 12, 12), datetime(2001, 12, 11), timedelta(1)),
    (datetime(2001, 12, 11), datetime(2001, 12, 12), timedelta(-1)),
]


@pytest.mark.parametrize("a,b,expected", testdata)
def test_timedistance_v0(a, b, expected):
    diff = a - b
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    assert diff == expected


@pytest.mark.parametrize("a,b,expected", testdata, ids=["forward", "backward"])
def test_timedistance_v1(a, b, expected):
    diff = a - b
    assert diff == expected


def idfn(val):
    if isinstance(val, (datetime,)):
        # note this wouldn't show any hours/minutes/seconds
        return val.strftime("%Y%m%d")


@pytest.mark.parametrize("a,b,expected", testdata, ids=idfn)
def test_timedistance_v2(a, b, expected):
    diff = a - b
    assert diff == expected


@pytest.mark.parametrize(
    "a,b,expected",
    [
        pytest.param(
            datetime(2001, 12, 12), datetime(2001, 12, 11), timedelta(1), id="forward"
        ),
        pytest.param(
            datetime(2001, 12, 11), datetime(2001, 12, 12), timedelta(-1), id="backward"
        ),
    ],
)
def test_timedistance_v3(a, b, expected):
    diff = a - b
    assert diff == expected
```

В `test_timedistance_v0` мы позволяем `pytest` самому генерировать ID.

В `test_timedistance_v1` мы определяем идентификаторы, используя список строк, которые будут использоваться в качестве ID. Они весьма лаконичны, но могут быть сложны для поддержания.

В `test_timedistance_v2` мы определяем `ids` с помощью функции, которая генерирует строковое представление, которое будет частью идентификатора теста. Здесь обозначения наших `datetime`-аргументов генерируются функцией `idfn`, но из-за того, что мы не можем формировать представление для объектов `timedelta`, для них все еще используется стандартное представление `pytest`:

```
$ pytest test_time.py --collect-only
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 8 items
<Module test_time.py>
  <Function test_timedistance_v0[a0-b0-expected0]>
  <Function test_timedistance_v0[a1-b1-expected1]>
  <Function test_timedistance_v1[forward]>
  <Function test_timedistance_v1[backward]>
```

(continues on next page)

(продолжение с предыдущей страницы)

```
  <Function test_timedistance_v2[20011212-20011211-expected0]>
  <Function test_timedistance_v2[20011211-20011212-expected1]>
  <Function test_timedistance_v3[forward]>
  <Function test_timedistance_v3[backward]>

========================== no tests ran in 0.12s ===========================
```

В `test_timedistance_v3` мы используем `pytest.param` для указания ID вместе с конкретными данными, вместо того, чтобы перечислять их по отдельности.

12.3.3 Быстрый запуск «testscenarios»

Вот быстрый способ запуска тестов, сконфигурированных с помощью test scenarios (дополнения от Роберта Коллинза для стандартного фреймворка `unittest`). Тут придется немного потрудиться с созданием корректных аргументов для Metafunc.parametrize:

```
# content of test_scenarios.py


def pytest_generate_tests(metafunc):
    idlist = []
    argvalues = []
    for scenario in metafunc.cls.scenarios:
        idlist.append(scenario[0])
        items = scenario[1].items()
        argnames = [x[0] for x in items]
        argvalues.append([x[1] for x in items])
    metafunc.parametrize(argnames, argvalues, ids=idlist, scope="class")


scenario1 = ("basic", {"attribute": "value"})
scenario2 = ("advanced", {"attribute": "value2"})


class TestSampleWithScenarios:
    scenarios = [scenario1, scenario2]

    def test_demo1(self, attribute):
        assert isinstance(attribute, str)

    def test_demo2(self, attribute):
        assert isinstance(attribute, str)
```

Это полностью самодостаточный пример, который можно запустить:

```
$ pytest test_scenarios.py
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items

test_scenarios.py ....                                                   [100%]

============================= 4 passed in 0.12s =============================
```

Если вы просто соберете тесты, то увидите „advanced“ и „basic“ в качестве переменных тестовой функции:

```
$ pytest --collect-only test_scenarios.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items
<Module test_scenarios.py>
  <Class TestSampleWithScenarios>
      <Function test_demo1[basic]>
      <Function test_demo2[basic]>
      <Function test_demo1[advanced]>
      <Function test_demo2[advanced]>

========================== no tests ran in 0.12s ==========================
```

12.3.4 Отсрочка настройки параметризованных ресурсов

Параметризация тестовой функции происходит во время сборки тестов. Хорошей идеей является настройка длительных по времени процессов, вроде подключения к базе данных, только при запуске такого теста. Вот простой пример, как можно этого добиться. Этот тест использует объект фикстуры db:

```
# content of test_backends.py

import pytest


def test_db_initialized(db):
    # a dummy test
    if db.__class__.__name__ == "DB2":
        pytest.fail("deliberately failing for demo purposes")
```

Теперь мы добавим конфигурацию тестов, которая генерирует два вызова функции `test_db_initialized` и реализует фабрику, которая создает объект базы данных для конкретного запуска теста:

```
# content of conftest.py
import pytest


def pytest_generate_tests(metafunc):
    if "db" in metafunc.fixturenames:
        metafunc.parametrize("db", ["d1", "d2"], indirect=True)


class DB1:
    "one database object"


class DB2:
    "alternative database object"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@pytest.fixture
def db(request):
    if request.param == "d1":
        return DB1()
    elif request.param == "d2":
        return DB2()
    else:
        raise ValueError("invalid internal test config")
```

Давайте сначала глянем, как все это выглядит во время сборки:

```
$ pytest test_backends.py --collect-only
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items
<Module test_backends.py>
  <Function test_db_initialized[d1]>
  <Function test_db_initialized[d2]>

========================== no tests ran in 0.12s ==========================
```

Потом запустим тесты:

```
$ pytest -q test_backends.py
.F                                                                   [100%]
================================= FAILURES =================================
_________________________ test_db_initialized[d2] __________________________

db = <conftest.DB2 object at 0xdeadbeef>

    def test_db_initialized(db):
        # a dummy test
        if db.__class__.__name__ == "DB2":
>           pytest.fail("deliberately failing for demo purposes")
E           Failed: deliberately failing for demo purposes

test_backends.py:8: Failed
1 failed, 1 passed in 0.12s
```

Первый вызов с `db == "DB1"` прошел, в то время как второй с `db == "DB2"` - упал. Наша фикстура `db` создавала каждую из баз данных на этапе настройки, в то время как `pytest_generate_tests` генерировала два соответствующих вызова `test_db_initialized` на этапе сборки.

12.3.5 Применение `indirect` к отдельному аргументу

Очень часто при параметризации используется более одного аргумента. К конкретному аргументу можно применять параметр `indirect`. Это можно сделать, передав список или кортеж имен аргументов параметру `indirect`. В нижеприведенном примере есть функция `test_indirect`, которая использует 2 фикстуры: `x` и `y`. Здесь мы передаем `indirect` список, который содержит имя фикстуры `x`, соответственно, он будет применен только к этому аргумента, и значение `a` будет передано соответствующей фикстуре:

```
# content of test_indirect_list.py

import pytest


@pytest.fixture(scope="function")
def x(request):
    return request.param * 3


@pytest.fixture(scope="function")
def y(request):
    return request.param * 2


@pytest.mark.parametrize("x, y", [("a", "b")], indirect=["x"])
def test_indirect(x, y):
    assert x == "aaa"
    assert y == "b"
```

Тест пройдет успешно:

```
$ pytest test_indirect_list.py --collect-only
========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item
<Module test_indirect_list.py>
  <Function test_indirect[a-b]>

========================== no tests ran in 0.12s ==========================
```

Обратите внимание, что каждый аргумент в списке `parametrize` должен быть явно объявлен в соответствующей тестовой функции или с помощью `indirect`.

12.3.6 Параметризация тестовых методов через конфигурацию класса

Ниже - пример тестовой функции `pytest_generate_tests`, реализующей схему параметризации, похожую на unittest parametrizer, но с более коротким кодом:

```
# content of ./test_parametrize.py
import pytest


def pytest_generate_tests(metafunc):
    # вызывается один раз для каждой тестовой функции
    funcarglist = metafunc.cls.params[metafunc.function.__name__]
    argnames = sorted(funcarglist[0])
    metafunc.parametrize(
        argnames, [[funcargs[name] for name in argnames] for funcargs in funcarglist]
    )


class TestClass:
    # карта, определяющая множество аргументов тестовой функции
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    params = {
        "test_equals": [dict(a=1, b=2), dict(a=3, b=3)],
        "test_zerodivision": [dict(a=1, b=0)],
    }

    def test_equals(self, a, b):
        assert a == b

    def test_zerodivision(self, a, b):
        with pytest.raises(ZeroDivisionError):
            a / b
```

Наш генератор тестов отслеживает определение множества аргументов для каждой тестовой функции на уровне класса. Давайте выполним:

```
$ pytest -q
F..                                                                  [100%]
================================= FAILURES =================================
_____________________ TestClass.test_equals[1-2] _____________________

self = <test_parametrize.TestClass object at 0xdeadbeef>, a = 1, b = 2

    def test_equals(self, a, b):
>       assert a == b
E       assert 1 == 2

test_parametrize.py:21: AssertionError
1 failed, 2 passed in 0.12s
```

12.3.7 Непрямая параметризация несколькими фикстурами

Вот реальный (урезанный) пример использования параметризации для тестированмя кроссплатформенной сериализации объектов различными интерпретаторами `Python`. Мы определяем функцию `test_basic_objects`, которая должна запускаться с различными множествами трех своих аргументов:

- `python1`: первый интерпретатор `python`, запускается для сериализации объекта
- `python2`: второй интерпретатор `python`, запускается для десериализации объекта
- `obj`: объект, который нужно записывать/считывать

```
"""
модуль, содержащий параметризованные тесты, тестирующие кросс-python
сериализацию через модуль pickle.
"""
import shutil
import subprocess
import textwrap

import pytest

pythonlist = ["python3.5", "python3.6", "python3.7"]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@pytest.fixture(params=pythonlist)
def python1(request, tmpdir):
    picklefile = tmpdir.join("data.pickle")
    return Python(request.param, picklefile)


@pytest.fixture(params=pythonlist)
def python2(request, python1):
    return Python(request.param, python1.picklefile)


class Python:
    def __init__(self, version, picklefile):
        self.pythonpath = shutil.which(version)
        if not self.pythonpath:
            pytest.skip("{!r} not found".format(version))
        self.picklefile = picklefile

    def dumps(self, obj):
        dumpfile = self.picklefile.dirpath("dump.py")
        dumpfile.write(
            textwrap.dedent(
                r"""
                import pickle
                f = open({!r}, 'wb')
                s = pickle.dump({!r}, f, protocol=2)
                f.close()
                """.format(
                    str(self.picklefile), obj
                )
            )
        )
        subprocess.check_call((self.pythonpath, str(dumpfile)))

    def load_and_is_true(self, expression):
        loadfile = self.picklefile.dirpath("load.py")
        loadfile.write(
            textwrap.dedent(
                r"""
                import pickle
                f = open({!r}, 'rb')
                obj = pickle.load(f)
                f.close()
                res = eval({!r})
                if not res:
                    raise SystemExit(1)
                """.format(
                    str(self.picklefile), expression
                )
            )
        )
        print(loadfile)
        subprocess.check_call((self.pythonpath, str(loadfile)))


@pytest.mark.parametrize("obj", [42, {}, {1: 3}])
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def test_basic_objects(python1, python2, obj):
    python1.dumps(obj)
    python2.load_and_is_true("obj == {}".format(obj))
```

Если не все требуемые интерпретаторы установлены, то некоторые множества аргументов будут пропущены; в противном случае запускаются все комбинации аргументов (3 первых интерпретатора x 3 вторых интерпретатора x 3 объекта для сериализации/десериализации):

```
. $ pytest -rs -q multipython.py
sssssssssss...sssssssssss                                            [100%]
======================== short test summary info ========================
SKIPPED [12] $REGENDOC_TMPDIR/CWD/multipython.py:29: 'python3.5' not found
SKIPPED [12] $REGENDOC_TMPDIR/CWD/multipython.py:29: 'python3.7' not found
3 passed, 24 skipped in 0.12s
```

12.3.8 Непрямая параметризация реализованных опций/импорта

Если вы хотите сравнить несколько вариантов реализации одного и того же API, то можете написать тестовые функции, которые получают уже импортированные опции и пропускаются в случае, если опция недоступна или ее не удалось импортировать. Пусть у нас есть «базовая» реализация, а остальные (возможно, оптимизированные) должны обеспечивать такие же результаты:

```
# content of conftest.py

import pytest


@pytest.fixture(scope="session")
def basemod(request):
    return pytest.importorskip("base")


@pytest.fixture(scope="session", params=["opt1", "opt2"])
def optmod(request):
    return pytest.importorskip(request.param)
```

Базовое воплощение нашей функции выглядит так:

```
# content of base.py
def func1():
    return 1
```

А ее оптимизированная версия так:

```
# content of opt1.py
def func1():
    return 1.0001
```

И наконец, небольшой тестовый модуль:

```
# content of test_module.py


def test_func1(basemod, optmod):
    assert round(basemod.func1(), 3) == round(optmod.func1(), 3)
```

Если вы запустите его с опцией `-rs`:

```
$ pytest -rs test_module.py
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 2 items

test_module.py .s                                                    [100%]

======================== short test summary info ========================
SKIPPED [1] $REGENDOC_TMPDIR/conftest.py:12: could not import 'opt2': No module named 'opt2'
====================== 1 passed, 1 skipped in 0.12s ======================
```

Как видите, у нас нет модуля `opt2`, и поэтому второй тест `test_func1` был пропущен. Несколько замечаний:

- фикстуры в `conftest.py` имеют уровень сессии, ибо нам не нужно импортировать модуль больше одного раза;
- если у вас есть несколько тестов и пропущенный импорт, счетчик пропущенных тестов (`SKIPPED [1]`) покажет большее число;
- для параметризации тестовых функций можно также использовать *@pytest.mark.parametrize*.

12.3.9 Установка маркера или ID для конкретного параметризованного теста

Чтобы применить маркер или установить ID конкретному тесту, используйте `pytest.param`. Например:

```
# content of test_pytest_param_example.py
import pytest


@pytest.mark.parametrize(
    "test_input,expected",
    [
        ("3+5", 8),
        pytest.param("1+7", 8, marks=pytest.mark.basic),
        pytest.param("2+4", 6, marks=pytest.mark.basic, id="basic_2+4"),
        pytest.param(
            "6*9", 42, marks=[pytest.mark.basic, pytest.mark.xfail], id="basic_6*9"
        ),
    ],
)
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

В примере у нас есть 4 параметризованных теста. Исключая первый тест, мы маркируем все остальные собственным маркером `basic`, а для четвертого используем еще и встроенй маркер `xfail`, чтобы отметить, что этот тест должен упасть. Некоторым тестам мы еще и устанавливаем ID для ясности.

Теперь давайте запустим маркированные `basic` тесты в режиме подробной трассировки:

```
$ pytest -v -m basic
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
```

(continues on next page)

(продолжение с предыдущей страницы)

```
rootdir: $REGENDOC_TMPDIR
collecting ... collected 17 items / 14 deselected / 3 selected

test_pytest_param_example.py::test_eval[1+7-8] PASSED                [ 33%]
test_pytest_param_example.py::test_eval[basic_2+4] PASSED            [ 66%]
test_pytest_param_example.py::test_eval[basic_6*9] XFAIL              [100%]

=============== 2 passed, 14 deselected, 1 xfailed in 0.12s ================
```

В результате:

- были собраны все четыре теста;
- один тест был отброшен, поскольку не имел пометки `basic`;
- все три теста с маркером `basic` были выбраны;
- тест `test_eval[1+7-8]` успешно прошел, но его ID был сгенерирован автоматически и мало что объясняет;
- тест `test_eval[basic_2+4]` прошел;
- тест `test_eval[basic_6*9]` должен был упасть и упал.

12.3.10 Параметризация с генерацией исключений

Чтобы написать параметризованные тесты, часть из которых могла бы генерировать исключения, используйте pytest.raises с декоратором pytest.mark.parametrize.

В дополнение к `raises` будет полезно определить простейший контекстный менеджер `does_not_raise`, например, так:

```
from contextlib import contextmanager
import pytest


@contextmanager
def does_not_raise():
    yield


@pytest.mark.parametrize(
    "example_input,expectation",
    [
        (3, does_not_raise()),
        (2, does_not_raise()),
        (1, does_not_raise()),
        (0, pytest.raises(ZeroDivisionError)),
    ],
)
def test_division(example_input, expectation):
    """Test how much I know division."""
    with expectation:
        assert (6 / example_input) is not None
```

В вышеприведенном примере первые три тестовых случая должны проходить в обычном режиме, а вот четвертый - генерировать `ZeroDivisionError`.

Если планируется поддержка только Python 3.7 и выше, для определения does_not_raise можно просто использовать nullcontext:

```
from contextlib import nullcontext as does_not_raise
```

Или, если поддерживается Python 3.3 и выше:

```
from contextlib import ExitStack as does_not_raise
```

Или, если хочется, можно запустить pip install contextlib2 и использовать:

```
from contextlib2 import nullcontext as does_not_raise
```

12.4 Работа с пользовательской маркировкой

Вот несколько примеров использования механизма *маркировки*.

12.4.1 Маркировка тестовых функций и отбор маркированных тестов для запуска

Тестовую функцию можно настроить с помощью маркировки:

```
# content of test_server.py

import pytest


@pytest.mark.webtest
def test_send_http():
    pass  # perform some webtest test for your app


def test_something_quick():
    pass


def test_another():
    pass


class TestClass:
    def test_method(self):
        pass
```

Теперь вы можете запускать тесты только с меткой webtest:

```
$ pytest -v -m webtest
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 3 deselected / 1 selected
```

(continues on next page)

(продолжение с предыдущей страницы)

```
test_server.py::test_send_http PASSED                                    [100%]

==================== 1 passed, 3 deselected in 0.12s =====================
```

Можно и наоборот - запустить все тесты, кроме помеченных `webtest`:

```
$ pytest -v -m "not webtest"
========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 1 deselected / 3 selected

test_server.py::test_something_quick PASSED                              [ 33%]
test_server.py::test_another PASSED                                      [ 66%]
test_server.py::TestClass::test_method PASSED                            [100%]

==================== 3 passed, 1 deselected in 0.12s =====================
```

12.4.2 Отбор тестов по идентификатору узла (node ID)

В качестве позиционных аргументов для отбора тестов можно передать `pytest` один или несколько идентификаторов узлов (см. ниже). Это облегчает отбор тестов по именам модулей, классов, методов или функций:

```
$ pytest -v test_server.py::TestClass::test_method
========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED                            [100%]

============================ 1 passed in 0.12s ===========================
```

Можно выбрать и сам класс:

```
$ pytest -v test_server.py::TestClass
========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 1 item

test_server.py::TestClass::test_method PASSED                            [100%]

============================ 1 passed in 0.12s ===========================
```

Или отобрать сразу несколько узлов:

```
$ pytest -v test_server.py::TestClass test_server.py::test_send_http
========================== test session starts ===========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
```

(continues on next page)

(продолжение с предыдущей страницы)

```
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 2 items

test_server.py::TestClass::test_method PASSED                        [ 50%]
test_server.py::test_send_http PASSED                                [100%]

============================ 2 passed in 0.12s =============================
```

Примечание: Идентификаторы узлов имеют формат `module.py::class::method` или `module.py::function`. Тесты собираются по идентификатрам узлов, так что при передаче `module.py::class` будут выбраны все тестовые методы класса. Для каждого параметра параметризованной фикстуры или функции также создаются узлы, так что идентификатор для отбора конкретного параметризованного теста должен включать значение параметра, например, `module.py::function[param]`.

Идентификаторы узлов упавшего теста отображаются в сводном отчете, если `pytest` запущен с опцией `-rf`. Идентификаторы узлов можно определять на основании списка собранных тестов, выводимого `pytest --collectonly`.

12.4.3 Использование опции `-k "выражение"` для отбора тестов по именам

Опцию `-k` командной строки можно использовать, чтобы указать подстроку, которая должна присутствовать в именах тестов (при использовании опции `-m` проверяется точное совпадение). Это облегчает отбор тестов по именам.

При этом сопоставление строк производится без учета регистра. Запустим с модулем из примера выше:

```
$ pytest -v -k http  # running with the above defined example module
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 3 deselected / 1 selected

test_server.py::test_send_http PASSED                                [100%]

====================== 1 passed, 3 deselected in 0.12s =======================
```

Можно также запустить все тесты, которые не содержат ключевого слова:

```
$ pytest -k "not send_http" -v
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 1 deselected / 3 selected

test_server.py::test_something_quick PASSED                          [ 33%]
test_server.py::test_another PASSED                                  [ 66%]
test_server.py::TestClass::test_method PASSED                        [100%]

====================== 3 passed, 1 deselected in 0.12s =======================
```

Или отборать все тесты, в именах которых есть подстрока «http» или «quick»:

```
$ pytest -k "http or quick" -v
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collecting ... collected 4 items / 2 deselected / 2 selected

test_server.py::test_send_http PASSED                                  [ 50%]
test_server.py::test_something_quick PASSED                            [100%]

===================== 2 passed, 2 deselected in 0.12s ======================
```

Примечание: Если вы используете выражение вида `"X and Y"`, то и `X`, и `Y` должны быть простыми именами без ключевых слов. К примеру, запуск с переданными `"pass"` или `"from"` приведет к синтаксической ошибке, поскольку `"-k"` вычисляет выражения с помощью `Python`-функции eval .

Однако, если аргумент `-k` является просто строкой, то таких ограничений нет, так же как и в случае с `"-k 'не строка'"`. Можно указывать и числа, например, `"-k 1.3"`, если ваши тесты параметризованы действительным числом `"1.3"`.

12.4.4 Регистрация маркеров

Зарегистрировать свои маркеры несложно:

```
# content of pytest.ini
[pytest]
markers =
    webtest: mark a test as a webtest.
```

Можно запросить список маркеров для тестового набора и увидеть, что в списке появился только что зарегистрированный маркер `webtest`:

```
$ pytest --markers
@pytest.mark.webtest: mark a test as a webtest.

@pytest.mark.filterwarnings(warning): add a warning filter to the given test. see https://docs.
↪pytest.org/en/latest/warnings.html#pytest-mark-filterwarnings

@pytest.mark.skip(reason=None): skip the given test function with an optional reason. Example:␣
↪skip(reason="no way of currently testing this") skips the test.

@pytest.mark.skipif(condition): skip the given test function if eval(condition) results in a True␣
↪value.  Evaluation happens within the module global context. Example: skipif('sys.platform ==
↪"win32"') skips the test if we are on the win32 platform. see https://docs.pytest.org/en/latest/
↪skipping.html

@pytest.mark.xfail(condition, reason=None, run=True, raises=None, strict=False): mark the test␣
↪function as an expected failure if eval(condition) has a True value. Optionally specify a reason␣
↪for better reporting and run=False if you don't even want to execute the test function. If only␣
↪specific exception(s) are expected, you can list them in raises, and if the test fails in other␣
↪ways, it will be reported as a true failure. See https://docs.pytest.org/en/latest/skipping.html

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times passing in␣
↪different arguments in turn. argvalues generally needs to be a list of values if argnames␣
↪specifies only one name or a list of tuples of values if argnames specifies multiple names.␣
↪Example: @parametrize('arg1', [1,2]) would lead to two calls of the decorated test function, one␣
↪with arg1=1 and another with arg1=2.see https://docs.pytest.org/en/latest/parametrize.html for␣
↪more info and examples.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all of the␣
↪specified fixtures. see https://docs.pytest.org/en/latest/fixture.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin machinery will try␣
↪to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin machinery will try␣
↪to call it last/as late as possible.
```

Пример добавления маркеров из плагина и работы с ними см. ниже *Настраиваемые маркеры и опции командной строки для контроля запуска тестов*.

Примечание: Мы рекомендуем явно регистрировать маркеры так, чтобы:

- Ваши маркеры определялись только в одном месте тестового набора
- Получение списка маркеров с помощью `pytest --markers` давало правильный результат
- Опечатки в маркерах рассматривались как ошибка при использовании опции `--strict-markers`.

12.4.5 Маркировка классов и модулей

Декоратор `pytest.mark` можно применять для классов, чтобы пометить все его тестовые методы:

```
# content of test_mark_classlevel.py
import pytest


@pytest.mark.webtest
class TestClass:
    def test_startup(self):
        pass

    def test_startup_and_more(self):
        pass
```

Такая запись равносильна применению декоратора к обеим тестовым функциям.

Можно также установить атрибут `pytestmark` тестовому классу `TestClass` таким образом:

```
import pytest


class TestClass:
    pytestmark = pytest.mark.webtest
```

или назначить список маркеров:

```
import pytest


class TestClass:
    pytestmark = [pytest.mark.webtest, pytest.mark.slowtest]
```

Можно также установить пометку на уровне модуля:

import pytest pytestmark = pytest.mark.webtest

равно как и список маркеров:

pytestmark = [pytest.mark.webtest, pytest.mark.slowtest]

В этом случае маркеры будут применяться (слева направо) ко всем функциям и методам модуля.

12.4.6 Маркировка отдельных тестов при использовании параметризации

Если тест параметризован, то маркировка такого теста равносильна маркировке каждого экземпляра теста с конкретным параметром. Тем не менее, можно пометить и отдельный экземпляр параметризованного теста:

```
import pytest


@pytest.mark.foo
@pytest.mark.parametrize(
    ("n", "expected"), [(1, 2), pytest.param(1, 3, marks=pytest.mark.bar), (2, 3)]
)
def test_increment(n, expected):
    assert n + 1 == expected
```

В приведенном выше примере маркером «foo» окажутся помечены все три запускаемых теста, а вот маркер «bar» будет применен только ко второму. Тем же способом можно пометить `skip` и `xfail` тесты, см. *Skip/xfail с параметризацией*.

12.4.7 Настраиваемые маркеры и опции командной строки для контроля запуска тестов

Плагины могут предоставлять настраиваемые маркеры и реализовывать определенное поведение на их основе. Вот полноценный пример добавления опции командной строки и параметризованного маркера тестовой функции для запуска тестов в определенных виртуальных средах:

```
# content of conftest.py

import pytest


def pytest_addoption(parser):
    parser.addoption(
        "-E",
        action="store",
        metavar="NAME",
        help="only run tests matching the environment NAME.",
    )


def pytest_configure(config):
    # register an additional marker
    config.addinivalue_line(
        "markers", "env(name): mark test to run only on named environment"
    )
```

(continues on next page)

(продолжение с предыдущей страницы)

```
def pytest_runtest_setup(item):
    envnames = [mark.args[0] for mark in item.iter_markers(name="env")]
    if envnames:
        if item.config.getoption("-E") not in envnames:
            pytest.skip("test requires env in {!r}".format(envnames))
```

Вот тестовый файл с использованием этого плагина:

```
# content of test_someenv.py

import pytest


@pytest.mark.env("stage1")
def test_basic_db_operation():
    pass
```

и пример запуска теста в виртуальной среде, отличной от «stage1»:

```
$ pytest -E stage2
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_someenv.py s                                                    [100%]

========================== 1 skipped in 0.12s ===========================
```

А здесь мы запускаем тест в нужном виртуальном окружении:

```
$ pytest -E stage1
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 1 item

test_someenv.py .                                                    [100%]

========================== 1 passed in 0.12s ============================
```

Опцию `--markers` всегда можно использовать для получения актуального списка доступных маркеров:

```
$ pytest --markers
@pytest.mark.env(name): mark test to run only on named environment

@pytest.mark.filterwarnings(warning): add a warning filter to the given test. see https://docs.
↪pytest.org/en/latest/warnings.html#pytest-mark-filterwarnings

@pytest.mark.skip(reason=None): skip the given test function with an optional reason. Example:␣
↪skip(reason="no way of currently testing this") skips the test.
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@pytest.mark.skipif(condition): skip the given test function if eval(condition) results in a True␣
↪value.  Evaluation happens within the module global context. Example: skipif('sys.platform ==
↪"win32"') skips the test if we are on the win32 platform. see https://docs.pytest.org/en/latest/
↪skipping.html

@pytest.mark.xfail(condition, reason=None, run=True, raises=None, strict=False): mark the test␣
↪function as an expected failure if eval(condition) has a True value. Optionally specify a reason␣
↪for better reporting and run=False if you don't even want to execute the test function. If only␣
↪specific exception(s) are expected, you can list them in raises, and if the test fails in other␣
↪ways, it will be reported as a true failure. See https://docs.pytest.org/en/latest/skipping.html

@pytest.mark.parametrize(argnames, argvalues): call a test function multiple times passing in␣
↪different arguments in turn. argvalues generally needs to be a list of values if argnames␣
↪specifies only one name or a list of tuples of values if argnames specifies multiple names.␣
↪Example: @parametrize('arg1', [1,2]) would lead to two calls of the decorated test function, one␣
↪with arg1=1 and another with arg1=2.see https://docs.pytest.org/en/latest/parametrize.html for␣
↪more info and examples.

@pytest.mark.usefixtures(fixturename1, fixturename2, ...): mark tests as needing all of the␣
↪specified fixtures. see https://docs.pytest.org/en/latest/fixture.html#usefixtures

@pytest.mark.tryfirst: mark a hook implementation function such that the plugin machinery will try␣
↪to call it first/as early as possible.

@pytest.mark.trylast: mark a hook implementation function such that the plugin machinery will try␣
↪to call it last/as late as possible.
```

12.4.8 Передача callable-объекта настраиваемым маркерам

Вот конфигурационный файл, который будет использоваться в последующих примерах:

```
# content of conftest.py
import sys


def pytest_runtest_setup(item):
    for marker in item.iter_markers(name="my_marker"):
        print(marker)
        sys.stdout.flush()
```

Настраиваемый маркер может иметь свое множество позиционных и именованных аргументов, т. е. свойств `args` и `kwargs`, которые можно передать как с помощью вызова callable-объекта, так и спомощью `pytest.mark.MARKER_NAME.with_args`. В большинстве случаев оба метода работают одинаково.

Однако, если единственным позиционным аргументом является callable-объект без именованных аргументов, использование `pytest.mark.MARKER_NAME(c)` не передаст `"c"` в качестве позиционного аргумента, а просто обернет `"c"` нашим маркером (см. *маркировка тестов*). К счастью, на помощь приходит `pytest.mark.MARKER_NAME.with_args`

```
# content of test_custom_marker.py
import pytest


def hello_world(*args, **kwargs):
    return "Hello World"
```

(continues on next page)

(продолжение с предыдущей страницы)

```
@pytest.mark.my_marker.with_args(hello_world)
def test_with_args():
    pass
```

Результатом запуска будет:

```
$ pytest -q -s
Mark(name='my_marker', args=(<function hello_world at 0xdeadbeef>,), kwargs={})
.
1 passed in 0.12s
```

Мы видим, что у нашего настраиваемого маркера есть свое множество аргументов, одним из которых является функция `hello_world`. В этом и заключается ключевое различие между созданием маркера в качестве callable-объекта, который за кулисами вызывает `__call__`, и использованием `with_args`.

12.4.9 Считывание маркера, который используется в разных местах

Если вы активно используете маркеры в своем тестовом наборе, то можете столкнуться с ситуацией, когда маркер применяется к тестовой функции несколько раз. Вы можете посмотреть все эти случаи, настроив плагин. К примеру, у нас есть модуль:

```
# content of test_mark_three_times.py
import pytest

pytestmark = pytest.mark.glob("module", x=1)


@pytest.mark.glob("class", x=2)
class TestClass:
    @pytest.mark.glob("function", x=3)
    def test_something(self):
        pass
```

Здесь у нас маркер «glob» применяется к одной и той же функции три раза. Мы можем увидеть это, прописав в файле `conftest.py`:

```
# content of conftest.py
import sys


def pytest_runtest_setup(item):
    for mark in item.iter_markers(name="glob"):
        print("glob args={} kwargs={}".format(mark.args, mark.kwargs))
        sys.stdout.flush()
```

Давайте запустим без перехвата вывода и посмотрим, что получится:

```
$ pytest -q -s
glob args=('function',) kwargs={'x': 3}
glob args=('class',) kwargs={'x': 2}
glob args=('module',) kwargs={'x': 1}
.
1 passed in 0.12s
```

12.4.10 Маркировка зависящих от платформы тестов

Предпложим, что у нас есть тестовый набор, в котором мы используем маркеры `pytest.mark.darwin`, `pytest.mark.win32` и т. п. для маркировки тестов, запускаемых на разных платформах. При этом в набор такжде входят тесты, которые должны проводиться на всех платформах, и они никак не помечены. Теперь, если вы хотите запустить тесты для конкретной платформы, может пригодиться такой плагин:

```
# content of conftest.py
#
import sys
import pytest

ALL = set("darwin linux win32".split())


def pytest_runtest_setup(item):
    supported_platforms = ALL.intersection(mark.name for mark in item.iter_markers())
    plat = sys.platform
    if supported_platforms and plat not in supported_platforms:
        pytest.skip("cannot run on platform {}".format(plat))
```

При его использовании тесты для остальных платформ будут пропущены. Давайте напишем небольшой модуль, чтобы посмотреть, как это выглядит:

```
# content of test_plat.py

import pytest


@pytest.mark.darwin
def test_if_apple_is_evil():
    pass


@pytest.mark.linux
def test_if_linux_works():
    pass


@pytest.mark.win32
def test_if_win32_crashes():
    pass


def test_runs_everywhere():
    pass
```

Здесь два теста должны быть пропущены, а два выполнены:

```
$ pytest -rs # this option reports skip reasons
========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items

```

(continues on next page)

(продолжение с предыдущей страницы)

```
test_plat.py s.s.                                                    [100%]

========================= short test summary info ==========================
SKIPPED [2] $REGENDOC_TMPDIR/conftest.py:12: cannot run on platform linux
======================= 2 passed, 2 skipped in 0.12s =======================
```

Обратите внимание, что если вы определяете платформу с помощью маркера и опции `-m`, как показано ниже,

```
$ pytest -m linux
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items / 3 deselected / 1 selected

test_plat.py .                                                       [100%]

====================== 1 passed, 3 deselected in 0.12s ======================
```

то непомеченные тесты запускаться не будут. Таким образом, это способ ограничиться выполнением конкретных тестов.

12.4.11 Автоматическое добавление маркеров на основе имен тестов

Если в вашем тестовом наборе имена тестовых функций отражают виды выполняемых тестов, вы можете реализовать hook, который автоматически задает маркировку для использования с опцией `-m`. Взгляните на этот тестовый модуль:

```
# content of test_module.py


def test_interface_simple():
    assert 0


def test_interface_complex():
    assert 0


def test_event_simple():
    assert 0


def test_something_else():
    assert 0
```

Мы хотим динамически маркировать тесты и можем сделать это в `conftest.py`:

```
# content of conftest.py

import pytest


def pytest_collection_modifyitems(items):
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    for item in items:
        if "interface" in item.nodeid:
            item.add_marker(pytest.mark.interface)
        elif "event" in item.nodeid:
            item.add_marker(pytest.mark.event)
```

Теперь можно использовать для отбора опцию `-m`:

```
$ pytest -m interface --tb=short
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items / 2 deselected / 2 selected

test_module.py FF                                                  [100%]

================================ FAILURES ================================
_________________________ test_interface_simple _________________________
test_module.py:4: in test_interface_simple
    assert 0
E   assert 0
_________________________ test_interface_complex _________________________
test_module.py:8: in test_interface_complex
    assert 0
E   assert 0
====================== 2 failed, 2 deselected in 0.12s ======================
```

Или можно выполнить и «event», и «interface» тесты:

```
$ pytest -m "interface or event" --tb=short
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR
collected 4 items / 1 deselected / 3 selected

test_module.py FFF                                                 [100%]

================================ FAILURES ================================
_________________________ test_interface_simple _________________________
test_module.py:4: in test_interface_simple
    assert 0
E   assert 0
_________________________ test_interface_complex _________________________
test_module.py:8: in test_interface_complex
    assert 0
E   assert 0
__________________________ test_event_simple ___________________________
test_module.py:12: in test_event_simple
    assert 0
E   assert 0
====================== 3 failed, 1 deselected in 0.12s ======================
```

12.5 Фикстура уровня сессии для поиска во всех собираемых тестах

Фикстура уровня сессии имеет доступ ко всем собираемым тестам. Ниже - пример фикстуры, которая просматривает все собираемые тесты, ищет тестовый класс, содержащий метод `callme` и вызывает его:

```
# content of conftest.py

import pytest


@pytest.fixture(scope="session", autouse=True)
def callattr_ahead_of_alltests(request):
    print("callattr_ahead_of_alltests called")
    seen = {None}
    session = request.node
    for item in session.items:
        cls = item.getparent(pytest.Class)
        if cls not in seen:
            if hasattr(cls.obj, "callme"):
                cls.obj.callme()
            seen.add(cls)
```

Теперь в тестовых классах можно определить метод `callme`, который будет вызываться перед запуском любых тестов:

```
# content of test_module.py


class TestHello:
    @classmethod
    def callme(cls):
        print("callme called!")

    def test_method1(self):
        print("test_method1 called")

    def test_method2(self):
        print("test_method1 called")


class TestOther:
    @classmethod
    def callme(cls):
        print("callme other called")

    def test_other(self):
        print("test other")


# works with unittest as well ...
import unittest


class SomeTest(unittest.TestCase):
    @classmethod
```

(continues on next page)

(продолжение с предыдущей страницы)

```
    def callme(self):
        print("SomeTest callme called")

    def test_unit1(self):
        print("test_unit1 method called")
```

Заапустим этот код без перехвата вывода:

```
$ pytest -q -s test_module.py
callattr_ahead_of_alltests called
callme called!
callme other called
SomeTest callme called
test_method1 called
.test_method1 called
.test other
.test_unit1 method called
.
4 passed in 0.12s
```

12.6 Изменение стандартных правил поиска тестов Python

12.6.1 Игнорирование путей при поиске тестов

Можно игнорировать отдельные директории и файлы при сборке тестов, используя опцию командной строки `--ignore=path`. `pytest` позволяет указывать несколько опций `--ignore`. Пример:

```
tests/
|-- example
|   |-- test_example_01.py
|   |-- test_example_02.py
|   '-- test_example_03.py
|-- foobar
|   |-- test_foobar_01.py
|   |-- test_foobar_02.py
|   '-- test_foobar_03.py
'-- hello
    '-- world
        |-- test_world_01.py
        |-- test_world_02.py
        '-- test_world_03.py
```

Теперь, если вызвать `pytest` с опциями `--ignore=tests/foobar/test_foobar_03.py --ignore=tests/hello/`, то он соберет только те тестовые модули, которые не удовлетворяют указанному шаблону:

```
========================== test session starts ==========================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
rootdir: $REGENDOC_TMPDIR, inifile:
collected 5 items

tests/example/test_example_01.py .                                [ 20%]
tests/example/test_example_02.py .                                [ 40%]
tests/example/test_example_03.py .                                [ 60%]
```

(continues on next page)

(продолжение с предыдущей страницы)

```
tests/foobar/test_foobar_01.py .                                    [ 80%]
tests/foobar/test_foobar_02.py .                                    [100%]

========================= 5 passed in 0.02 seconds =========================
```

Опция `--ignore-glob` позволяет игнорировать пути в виде шаблонов Unix. К примеру, если вы хотите исключить тестовые модули, которые заканчиваются на `_01.py`, можно запустить `pytest` с опцией `--ignore-glob='*_01.py'`.

12.6.2 Отмена подбора теста

С помощью опции `--deselect=item` можно отменить подбор конкретного теста. Допустим, `tests/foobar/test_foobar_01.py` содержит `test_a` и `test_b`, и мы хотим выполнить все тесты из `tests/` *кроме* `tests/foobar/test_foobar_01.py::test_a`. Это можно сделать, запустив `pytest` с опцией `--deselect tests/foobar/test_foobar_01.py::test_a`. `pytest` поддерживает и множественное применение опции.

12.6.3 Повторение путей в командной строке

По умолчанию `pytest` игнорирует задвоенные пути в командной строке:

```
pytest path_a path_a

...
collected 1 item
...
```

Тесты будут собраны только один раз.

Чтобы собрать тесты дважды, используйте опцию `--keep-duplicates`. Пример:

```
pytest --keep-duplicates path_a path_a

...
collected 2 items
...
```

Из-за особенностей поиска, если вы 2 раза укажете один и тот же тестовый файл, `pytest` подберет его дважды, даже если опция `--keep-duplicates` не применялась. Пример:

```
pytest test_a.py test_a.py

...
collected 2 items
...
```

12.6.4 Изменение правил рекурсивного обхода

Опцию norecursedirs можно задавать в ini-файле, например, в корневом `pytest.ini` вашего проекта:

```
# content of pytest.ini
[pytest]
norecursedirs = .svn _build tmp*
```

Такая запись указывает `pytest` не проводить рекурсивный поиск тестов в «build»-директориях «sphinx» и в директориях с префиксом `tmp`

12.6.5 Изменение соглашений по именам тестов для поиска

Вы можете настроить свои правила для поиска тестов по имени, установив опции python_files, python_classes and python_functions. Вот примеры:

```
# content of pytest.ini
# Пример 1: pytest  должен искать "check" вместо "test"
# можно также определить в tox.ini или setup.cfg file, при этом секция
# имен в setup.cfg files должна быть "tool:pytest"
[pytest]
python_files = check_*.py
python_classes = Check
python_functions = *_check
```

Такая настройка заставит `pytest` искать файлы по шаблону `check_*.py`, классы - по префиксу `Check`, а функции и методы по шаблону `*_check`. Допустим, у нас есть:

```
# content of check_myapp.py
class CheckMyApp:
    def simple_check(self):
        pass

    def complex_check(self):
        pass
```

Тогда собранные тесты будут выглядеть вот так:

```
$ pytest --collect-only
========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 2 items
<Module check_myapp.py>
  <Class CheckMyApp>
      <Function simple_check>
      <Function complex_check>

========================== no tests ran in 0.12s ==========================
```

Можно проверять на соответствие нескольким глобальным шаблонам - в этом случае при описании шаблоны нужно разделить пробелом:

```
# Пример 2: pytest должен искать файлы с "test" и "example"
# вставляется в pytest.ini, tox.ini, или setup.cfg  (заменив "pytest"
# на "tool:pytest" в setup.cfg)
[pytest]
python_files = test_*.py example_*.py
```

Примечание: параметры `python_functions` и `python_classes` не оказывают никакого действия на поиск `unittest.TestCase`, поскольку обнаружение таких тестов производится средствами `unittest`.

12.6.6 Аргументы командной строки как имена пакетов

Чтобы заставить `pytest` интерпретировать аргументы как имена пакетов, получать их системные пути и запускать тесты, можно использовать опцию `--pyargs`. Например, если у вас установлен `unittest2`, можно выполнить

```
pytest --pyargs unittest2.test.test_skipping -q
```

для запуска соответствующего тестового модуля. Как и со всеми остальными опциями, можно сделать ее использование постоянным с помощью addopts в ini-файле:

```
# content of pytest.ini
[pytest]
addopts = --pyargs
```

После этого простой вызов вида `pytest NAME` будет проверять существование `NAME` В качестве пригодного для импорта модуля и рассматривать его как путь в файловой системе.

12.6.7 Просмотр дерева найденных тестов

Всегда можно посмоетреть дерево собранных тестов без их запуска:

```
. $ pytest --collect-only pythoncollection.py
=========================== test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 3 items
<Module CWD/pythoncollection.py>
  <Function test_function>
  <Class TestClass>
      <Function test_method>
      <Function test_anothermethod>

========================== no tests ran in 0.12s ===========================
```

12.6.8 Настройка поиска тестов

Можно легко заставить `pytest` искать тесты в любом `python`-файле:

```
# content of pytest.ini
[pytest]
python_files = *.py
```

Однако, во многих проектах есть файл `setup.py`, который не хотелось бы импортировать. Более того, там могут присутствовать файлы, которые можно импортировать только определенной версией `Python`. В таких случаях можно динамически определить игнорируемые файлы, перечислив их в `conftest.py`:

```
# content of conftest.py
import sys

collect_ignore = ["setup.py"]
if sys.version_info[0] > 2:
    collect_ignore.append("pkg/module_py2.py")
```

Пусть у вас есть такой модуль:

```
# content of pkg/module_py2.py
def test_only_on_python2():
    try:
        assert 0
    except Exception, e:
        pass
```

и макет файла `setup.py`:

```
# content of setup.py
0 / 0  # если будет импортирован - вызовет исключение
```

Тогда при запуске `pytest` в интерпретаторе `Python 2` мы соберем 1 тест, а файл“setup.py“ будет проигнорирован:

```
#$ pytest --collect-only
====== test session starts ======
platform linux2 -- Python 2.7.10, pytest-2.9.1, py-1.4.31, pluggy-0.3.1
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 1 items
<Module 'pkg/module_py2.py'>
  <Function 'test_only_on_python2'>

====== no tests ran in 0.04 seconds ======
```

А при запуске на `Python 3` будут проигнорированы все файлы:

```
$ pytest --collect-only
============================ test session starts ============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR, inifile: pytest.ini
collected 0 items

========================== no tests ran in 0.12s ===========================
```

Для определения файлов, которые должны быть пропущены, можно также добавлять в `collect_ignore_glob` шаблоны в стиле Unix

В следующем примере в `conftest.py` игнорируется файл `setup.py` и все файлы, которые оканчиваются на `*_py2.py` и запускаются с помощью `python` версии 3 и выше:

```
# content of conftest.py
import sys

collect_ignore = ["setup.py"]
if sys.version_info[0] > 2:
    collect_ignore_glob = ["*_py2.py"]
```

12.7 Работа с тестами не на python

12.7.1 Простой пример поиска тестов в файлах «yaml»

Вот пример файла `conftest.py` (полученного из плагина pytest-yamlwsgi). Этот `conftest.py` собирает файлы вида `test*.yaml` и выполняет их содержимое в виде тестов:

```
# content of conftest.py
import pytest


def pytest_collect_file(parent, path):
    if path.ext == ".yaml" and path.basename.startswith("test"):
        return YamlFile.from_parent(parent, fspath=path)


class YamlFile(pytest.File):
    def collect(self):
        import yaml  # we need a yaml parser, e.g. PyYAML

        raw = yaml.safe_load(self.fspath.open())
        for name, spec in sorted(raw.items()):
            yield YamlItem.from_parent(self, name=name, spec=spec)


class YamlItem(pytest.Item):
    def __init__(self, name, parent, spec):
        super().__init__(name, parent)
        self.spec = spec

    def runtest(self):
        for name, value in sorted(self.spec.items()):
            # some custom test execution (dumb example follows)
            if name != value:
                raise YamlException(self, name, value)

    def repr_failure(self, excinfo):
        """ called when self.runtest() raises an exception. """
        if isinstance(excinfo.value, YamlException):
            return "\n".join(
                [
                    "usecase execution failed",
                    "   spec failed: {1!r}: {2!r}".format(*excinfo.value.args),
                    "   no further details known at this point.",
                ]
            )

    def reportinfo(self):
        return self.fspath, 0, "usecase: {}".format(self.name)


class YamlException(Exception):
    """ custom exception for error reporting. """
```

Можно создать простой тестовый файл:

```
# test_simple.yaml
ok:
    sub1: sub1

hello:
    world: world
    some: other
```

И если вы установите PyYAML или совместимый YAML-парсер, то сможете запустить тесты:

```
nonpython $ pytest test_simple.yaml
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/nonpython
collected 2 items

test_simple.yaml F.                                                      [100%]

================================= FAILURES =================================
______________________________ usecase: hello ______________________________
usecase execution failed
   spec failed: 'some': 'other'
   no further details known at this point.
======================== 1 failed, 1 passed in 0.12s ========================
```

У нас получился один прошедший тест на проверку `sub1: sub1` и один упавший. Очевидно, вы можете захотеть реализовать в `conftest.py` более интересную интерпретацию значений. Так можно легко написать свой собственный «язык тестирования».

Примечание: `repr_failure(excinfo)` вызывается для представления упавших тестов. Если вы сами будете генерировать узлы, то сможете возвращать любое строковое представление ошибки по вашему выбору. В отчете оно будет выделено красным.

`reportinfo()` используется для представления расположения теста, а также в режиме `verbose`:

```
nonpython $ pytest -v
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y -- $PYTHON_PREFIX/bin/python
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/nonpython
collecting ... collected 2 items

test_simple.yaml::hello FAILED                                           [ 50%]
test_simple.yaml::ok PASSED                                              [100%]

================================= FAILURES =================================
______________________________ usecase: hello ______________________________
usecase execution failed
   spec failed: 'some': 'other'
   no further details known at this point.
======================== 1 failed, 1 passed in 0.12s ========================
```

При разработке собственного поиска и выполнения тестов интересно будет взглянуть на дерево собранных тестов:

```
nonpython $ pytest --collect-only
============================= test session starts ==============================
platform linux -- Python 3.x.y, pytest-5.x.y, py-1.x.y, pluggy-0.x.y
cachedir: $PYTHON_PREFIX/.pytest_cache
rootdir: $REGENDOC_TMPDIR/nonpython
collected 2 items
<Package $REGENDOC_TMPDIR/nonpython>
  <YamlFile test_simple.yaml>
    <YamlItem hello>
    <YamlItem ok>

========================== no tests ran in 0.12s ===========================
```