GeekBrains

Автоматизация тестирования консольных приложений Linux на Python

Лекция 3
Продвинутая работа с Pytest

Оглавление

Введение

Ранее в этом курсе мы узнали:

- Зачем писать автотесты.
- Почему их нужно писать на Python.
- Какие виды тестов можно автоматизировать.
- Как вызывать из Python команды операционной системы.
- Как писать простейшие автотесты с использованием Pytest.

В этом уроке мы подробнее познакомимся с фреймворком Pytest. Вы узнаете как и зачем использовать фикстуры на Pytest, где хранить параметры тестов, научитесь формировать отчёты в Pytest и настраивать его запуск при помощи файла конфигурации.

Это центральный урок нашего курса, здесь разбираются ключевые особенности Pytest, поэтому на уроке будет много демонстрации написания реального кода.

В реальной работе эта тема важна, поскольку, вероятнее всего, написание серьёзных тестовых проектов на Pytest потребует знания особенностей, которые мы сегодня изучим.

В частности, на этой лекции вы узнаете:

- Как вынести параметры тестов в отдельный файл.
- Как использовать тестовые фикстуры.
- Как вынести фикстуры в отдельный файл.
- Как использовать фикстуры с финализацией.
- Как формировать отчёты в различных форматах.
- Как конфигурировать запуск Pytest.

💡 Для более детального понимания последовательности работы с примерами посмотрите видеолекцию.

Термины

Декоратор — это паттерн проектирования (design pattern) в Python, а также функция второго уровня, то есть принимающая другие функции в качестве переменных и возвращающая их.

Фикстура — специальным образом оформленная вспомогательная функция в Pytest, которая выполняется до или после тестовой функции.

Декораторы и фикстуры

Декораторы — это обёртки вокруг Python-функций (или классов), которые изменяют работу того, к чему они применяются. В Python декораторы используются в основном для декорирования функций (или, соответственно, методов).

Декоратор — это функция, которая принимает функцию и возвращает функцию. Звучит странно? Рассмотрим пример.

Предположим, имеется функция, которую мы хотим запустить повторно в том случае, если при её первом запуске произойдёт сбой. То есть нам нужна функция (декоратор), которая вызывает нашу функцию один или два раза (это зависит от того, возникнет ли ошибка при первом вызове функции).

```
def retry(func):
    def wrapper():
        try:
            func()
        except:
            time.sleep(1)
            func()
    return wrapper

@retry
def might_fail():
    print("might_fail")
    raise Exception

might_fail()
```

Обратите внимание, что в этом примере мы декорируем функцию might_fail() с использованием конструкции, которая выглядит @retry. После имени декоратора нет круглых скобок. В результате получается, что когда мы, как обычно, вызываем функцию might_fail(), на самом деле, вызывается декоратор retry, которому передаётся, в виде первого аргумента, целевая функция (might_fail).

В Pytest очень важную роль играют специальные декораторы, которые называются фикстурами. Тестовые фикстуры инициализируют тестовые функции. Они обеспечивают надёжность тестов, согласованность и повторяемость их результатов.

При инициализации можно настраивать сервисы, состояния, переменные окружения. Доступ к ним осуществляется через аргументы тестовых функций.

Тестовые функции принимают фикстуры как входящий аргумент с тем же именем. Для каждого такого аргумента функция-фикстура предоставляет объект фикстуры. Чтобы зарегистрировать функцию как фикстуру, нужно использовать декоратор @pytest.fixture.

Фикстуры позволяют тестовым функциям легко получать предварительно инициализированные объекты и работать с ними, не заботясь об импорте/установке/очистке. Фикстуры могут выполнять работу, а могут возвращать данные в тестовую функцию.

Иногда хочется, чтобы фикстуры вызывались автоматически, без явного указания их в качестве аргумента. Для этого в аргументах такой фикстуры нужно указать *autouse=True.*

Давайте модернизируем наши тесты архиватора, с использованием фикстур.

Создадим фикстуры, которые создают тестовые каталоги и очищают их. Перед некоторыми шагами нам нужна будет очистка.

```
1 @pytest.fixture()
2 def make_folders():
3     return checkout("mkdir {} {} {} {}".format(folder_in,
  folder_out, folder_ext, folder_ext2), "")
4 @pytest.fixture()
5 def clear_folders():
6     return checkout("rm -rf {}/* {}/* {}/* {}/*".format(folder_in,
  folder_out, folder_ext, folder_ext2), "")
```

Также напишем фикстуру, которая создаёт тестовые файлы размером 1Мб со случайными именами длиной 5 символов. Пусть таких файлов пока будет 5.

```
1 @pytest.fixture()
2 def make_files():
3     list_off_files = [ ]
4     for i in range(5):
5         filename = ''.join(random.choices(string.ascii_uppercase +
  string.digits, k=5))
6         if checkout("cd {}; dd if=/dev/urandom of={} bs=1M count=1
  iflag=fullblock".format(folder_in, filename), ""):
7             list_off_files.append(filename)
8     return list_off_files
```

Также в одном из тестов нам понадобится создать подкаталог и файл в нём. Создадим для этого фикстуру, которая создаёт файл, каталог и возвращает их имена. Если какой-то объект не удалось создать, то вернётся None.

```
@pytest.fixture()
def make_subfolder():
    testfilename = ''.join(random.choices(string.ascii_uppercase + string.digits, k=5))
    subfoldername = ''.join(random.choices(string.ascii_uppercase + string.digits, k=5))
    if not checkout("cd {}; mkdir {}".format(folder_in, subfoldername), ""):
        return None, None
    if not checkout("cd {}/{}; dd if=/dev/urandom of={} bs=1M count=1 iflag=fullblock".format(folder_in, subfoldername, testfilename), ""):
        return subfoldername, None
    else:
        return subfoldername, testfilename
```

Передадим фикстуры в тесты. Перед первым тестом нам нужно создать каталоги, очистить их, а также создать файлы.

```
def test_step1(make_folders, clear_folders, make_files):
    # test1
    res1 = checkout("cd {}; 7z a {}/arx".format(folder_in, folder_out), "Everything is Ok")
    res2 = checkout("ls {}".format(folder_out), "arx.7z")
    assert res1 and res2, "test1 FAIL"
```

Во втором тесте у нас довольно много условий. Чтобы сократить запись проверки условий, будем добавлять результат каждого условия в список, а в конце проверять, что все элементы списка истинны.

```python
def test_step2(clear_folders, make_files):
    # test2
    res = []
    res.append(checkout("cd {}; 7z a {}/arx".format(folder_in, folder_out), "Everything is Ok"))
    res.append(checkout("cd {}; 7z e arx.7z -o{} -y".format(folder_out, folder_ext), "Everything is Ok"))
    for item in make_files:
        res.append(checkout("ls {}".format(folder_ext), item))
    assert all(res)
```

💡 При использовании фикстур нам нужно импортировать модуль pytest.

```python
def test_step3():
    # test3
    assert checkout("cd {}; 7z t arx.7z".format(folder_out), "Everything is Ok"), "test3 FAIL"

def test_step4():
    # test4
    assert checkout("cd {}; 7z u arx2.7z".format(folder_in), "Everything is Ok"), "test4 FAIL"
```

```python
def test_step5(clear_folders, make_files):
    # test5
    res = []
    res.append(checkout("cd {}; 7z a {}/arx".format(folder_in, folder_out), "Everything is Ok"))
    for i in make_files:
        res.append(checkout("cd {}; 7z l arx.7z".format(folder_out), i))
    assert all(res), "test5 FAIL"
```

В данном случае pytest использует следующий алгоритм для вызова тестовой функции:

1. pytest находит функцию test_step5 по её префиксу test_. Ей передаются аргументы с именами clear_folders и make_files, поэтому pytest ищет и находит функции с этими именами, помеченные как фикстура.

2. Фикстуры clear_folders и make_files вызываются для создания объекта-функции. Они очищают тестовые каталоги и создают тестовые файлы. Функция очистки не передаёт в тест ничего значимого. А функция создания файлов возвращает список их имён.

3. Затем вызывается сама функция test_step5, она проверяет наличие в выводе файлов, переданных ей через фикстуру make_files.

```
def test_step6(clear_folders, make_files, make_subfolder):
    # test6
    res = []
    res.append(checkout("cd {}; 7z a {}/arx".format(folder_in,
folder_out), "Everything is Ok"))
    res.append(checkout("cd {}; 7z x arx.7z -o{}
-y".format(folder_out, folder_ext2), "Everything is Ok"))
    for i in make_files:
        res.append(checkout("ls {}".format(folder_ext2), i))
    res.append(checkout("ls {}".format(folder_ext2),
make_subfolder[0]))
    res.append(checkout("ls {}/{}".format(folder_ext2,
make_subfolder[0]), make_subfolder[1]))
    assert all(res), "test6 FAIL"
```

```
def test_step7():
    # test7
    assert checkout("cd {}; 7z d arx.7z".format(folder_out),
"Everything is Ok"), "test7 FAIL"

def test_step8(clear_folders, make_files):
    # test8
    res = []
    for i in make_files:
        res.append(checkout("cd {}; 7z h {}".format(folder_in, i),
"Everything is Ok"))
        hash = getout("cd {}; crc32 {}".format(folder_in,
i)).upper()
        res.append(checkout("cd {}; 7z h {}".format(folder_in, i),
hash))
    assert all(res), "test8 FAIL"
```

Вынос параметров в отдельный файл

Хранить данные в коде нехорошо, поэтому мы вынесем их в отдельный файл. Это нужно для того, чтобы данные можно было легко изменить в одном месте. Причём, в конфигурационном файле изменить параметры безопасно может любой пользователь. Можно использовать разные форматы, одним из популярных и удобных является формат yaml.

YAML — это язык для хранения информации в понятном для человека формате. Его название расшифровывается «ещё один язык разметки». Однако, позже расшифровку изменили на «YAML не язык разметки», чтобы отличать его от настоящих языков разметки.

YAML обычно применяют для создания конфигурационных файлов в программах виртуализации, или для управления контейнерами в работе DevOps.

Всё больше и больше компаний используют DevOps и виртуализацию, поэтому знание YAML — это очень важно для современного разработчика. Кроме того, YAML легко интегрировать, благодаря поддержке Python (используя PyYAML библиотеку, Docker или Ansible) и других популярных технологий.

Особенности yaml:

- В синтаксисе YAML-файлов используется система отступов, как в Python. Необходимо использовать пробелы, а не табуляцию, чтобы избежать путаницы.
- Это избавляет от лишних символов, которые есть в JSON и XML (кавычки, скобки, фигурные скобки).В итоге читаемость файла значительно повышается.
- Большинство данных в YAML-файле хранятся в виде пары ключ-значение, разделённых двоеточием, где ключ — это имя пары, а значение — связанные данные.
- Кавычки в YAML не нужны.
- Для работы с yaml в python нужен установленный модуль pyyaml.
- В целях безопасности вы всегда должны использовать yaml.safe_load и yaml.safe_dump в качестве стандартных методов ввода/вывода для YAML (методы без префикса save небезопасны, поскольку могут загрузить вредоносный код).

YAML

```yaml
simple-property: a simple value

object-property:
  a-property: a value
  another-property: another value


array-property:
  - item-1-property-1: one
    item-1-property-2: 2
  - item-2-property-1: three
    item-2-property-2: 4

# no comment in JSON
```

JSON

```json
{
  "simple-property": "a simple value",

  "object-property": {
    "a-property": "a value",
    "another-property": "another value"
  },

  "array-of-objects": [
    { "item-1-property-1": "one",
      "item-1-property-2": 2 },
    { "item-2-property-1": "three",
      "item-2-property-2": 4 }
  ]
}
```

Наш конфигурационный файл config.yaml будет выглядеть следующим образом.

```yaml
folder_in: /home/zerg/tst
folder_out: /home/zerg/out
folder_ext: /home/zerg/folder1
folder_ext2: /home/zerg/folder2
count: 5
```

Импорт данных и обращение к ним выглядят так:

```python
import yaml


with open('config.yaml') as f:
    # читаем документ YAML
    data = yaml.safe_load(f)


@pytest.fixture()
def make_folders():
    return checkout("mkdir {} {} {} {}".format(data["folder_in"],
  data["folder_in"], data["folder_ext"], data["folder_ext2"]), "")
```

Работа с conftest.py

Если вы планируете использовать фикстуру в нескольких тестах (файлах с тестами), то можно объявить её в специальном файле conftest.py. При этом импортировать её не нужно, pytest найдёт её автоматически.

Поиск фикстур начинается с тестовых классов (у нас их пока нет, но сейчас мы их напишем), затем они ищутся в тестовых модулях (файлах с тестами) и в файлах conftest.py, и, в последнюю очередь, во встроенных и сторонних плагинах.

Давайте перепишем наш тест, создав тестовый класс и перенеся фикстуры в conftest.py.

```python
import pytest
from checkers import checkout
import random, string
import yaml

with open('config.yaml') as f:
    # читаем документ YAML
    data = yaml.safe_load(f)

@pytest.fixture()
def make_folders():
    return checkout("mkdir {} {} {} {}".format(data["folder_in"], data["folder_in"], data["folder_ext"], data["folder_ext2"]), "")

@pytest.fixture()
def clear_folders():
    return checkout("rm -rf {}/* {}/* {}/* {}/*".format(data["folder_in"], data["folder_in"], data["folder_ext"], data["folder_ext2"]), "")
@pytest.fixture()
def make_files():
    list_off_files = [ ]
    for i in range(5):
        filename = ''.join(random.choices(string.ascii_uppercase + string.digits, k=5))
        if checkout("cd {}; dd if=/dev/urandom of={} bs=1M count=1 iflag=fullblock".format(data["folder_in"], filename), ""):
            list_off_files.append(filename)
    return list_off_files

@pytest.fixture()
def make_subfolder():
    testfilename = ''.join(random.choices(string.ascii_uppercase + string.digits, k=5))
    subfoldername = ''.join(random.choices(string.ascii_uppercase + string.digits, k=5))
    if not checkout("cd {}; mkdir {}".format(data["folder_in"], subfoldername), ""):
        return None, None
    if not checkout("cd {}/{}; dd if=/dev/urandom of={} bs=1M count=1 iflag=fullblock".format(data["folder_in"], subfoldername, testfilename), ""):
        return subfoldername, None
    else:
        return subfoldername, testfilename
```

```python
from checkers import checkout, getout
import yaml

with open('config.yaml') as f:
    # читаем документ YAML
    data = yaml.safe_load(f)

class TestPositive:
    def test_step1(self, make_folders, clear_folders, make_files):
        # test1
        res1 = checkout("cd {}; 7z a {}/arx".format(data["folder_in"], data["folder_out"]), "Everything is Ok")
        res2 = checkout("ls {}".format(data["folder_out"]), "arx.7z")
        assert res1 and res2, "test1 FAIL"
```

Как видите, код в нашем файле с тестами сократился и читать его стало удобнее.

Финализаторы в фикстуре

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 после завершения работы тестовой функции.

Дополнительно создадим фикстуру, которая будет отображать время перед стартом шага теста и время после его завершения.

```python
@pytest.fixture(autouse=True)
def print_time():
    print("Start: {}".format(datetime.now().strftime("%H:%M:%S.%f")))
    yield
    print("Finish: {}".format(datetime.now().strftime("%H:%M:%S.%f")))
```

Отчётность

Когда pytest запущен в командной строке, он иногда выводит непонятную стену текста. Визуальные отчёты гораздо лучше представляют информацию о результатах тестирования, особенно для тех, кто не связан с разработкой.

Мы рассмотрим два вида отчётности в форматах junitxml и html.

Чтобы сгенерировать отчёт в формате junitxml, нужно выполнить команду

pytest test.py --junitxml=report.xml

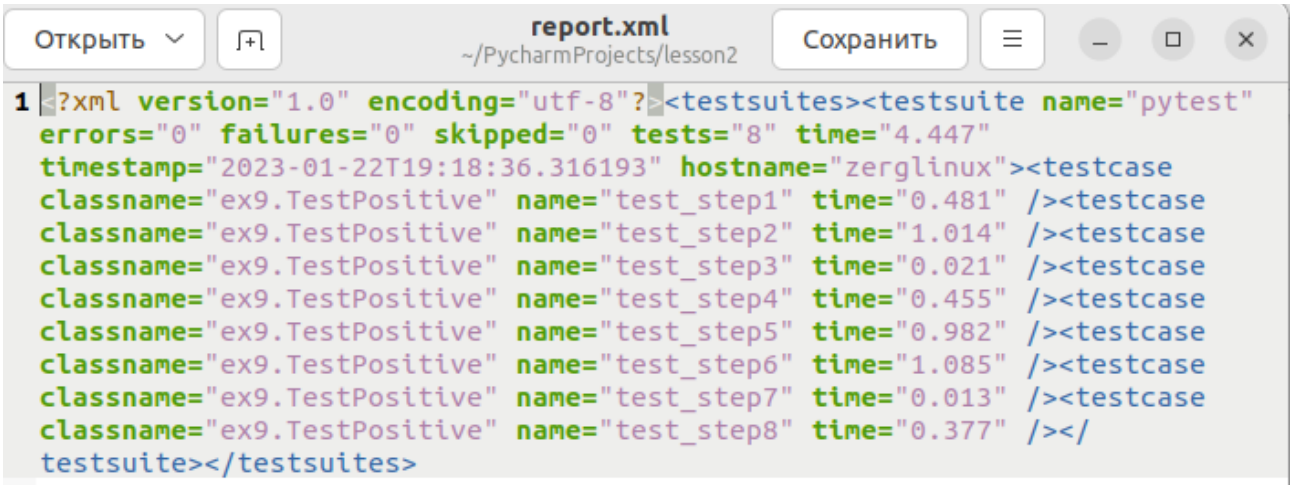
Что это за формат?

Это формат xml-отчётов, совместимый с фреймворком тестирования JUnit. Ниже информация об этом фреймворке:

Возникает вопрос, не перепутал ли преподаватель курс?

Нет, JUnit довольно универсальный формат, с которым умеет работать множество прикладных систем. Нет ничего плохого, что мы будем формировать такой отчёт при помощи python.

Формат junitxml удобен в первую очередь для обмена информацией между различными информационными системами. Для человека такой отчёт не очень удобен.

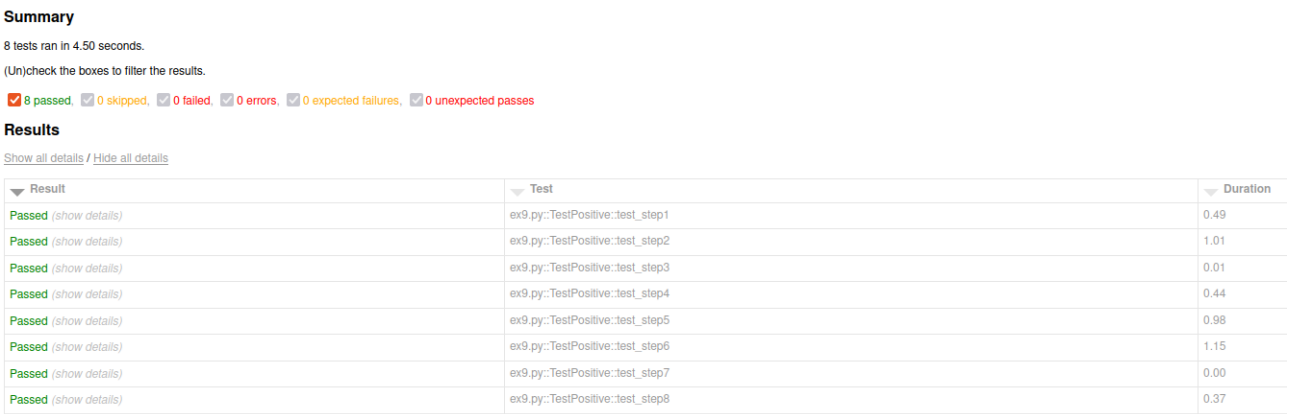


Добавление плагина pytest-html к вашим тестам позволит вам создавать симпатичные HTML-отчёты всего одной простой опцией командной строки.

pip install pytest-html

pytest test.py --html=report.html

В таком отчёте содержится подробная информация о тестах, включая консольный вывод.



Summary

8 tests ran in 4.50 seconds.

(Un)check the boxes to filter the results.

8 passed, 0 skipped, 0 failed, 0 errors, 0 expected failures, 0 unexpected passes

Results

Show all details / Hide all details

Result	Test	Duration
Passed (show details)	ex9.py::TestPositive::test_step1	0.49
Passed (show details)	ex9.py::TestPositive::test_step2	1.01
Passed (show details)	ex9.py::TestPositive::test_step3	0.01
Passed (show details)	ex9.py::TestPositive::test_step4	0.44
Passed (show details)	ex9.py::TestPositive::test_step5	0.98
Passed (show details)	ex9.py::TestPositive::test_step6	1.15
Passed (show details)	ex9.py::TestPositive::test_step7	0.00
Passed (show details)	ex9.py::TestPositive::test_step8	0.37

```
Passed (hide details)    ex9.py::TestPositive::test_step3    0.01

----------------------------Captured stdout setup-----------------------------
Start: 19:43:47.735787

----------------------------Captured stdout call-----------------------------

7-Zip [64] 16.02 : Copyright (c) 1999-2016 Igor Pavlov : 2016-05-21
p7zip Version 16.02 (locale=ru_RU.UTF-8,Utf16=on,HugeFiles=on,64 bits,12 CPUs AMD Ryzen 5 4600H with Radeon Graphics          (860F01),ASM,AES-NI)

Scanning the drive for archives:
1 file, 10486817 bytes (11 MiB)

Testing archive: arx.7z
--
Path = arx.7z
Type = 7z
Physical Size = 10486817
Headers Size = 317
Method = LZMA2:6m
Solid = +
Blocks = 2

Everything is Ok

Folders: 1
Files: 11
Size:       10485940
Compressed: 10486817


----------------------------Captured stdout teardown-----------------------------
Finish: 19:43:47.746277
```

Для pytest есть ещё более продвинутый плагин для генерации красивых отчётов.

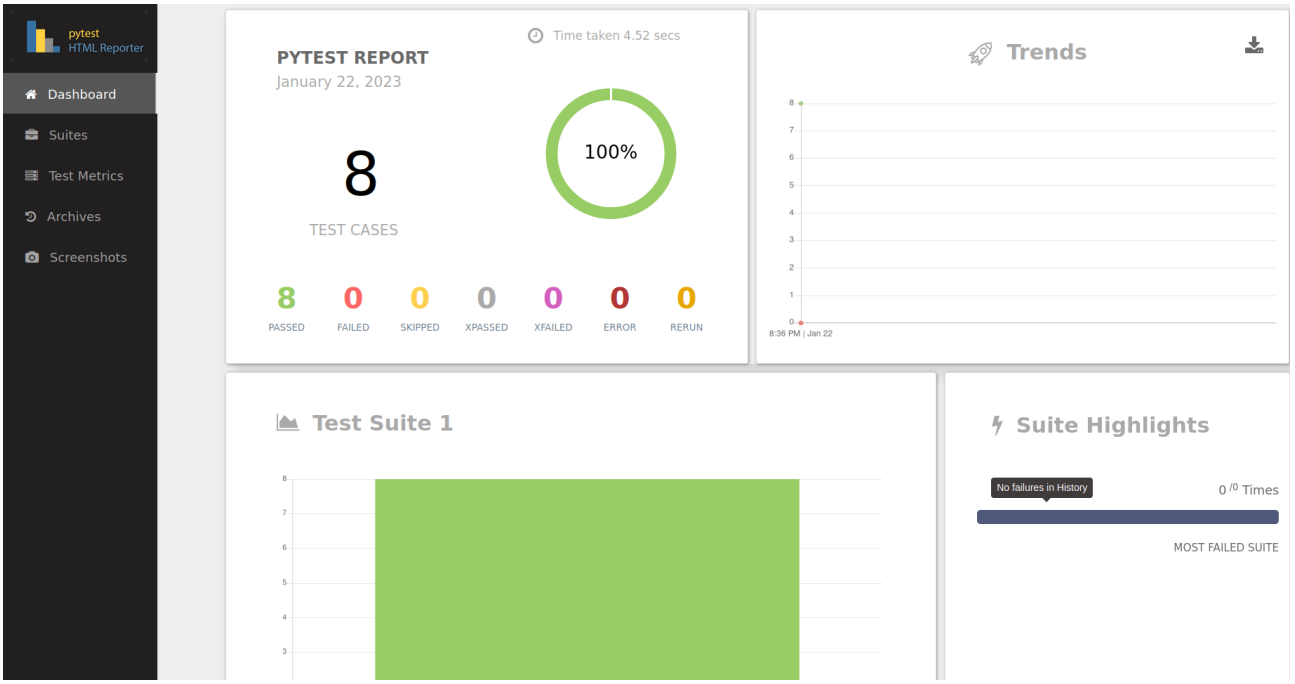
Устанавливается он командой

pip3 install pytest-html-reporter

Отчёт генерируется запуском pytest с ключом.

--html-report=report.html

Выглядит такой отчёт очень современно.



Файл pytest.ini

В предыдущих примерах нам приходилось запускать pytest с различными ключами. Возникает вопрос, нельзя ли сохранить нужный нам набор ключей, чтобы каждый раз не вводить его.

В этом, а также во многом другом нам поможет *pytest.ini* — основной файл конфигурации Pytest, который позволяет вам изменить поведение по умолчанию.

Параметр addopts позволяет сохранить параметры, их не надо будет вводить вручную. Они будут добавляться при каждом запуске pytest.

```
1 [pytest]
2 addopts = -rsxX -l --tb=short --strict
```

Параметр minversion позволяет указать минимальную версию pytest, ожидаемую для тестов. Например, я задумал использовать approx() при тестировании чисел с плавающей запятой для определения «достаточно близкого» равенства в тестах. Но эта функция не была введена в pytest до версии 3.0. Чтобы избежать путаницы, я добавляю следующее в проекты, которые используют approx():

```
1 [pytest]
2 minversion = 3.0
```

Таким образом, если кто-то пытается запустить тесты, используя более старую версию pytest, появится сообщение об ошибке.

testspaths — это список каталогов относительно корневого каталога для поиска тестов. Он используется только в том случае, если в качестве аргумента не указан каталог, файл или nodeid.

```
1 [pytest]
2 testpaths = tests
```

Теперь, если вы запускаете pytest из каталога tasks_proj, pytest будет искать только в tasks_proj/tests.

python_files изменяет правило обнаружения тестов по умолчанию, которое заключается в поиске файлов, начинающихся с test_* или имеющих в конце *_test.

Допустим, у вас есть пользовательский тестовый фреймворк, в котором вы назвали все свои тестовые файлы check_<something>.py. Кажется разумным. Вместо того чтобы переименовывать все ваши файлы, просто добавьте строку в pytest.ini следующим образом:

```
1 [pytest]
2 python_files = test_* *_test check_*
```

python_functions действует как и предыдущая настройка, но для тестовых функций и имён методов. Значение по умолчанию — test_*. А чтобы добавить check_*, сделайте это:

```
1 [pytest]
2 python_functions = test_*  check_*
```

Однако, для сохранения совместимости, лучше не менять правила обнаружения тестов без очень веской причины.

Установка xfail_strict = true приводит к тому, что тесты, помеченные @pytest.mark.xfail, не распознаются, как вызвавшие ошибку. Это так называемая «Маркировка тестов ожидающих сбоя».

Итоги

Итак, на этой лекции вы узнали, как работать с фреймворком Pytest на продвинутом уровне, научились писать фикстуры, формировать отчёты и конфигурировать запуск тестов.

Вы узнали:

- Как вынести параметры тестов в отдельный файл.
- Как использовать тестовые фикстуры.
- Как вынести фикстуры в отдельный файл.
- Как использовать фикстуры с финализацией.
- Как формировать отчёты в различных форматах.
- Как конфигурировать запуск Pytest.

Этих знаний достаточно, чтобы работать с Pytest и писать на нём достаточно сложные тесты.

Что можно почитать ещё?

1. Здесь описана генерация отчётов для Allure с pytest статья на habr.
2. Здесь описана работа с yaml.
3. Хорошая книга про pytest (на английском) Python Testing with pytest, Second Edition by Brian Okken. Достаточно нескольких первых глав.

Используемая литература

1. Python Testing with pytest, Second Edition by Brian Okken.
2. Документация pytest.
3. Описание yaml.