Фреймворк pytest в Python, тестирование кода

Понимание тестирования при помощи pytest

Фреймворк pytest позволяет легко писать небольшие, удобочитаемые тесты и может масштабироваться для поддержки сложного функционального тестирования приложений и библиотек.

Для работы pytest требуется: Python 3.7+ или PyPy3.

Установка модуля pytest в виртуальное окружение.

Модуль pytest размещен на PyPI, поэтому установка относительно проста.

```
# создаем виртуальное окружение, если нет
$ python3 -m venv .venv --prompt VirtualEnv
# активируем виртуальное окружение
$ source .venv/bin/activate
# обновляем `pip`
(VirtualEnv):~$ python3 -m pip install -U pip
# ставим модуль `pytest`
(VirtualEnv):~$ python3 -m pip install -U pytest
```

Общие сведения о тестировании при помощи pytest

Тесты предназначены для того, чтобы посмотреть на результат поведения кода в определенной ситуации и убедиться, что результат соответствует тому, который ожидается. Поведение нельзя измерить эмпирически, поэтому написание тестов может быть сложной задачей.

Поведение - это то, как некоторая система действует в ответ на конкретную ситуацию и/или действия. Но как именно и почему что-то делается, не так важно, как то, что было сделано.

Можно думать о тесте как о четырех этапах:

- Подготовка.
- Действие.
- Утверждение.
- Очистка.

Подготовка.

ПОДГОТОВКА означает практически все, кроме *ДЕЙСТВИЯ*. Проще говоря - это подготовка окружения и объектов, которые требуются для проведения *ДЕЙСТВИЯ*. Это может быть все что угодно: подготовка к инициализации объектов, создание временных каталогов и файлов, запуск/остановку служб, ввод записей в базу данных или даже такие вещи, как определение URL-адреса для запроса, создание некоторых учетных данных для еще не существующего пользователя или просто ожидание завершения какого-либо процесса.

Для стадии подготовки к тесту фреймворк pytest использует так называемые фикстуры - это простые функции, определяемые тестировщиком. При помощи декоратора @pytest.fixture можно сообщить pytest, что конкретная функция является подготовительной. Фи pytest очень гибкий инструмент. Фикстуры могут принимать агрументы, для одного теста фикстур может быть несколько, одна фикстура может запрашивать другую фикстуру, одну фикстуру может использовать несколько тестов и т.д.

Действие.

ДЕЙСТВИЕ это то, что изменит состояние, которое запускает поведение, подлежащее тестированию. Именно это поведение осуществляет изменение тестируемой системы (SUT) по которому можно судить об устойчивости этой системы. Обычно *ДЕЙСТВИЕ* принимает форму вызова функции/метода с *ПОДГОТОВЛЕННЫМ* контекстом для моделирования той или иной ситуации.

Для моделирования различных состояний в момент тестирования фреймворк pytest использует помощник @pytest.mark. Например:

- Почти всегда, тестируемые методы или функции в приложении требуют передачу определенных аргументов. За передачу параметров тестируемым функциям/методам в pytest отвечает декоратор @pytest.mark.parametrize(). Передать параметры также можно с помощью фикстур.
- Возникают ситуации, когда требуется пропустить группу тестов. Например, для разных ОС необходимы различные группы тестов. С такими ситуациями помогут справиться декораторы @pytest.mark.skip() и @pytest.mark.skipif().
- При помощи помощника @pytest.mark можно пометить тесты, например дающие сбой, и запустить их отдельно.
- и т.д.

Утверждение.

УТВЕРЖДЕНИЕ - это место, где проверяется результирующее состояние тестируемой системы, то есть, что *соответствует* или *не соответствует* ожидаемому результату/поведению. *Утверждение* в тесте - это проведение сравнения полученного состояния во время *ДЕЙСТВИЯ* с неким, наперед известным/ожидаемым результатом, на основании которого можно судить об устойчивости системы. Например: если итоговая переменная result должна иметь список целых чисел в диапазоне от 0 до 10, то утверждение будет выглядеть следующим образом:

```
# длинные утверждения не дают сразу понять,
# что же здесь хотели проверить
assert (isinstance(result, list) and
        all([isinstance(n, int) for n in result]) and
        all([0 <= n <= 10 for n in result]))
```

Что бы не писать длинные утверждения, их можно разнести по разным тестовым функциям, которые в свою очередь можно объединить в группу при помощи класса или модуля. Группы тестов помогают использовать одни и те же значения переменных для тестируемой функции/метода. Это поможет избежать передачу параметров для каждого отдельного теста, при тестировании одной и той же функции.

Очистка.

ОЧИСТКА - это то, где тест сам по себе завершает работу, и что бы случайно не повлиять на итоги других тестов, производится очистка *ПОДГОТОВЛЕННОГО* контекста (если это требуется). Например, удаление тестовых пользователей/записей из баз данных, очистка временных файлов из рабочей директории и т.д.

Очистку можно производить как в самих тестовых функциях, так и в фикстурах.

> *По своей сути, тест - это этапы ДЕЙСТВИЯ и УТВЕРЖДЕНИЯ, а этап ПОДГОТОВКИ обеспечивает только контекст. Поведение существует между ДЕЙСТВИЕМ и УТВЕРЖДЕНИЕМ.*

Общий примеры использования фреймворка pytest.

Базовое использование фикстур @pytest.fixture.

На базовом уровне, тестовые функции запрашивают необходимые им фикстуры, объявляя их в качестве аргументов.

Когда pytest запускает тест, он просматривает аргументы в сигнатуре этой тестовой функции, а затем ищет фикстуры с теми же именами, что и эти аргументы. Как только pytest находит их, он запускает эти фикстуры, фиксирует то, что они вернули (если возвращают), и передает эти объекты в тестовую функцию в качестве аргументов.

```
##############################
###### ТЕСТИРУЕМЫЙ КОД ######
##############################
class Fruit:
    def __init__(self, name):
        self.name = name
        self.cubed = False

    def cube(self):
        self.cubed = True


class FruitSalad:
    def __init__(self, *fruit_bowl):
        self.fruit = fruit_bowl
```

```
        self._cube_fruit()

    def _cube_fruit(self):
        for fruit in self.fruit:
            fruit.cube()

############################
###### ТЕСТИРОВАНИЕ ######
############################
import pytest

# ПОДГОТОВКА
@pytest.fixture
def fruit_bowl():
    return [Fruit("apple"), Fruit("banana")]

def test_fruit_salad(fruit_bowl):
    # ДЕЙСТВИЕ
    fruit_salad = FruitSalad(*fruit_bowl)

    # УТВЕРЖДЕНИЕ
    assert all(fruit.cubed for fruit in fruit_salad.fruit)
```

В этом примере test_fruit_salad() "*запрашивает*" fruit_bowl() (т.е. def test_fruit_salad(fruit_bowl):), и когда pytest увидит это, он выполнит функцию-фикстуру fruit_bowl и передаст возвращаемый объект в test_fruit_salad в качестве аргумента fruit_bowl.

А так одна фикстура, может запрашивать/взаимодействовать с другой фикстурой:

```
##############################
###### ТЕСТИРУЕМЫЙ КОД ######
##############################
class Fruit:
    def __init__(self, name):
        self.name = name

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

############################
###### ТЕСТИРОВАНИЕ ######
############################
import pytest

@pytest.fixture
def my_fruit():
    return Fruit("apple")

@pytest.fixture
def fruit_basket(my_fruit):
    return [Fruit("banana"), my_fruit]

def test_my_fruit_in_basket(my_fruit, fruit_basket):
    assert my_fruit in fruit_basket
```

Передача параметров в тестируемый метод/функцию.

Фреймворк pytest позволяет параметризировать тест на нескольких уровнях:

- По средствам фикстуры @pytest.fixture().
- При помощи помощника @pytest.mark.

Разберем способ передачи параметров при помощи помощника @pytest.mark.parametrize(). Тест будем запускать из отдельной папки, предназначенной для тестов.

Для этого создадим необходимые папки и файлы:

```
# создаем необходимые папки
$ mkdir -p ./testing/test_folder
# создаем необходимые файлы (они будут пустые)
$ touch ./testing/util.py ./testing/test_folder/test_util.py
```

Вверх

Да[illegible]рываем файл ./testing/util.py и вставляем в него код ниже.

```python
# файл ./testing/util.py
def str_to_num(str):
    """Вспомогательная функция:
    преобразует строку в число"""
    if '.' in str and str.replace('.', '').isdigit():
        return float(str)
    elif str.isdigit():
        return int(str)

def str_to_int_list(str_lst):
    """Тестируемая функция:
    преобразует список строк в список целых чисел"""
    num_list = []
    for item in str_lst:
        n = str_to_num(item)
        if n is not None:
            if isinstance(n, float):
                n = round(n)
            if 0 <= n <= 10:
                num_list.append(n)
    return num_list
```

И наконец открываем файл ./test_folder/test_util.py и вставляем в него код с тестом, который расположен ниже.

Обратите внимание как в файле теста происходит импорт from util import str_to_int_list, хотя файл util.py находится на уровень выше... (подробнее об импорте при тестировании в материале "Интеграция тестов pytest с проектом.")

```python
# файл ./testing/test_folder/test_util.py
import pytest
# обратите внимание на импорт `util`
from util import str_to_int_list

@pytest.mark.parametrize("str_lst", [
    # определяем значения, которые будет
    # принимать переменная `str_lst``
    ['8.3', '11', 'девять', '1', '5', '3'],
    ['пять', '-1', '-13', '7', '3.9', '4'],
    ['5ять', '1,5', '6.3', '2,0', 'два', '9']
    ])
class Test_str_to_int_list():
    """Группа тестов, которая запускается с одними и теми же
    параметрами и проверяет функцию на разные утверждения"""

    # переменная `str_lst` передается из  помощника
    # `@pytest.mark` в объект класса неявно
    def test_is_list(self, str_lst):
        """Результат должен быть в виде списка"""
        result = str_to_int_list(str_lst)
        assert isinstance(result, list)

    def test_int_to_list(self, str_lst):
        """Список должен содержать только целые числа"""
        result = str_to_int_list(str_lst)
        assert all([isinstance(n, int) for n in result])

    def test_0_10(self, str_lst):
        """Числа в списке должны быть в диапазоне от 0 до 10"""
        result = str_to_int_list(str_lst)
        assert all([0 <= n <= 10 for n in result])

# моделирование исключительных ситуаций
@pytest.mark.parametrize("str_lst", [
    # передается пустой список
    [],
    # и список не содержащий цифры
    ['1.ин', 'два', 'три', '4етыре']
    ])
def test_empty_list(str_lst):
    """Тест на исключительные ситуации"""
    result = str_to_int_list(str_lst)
    assert result == []
```

Вверх

Запускаем тесты. Для этого активируем виртуальное окружение, где установлен pytest и затем перейдем в папку ./testing:

```
# активируем виртуальное окружение
$ source .venv/bin/activate
# перейдем в папку `./testing`
(VirtualEnv) :~$ cd testing
# запускаем тест
(VirtualEnv) :~/testing$ python3 -m pytest -v
```

Готово, тесты должны отработать.

Содержание раздела:

Вверх