Юнит тестирование

Что такое модульное тестирование

Среди фаз тестирования принято выделять:
- Модульное тестирование
- Интегральное тестирование
- Системное тестирование
- Приемное тестирование

Таким образом, модульное тестирование является первой фазой тестирования. Этот тип можно охарактеризовать, как низкое уровневое тестирование, сфокусированое на небольших участках кода. Такое тестирование пишется самими же программистами. Такие тесты проходят быстрее чем другие виды тестирования.

Интересный вопрос, что такое unit?
В объектно ориентированом подходе за unit принято принимать класс, в процедурном и функциональном программировании — отдельную функцию.

Frameworks для unit testing

unittest
unittest – это фрэймворк из стандартной библиотеки языка Python. Его архитектура выполнена в стиле xUnit. xUnit представляет собой семейство framework'ов для тестирования в разных языках программирования.

nose
nose - это расширение для unittest, которое позволяет облегчить написание тестов с помощью библиотеки unittest. Как раз девизом nose является фраза "nose extends unittest to make testing easier".

pytest
pytest довольно мощный инструмент для тестирования. При разработке на pytest просто пишете функции, которые должны начинаться с "test_" и используете assert'ы, встроенные в Python.

Простые примеры pytest

Используем стандартный assert

```
def func(x):
    return x + 1

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

Проверяем, что было поднято исключение

```
def f():
    raise SystemExit(1)
def test_mytest():
    with pytest.raises(SystemExit):
        f()
```

Групируем тесты логически

```
class TestCode:
    def test_one(self):
        assert True

    def test_two(self):
        pass
```

Используем fixture

```
@pytest.fixture
def conftest():
    print('do important work')
def test_one(conftest):
    assert True
```

Цель введения fixture это обеспечения необходимых предусловий для тестов, которое можно переиспользовать при запуске разных тестов.

Продвинутое использование pytest

Для fixture можно выбирать scope: function, class, module, session
Таким образом задается порядок выполнения fixture

```
@pytest.fixture(scope='function')
def func():
    print('function scope')

@pytest.fixture(scope='module')
def class_():
    print('module scope')

def test_one(func, module):
    assert False
```

Вместо

```
@pytest.fixture(scope='module')
def class_():
    print('module scope')
```

Можем использовать

```
@pytest.fixture(scope='module', autouse = True)
def class_():
    print('module scope')
```

Тогда наша fixture будет использоваться во всех тестах нашего файла

Можно добавить fixture в **conftest.py** во всех тестах нашего модуля

Продвинутое использование pytest

fixture может использовать fixture:

```
@pytest.fixture(scope='session')
def up_level_fixture():
    print('do up level work')

@pytest.fixture(scope='module')
def down_level_fixture(up_level_fixture):
    print('do down level work')

def test_fixtures(down_level_fixture):
    assert False
```

Можем пропускать тесты в некритичном случае или наоборот

```
@pytest.fixture(scope='module')
def down_level_fixture(up_level_fixture):
    try:
        print('do down level work')
        raise Exception()
    except:
        if os.getenv('CI', False):
            pytest.skip('can\'t do')
        else:
            pytest.fail('can\'t do')
```

Продвинутое использование pytest

fixture может можно параметризовать, тогда выполнятся все варианты тестов

```
@pytest.fixture(params=['kek', 'lol', 'wow'])
def action(request):
    return request.param

@pytest.fixture(params=['boy', 'girl'])
def man(request):
    return request.param

def test_man_action(man, action):
    print('{} do {}'.format(man, action))
    if man == 'girl' and action == 'wow':
        assert True
    else:
        assert False
```

Тесты можно маркировать

```
@pytest.mark.complicated
def test_complicated():
    print('complicated test')
    assert False
```

И вызывать с ключем -k complicated или вызывать в принципе нужные тесты с ключем -m

Можем их пропускать

```
@pytest.mark.complicated
@pytest.mark.skipif(is_really_compicated()
, reason='Test really compicated')
def test_complicated():
    print('complicated test')
    assert False
```

Еще больше примеров

Мы можем параметризовать и маркировать тесты, давайте совместим

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

У pytest есть стандартные маркеры pytest --markers, один из них **xfail —** полезно использовать, когда только разрабатываем тест

```
@pytest.fixture
def open_file():
    print('open file')
    print('callback to close file')
    return 'file'

def test_yield(open_file):
    print(open_file)
    assert True
```

```
@pytest.fixture
def open_file():
    print('open file')
    yield 'file'
    print('close file')

def test_yield(open_file):
    print(open_file)
    assert True
```

Для классов опять же

```
@pytest.mark.usefixtures("lol")
class TestCode:
    def test_one(self):
        assert True

    def test_two(self):
        pass
```

Управление маркерами

- Можно создавать свои 'хитрые' маркеры
- Можно создать регистер маркеров **pytest.ini**
- Можно конролировать использоание только своих маркеров (есть в документации, но у меня не вышло)

pytest.ini

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

Конец