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`.