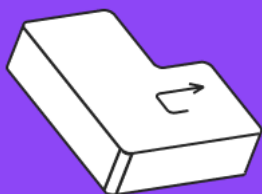




Автоматизация тестирования консольных приложений Linux на Python

Лекция 2

Знакомство с фреймворком Pytest



Оглавление

Введение	3
Повторение материала, работа с stdout	3
Оператор assert в Python	6
Фреймворки для тестирования на Python	8
Использование Pytest	9
Как выглядит запуск тестов через pytest?	10
Ключи и параметры Pytest	11
Итоги	14
Что можно почитать ещё?	15
Используемая литература	15

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.

В этом уроке мы познакомимся с фреймворками для автоматизации тестирования на Python и, в частности, с популярным фреймворком Pytest. Вы узнаете, как писать тесты на Pytest, запускать их с использованием различных ключей и параметров.

Для нашего курса это очень важная тема, потому что далее в курсе мы будем развивать проект автоматизации тестов именно на Pytest.

В реальной работе и жизни эта тема также важна, поскольку Pytest является одним из самых популярных и востребованных фреймворков. Изучив Pytest, вы сможете использовать в своей будущей профессиональной деятельности, упростив и систематизировав написание автотестов.

В частности, на этой лекции вы узнаете:

- Как использовать оператор `assert`.
- Какие есть фреймворки для тестирования на Python.
- Как использовать Pytest.
- Какие есть ключи и параметры Pytest.



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

Повторение материала, работа с `stdout`

На прошлом семинаре мы написали первый небольшой тест с использованием `subprocess.run()`. Вспомним, как он работает, и напишем более сложный тест знакомого нам из первой лекции архиватора 7z (только теперь для Линукс).

```

1 import subprocess
2 if __name__ == '__main__':
3     # test1
4     result1 = subprocess.run("cd /home/zerg/tst; 7z a ../out/arc2",
5                               shell=True, stdout=subprocess.PIPE, encoding='utf-8')
6     if result1.returncode == 0 and "Everything is Ok" in
7     result1.stdout:
8         print("test1 SUCCESS")
9     else:
10        print("test1 FAIL")
11    # test2
12    result2 = subprocess.run("cd /home/zerg/out; 7z e arc2.7z
13                             /home/zerg/folder1", shell=True, stdout=subprocess.PIPE,
14                             encoding='utf-8')
15    if result2.returncode == 0 and "Everything is Ok" in
16    result2.stdout:
17        print("test2 SUCCESS")
18    else:
19        print("test2 FAIL")
20    # test3
21    result3 = subprocess.run("cd /home/zerg/out; 7z t arc2.7z",
22                              shell=True, stdout=subprocess.PIPE,
23                              encoding='utf-8')
24    if result3.returncode == 0 and "Everything is Ok" in
25    result3.stdout:
26        print("test3 SUCCESS")
27    else:
28        print("test3 FAIL")

```

Итак, мы написали **три теста**.

Первый проверяет, что команда архивации содержит текст «Everything is OK» и завершается с кодом 0.

Второй проверяет, что команда разархивации содержит текст «Everything is OK» и завершается с кодом 0.

Третий проверяет, что команда тестирования файлов в архиве содержит текст «Everything is OK» и завершается с кодом 0.

Обратим внимание на получение вывода запущенной программы.

Консольная программа обычно работает с тремя потоками данных:

- stdin — поток ввода;
- stdout — поток вывода;
- stderr — поток ошибок.

Считается, что ошибки программы должны писаться в поток ошибок. Но не всегда это так. Некоторые программы пишут ошибки только в stdout, некоторые в stderr, а некоторые и туда, и туда. Поэтому при построении негативных сценариев тестирования нам важно читать поток stderr программы, информации об ошибках в stdin может и не быть.

Ниже представлены значения, которые могут принимать аргументы stdin, stdout или stderr модуля subprocess.

subprocess.DEVNULL

Специальное значение, которое может использоваться в качестве аргумента stdin, stdout или stderr и указывает, что будет использоваться специальный файл os.devnull.

```
1 import subprocess
2 result = subprocess.run(['ping', '-c', '3', '-n', 'yandex.ru'],
3
4                               stdout=subprocess.DEVNULL,
5                               encoding='utf-8')
6 print(result.stdout)
7 # None
```

subprocess.PIPE

Специальное значение, которое указывает, что канал к стандартному потоку должен быть открыт. При его указании в stdout туда попадает стандартный поток stdout.

```
1 import subprocess
2 result = subprocess.run(['ping', '-c', '3', '-n', 'yandex.ru'],
3
4                               stdout=subprocess.PIPE, encoding='utf-8')
5 print(result.stdout)
6 # PING yandex.ru (5.255.255.50) 56(84) bytes of data:
7 # 64 bytes from 5.255.255.50: icmp_seq=1 ttl=249 time=14.5 ms
8 # 64 bytes from 5.255.255.50: icmp_seq=2 ttl=249 time=14.5 ms
9 # 64 bytes from 5.255.255.50: icmp_seq=3 ttl=249 time=14.6 ms
10 # — yandex.ru ping statistics —
11 # 3 packets transmitted, 3 received, 0% packet loss, time 2003ms
12 # rtt min/avg/max/mdev = 14.551/14.578/14.619/0.102 ms
13
```

💡 subprocess.STDOUT

Значение, которое может использоваться в качестве аргумента `stderr` и указывает, что поток ошибок `stderr` должен быть добавлен к стандартному выводу `stdout`.

```
1 import subprocess
2 result = subprocess.run(['ping', '-c', '3', '-n', 'host.host'], \
3                           stderr=subprocess.STDOUT, \
4                           stdout=subprocess.PIPE, encoding='utf-8')
5 result.stdout
6 # 'ping: host.host: Неизвестное имя или служба\n'
7
```

Оператор `assert` в Python

Сейчас мы немного оптимизируем наш код, оформив проверки в виде функций и используя вместо проверок `if-else` оператор `assert`.

Оператор **`assert`** — это встроенный оператор в Python, используемое для отладки кода.

Оператор `assert` работает как логическое выражение, проверяя, является ли заданное условие истинным или ложным. Если условие истинно, то ничего не происходит и выполняется следующая строка кода. Если же условие ложно, оператор `assert` останавливает выполнение программы и выдаёт ошибку. В этом случае `assert` работает, как ключевое слово `raise` и выводит исключение. Исключение, вызванное оператором `assert`, также называется `AssertionError`.

Синтаксис использования оператора `assert` следующий:

`assert <condition>, <message>`

Предположим, что нам нужно оформить тест таким образом, чтобы его шаги были зависимы, то есть в случае возникновения ошибки на каком-то шаге программа завершалась и следующие шаги не выполнялись.

Мы перепишем программу следующим образом:

```

1 import subprocess
2 def checkout(cmd, text):
3     result = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE,
4                             encoding='utf-8')
5     if text in result.stdout and result.returncode == 0:
6         return True
7     else:
8         return False
9
10 if __name__ == '__main__':
11     # test1
12     assert checkout("cd /home/zerg/tst; 7z a ../out/arx2",
13                    "Everything is Ok"), "test1 FAIL"
14     # test2
15     assert checkout("cd /home/zerg/out; 7z e arx2.7z /home/zerg
16                    /folder1", "Everything is Ok"), "test2 FAIL"
17     # test3
18     assert checkout("cd /home/zerg/out; 7z t arx2.7z", "Everything
19                    is Ok"), "test3 FAIL"

```

Этот код уже выглядит более компактным и удобным.

Для небольших тестовых проектов этим можно было бы и ограничиться. Но если вы пишете большой и сложный проект, то вам не обойтись без дополнительных инструментов, которые предоставляют множество удобных функций и дополнительных возможностей.

Такие инструменты называются фреймворками.

Фреймворк — программная платформа, определяющая структуру программной системы; программное обеспечение, облегчающее разработку и объединение разных компонентов большого программного проекта.

«Фреймворк» отличается от понятия библиотеки тем, что библиотека может быть использована в программном продукте просто как набор подпрограмм близкой функциональности, не влияя на архитектуру программного продукта и не накладывая на неё никаких ограничений. В то время как «фреймворк» диктует правила построения архитектуры приложения, задавая на начальном этапе разработки поведение по умолчанию — «каркас», который нужно будет расширять и изменять согласно указанным требованиям.







Например, используемая нами `subprocess` — библиотека, поскольку она просто предоставляет нам методы выполнения команд внутри ОС.

В качестве примера фреймворка можно привести Django — это не просто одна библиотека, а целый набор библиотек и инструментов для веб-разработки.

Фреймворки для тестирования на Python

Существует множество фреймворков, облегчающих написание тестов на Python. Все они имеют свои сильные и слабые стороны. Некоторые из них лучше подходят для модульного тестирования, некоторые для функционального, другие же универсальны.

В таблице ниже приведены несколько наиболее популярных фреймворков.

	Лицензия	Установка	Применение	Особенности
Robot 	Free software (ASF License)	библиотека	Приёмочное тестирование	управление ключевыми словами
PyTest 	Free software (MIT License)	отдельная	Модульное, приёмочное, функциональное тестирование	классы и фикстуры для упрощения тестирования
unittest 	Free software (MIT License)	часть стандартной библиотеки	Модульное тестирование	быстрота и гибкость
DocTest 	Free software (MIT License)	часть стандартной библиотеки	Модульное тестирование	использование интерактивной командной строки Python при работе с тестируемым приложением
Nose2 	Free software (BSD License)	Плагин для unittest	Продвинутое модульное тестирование	Большое число плагинов
Testify 	Free software (ASF License)	плагин для unittest	Продвинутое модульное тестирование	Расширяет функциональность unittest и nose2

В своей практике я использовал unittest и pytest. Как видим, многие фреймворки (Testify, Nose2 расширяют возможности unittest). Однако, написанием модульных тестов чаще всего заняты программисты. Довольно интересным является фреймворк Robot, информация по нему есть в списке литературы.

Использование Pytest

В этом курсе нас прежде всего интересует функциональное тестирование. Поэтому мы из приведённого выше списка выберем фреймворк Pytest.

Pytest это надёжный инструмент тестирования в Python, он может быть использован практически для всех типов и уровней тестирования программного обеспечения. Многие проекты по всему интернету (включая Mozilla и Dropbox)[2] переключились с использования unittest или nose на pytest. Почему? Потому что pytest предлагает мощные функции, такие как переопределение операции assert, сторонние модели плагинов и мощную, но простую модель фикстур (о ней мы поговорим на следующем занятии).

Pytest — это больше, чем фреймворк, это платформа для тестирования программного обеспечения. Это программа командной строки, предоставляющая собой инструмент, который автоматически находит написанные тесты, запускает их и пишет отчёты с результатом. Она имеет библиотеку плагинов, которые можно использовать в тестах, чтобы тестировать более эффективно. Он может быть расширен путём написания собственных плагинов или. Его можно использовать для тестирования дистрибутивов Python. И он легко интегрируется с другими инструментами, такими как непрерывная интеграция и веб-автоматизация.

Вот несколько причин, почему pytest выделяется среди многих других тестовых фреймворков:

- Простые тесты легко написать в pytest.
- Сложные тесты тоже просто писать.
- Тесты очень легко читаются.
- Вы можете начать работу в pytest очень быстро.
- Можно использовать pytest для запуска тестов, написанных для unittest или nose.

Как выглядит запуск тестов через pytest?

В командной строке запускается команда `pytest`, которому можно передать имя тестового файла.

PyTest сам собирает все тесты по имени `test_*` (`Test_*` для имени классов) для всех файлов в папке (рекурсивно обходя вложенные папки) или же для указанного файла.

Есть требования к наименованию тестовых файлов и директорий, чтобы Pytest автоматически мог их найти.

- Имя файла должно начинаться с `test` или заканчиваться на `test.py`.
- Имя тестовой функции внутри файла должно начинаться с `test_`, например, `test_gb`.

```
1 import subprocess
2 def checkout(cmd, text):
3     result = subprocess.run(cmd, shell=True, stdout=subprocess.PIPE,
4                             encoding='utf-8')
5     if text in result.stdout and result.returncode == 0:
6         return True
7     else:
8         return False
9
10 def test_step1():
11     # test1
12     assert checkout("cd /home/zerg/tst; 7z a ../out/arx2",
13                    "Everything is Ok"), "test1 FAIL"
14
15 def test_step2():
16     # test2
17     assert checkout("cd /home/zerg/out; 7z e arx2.7z -o/home
18                    /zerg/folder1 -y", "Everything is Ok"), "test2 FAIL"
19
20 def test_step3():
21     # test3
22     assert checkout("cd /home/zerg/out; 7z t arx2.7z", "Everything
23                    is Ok"), "test3 FAIL"
```

Вопрос 1: Assert из предыдущего примера и `assert` из последнего примера один и тот же?

Ответ: Нет, так как первый прерывает выполнение остальных тестов, а второй – нет.

Вопрос 2: Почему мы не импортировали модуль `pytest` в файле с тестами?

Ответ: В этом нет необходимости, т. к. файл запускается не напрямую, а передаётся в `pytest`.

Ключи и параметры Pytest

Рассмотрим некоторые опции запуска pytest. Это далеко не полный список, но этих опций для начала вполне достаточно.

```
1 (venv) zerg@zerglinux:~/PycharmProjects/lesson2$ pytest -h
2 usage: pytest [options] [file_or_dir] [file_or_dir] [... ]
3
4 positional arguments:
5   file_or_dir
6
7 general:
8   -k EXPRESSION          Only run tests which match the given
                          substring expression. An expression is a Python evaluable
                          expression where all names are substring-matched against test names
                          and their parent classes.
9   --sw-skip, --stepwise-skip
10  Ignore the first failing test but stop on the next failing test.
    Implicitly enables --stepwise.
11
```

--collect-only

Параметр `--collect-only` показывает, какие тесты будут выполняться с заданными параметрами и конфигурацией, но не запускает их.

Параметр `--collect-only` полезен для проверки правильности выбора других опций, которые выбирают тесты перед запуском тестов.

-k EXPRESSION

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

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

```
1 $ cd /path/to/code/ch1
2 $ pytest -k "asdict or defaults" --collect-only
```

-m MARKEXPR

Маркеры-один из лучших способов пометить подмножество тестовых функций для совместного запуска. В качестве примера, один из способов запустить `test_replace()` и `test_member_access()`, даже если они находятся в отдельных файлах, это пометить их. Можно использовать любое имя маркера.

Допустим, вы хотите использовать `run_these_please`. Отметим тесты, используя декоратор `@pytest.mark.run_these_please`:

```
1 import pytest
2 ...
3 @pytest.mark.run_these_please
4 def test_member_access():
5 ...
```

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

```
1 $ cd /path/to/code/ch1/tasks
2 $ pytest -v -m run_these_please
```

Выражение маркера не обязательно должно быть одним маркером. Вы можете использовать такие варианты, как `-m "mark1 and mark2"` для тестов с обоими маркерами, `-m "mark1 and not mark2"` для тестов, которые имеют метку 1, но не метку 2, `-m "mark1 or mark2"` для тестов с одним из и т. д.

-x, --exitfirst

Нормальным поведением `pytest` является запустить все тесты, которые он найдёт. Если тестовая функция обнаружит сбой `assert` или `exception`, выполнение этого теста останавливается, и тест завершается ошибкой. И тогда `pytest` запускает следующий тест. Обычно это решение подходит, однако, это не удобно при отладке проблемы. Опция `-x` прерывает тесты при ошибке (как стандартный `assert python`).

--maxfail=num

Параметр `-x` приводит к остановке после первого отказа теста. Если вы хотите, чтобы некоторое количество сбоев было допущено, но не очень много, используйте параметр `--maxfail`, чтобы указать, сколько ошибок допускается получить.

`-s` и `--capture=method`

Флаг `-s` позволяет печатать операторы — или любой другой вывод, который обычно печатается в *stdout*, чтобы фактически быть напечатанным в стандартном выводе во время выполнения тестов. Это сокращённый вариант для `--capture=no`.

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

`-lf`, `--last-failed`

При сбое одного или нескольких тестов способ выполнения только неудачных тестов, полезен для отладки. `--ff`, `--failed-first`

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

`-v`, `--verbose`

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

`-q`, `--quiet`

Опция `-q/--quiet` противоположна `-v/--verbos`. Она сокращает объём информации в отчёте.

`-l`, `--showlocals`

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

`--tb=style`

Параметр `--tb=style` изменяет способ вывода пакетов трассировки для сбоев. При сбое теста `pytest` отображает список сбоев и так называемую обратную трассировку, которая показывает точную строку, в которой произошёл сбой. Хотя `tracebacks` полезны большую часть времени, бывают случаи, когда они раздражают. Вот где опция `--tb=style` пригодится. Полезны стили `short`, `line` и `no`.

- `--tb=no` полностью удаляет трассировку.
- `--tb=line` во многих случаях достаточно, чтобы показать, что не так. Если у вас много неудачных тестов, этот параметр может помочь отобразить шаблон в сбоях. Он сохраняет ошибку в одной строке.
- `--tb=short` показывает сокращённый вариант трассировки, то есть печатает только строку `assert` и символ `E` без контекста.

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

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

`--durations=N`

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

Итоги

Итак, на этой лекции вы узнали, какие есть фреймворки для написания автотестов на Python и познакомились с фреймворком `Pytest`, научились писать и выполнять на нём простые тесты.

Вы узнали:

- Как использовать оператор `assert`.
- Какие есть фреймворки для тестирования на Python.
- Как использовать `Pytest`.
- Какие есть ключи и параметры `Pytest`.

Это базовые знания о Pytest, но их уже достаточно, чтобы писать функциональные тесты с использованием этого фреймворка. Этим мы и займёмся на следующем семинаре.

Что можно почитать ещё?

1. Здесь описаны практики эффективного тестирования с pytest [статья на habr](#).
2. 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. [Топ-6 тестовых фреймворков для Python](#).
3. [Тестирование устройств с помощью Robot](#).