Unit testing

Олег Ломака
oleg.lomaka@gmail.com
Prom.ua

unittest

unittest.TestCase

- Методы класса TestCase, начинающиеся на test_ считаются тестами и выполняются последовательно
- TestCase.setUp() — подготавливаем данные для теста
- TestCase.tearDown() — очищаем ресурсы после теста
- setUp/tearDown будет выполнено перед/после каждого вызова test_ метода.

```python
import unittest

class TestStringMethods(unittest.TestCase):

    def test_upper(self):
        self.assertEqual('foo'.upper(), 'FOO')

    def test_isupper(self):
        self.assertTrue('FOO'.isupper())
        self.assertFalse('Foo'.isupper())


if __name__ == '__main__':
    unittest.main()
```

Результаты работы теста

- AssertionError — test failed. Но не надо выбрасывать этот exception самому. Нужно использовать TestCase.fail(msg=None) *или* TestCase.assert*()
- SkipTest — skip this test
- Other Exception — test error

Варианты проверок

Method	Checks that
`assertEqual(a, b)`	`a == b`
`assertNotEqual(a, b)`	`a != b`
`assertTrue(x)`	`bool(x) is True`
`assertFalse(x)`	`bool(x) is False`
`assertIs(a, b)`	`a is b`
`assertIsNot(a, b)`	`a is not b`
`assertIsNone(x)`	`x is None`
`assertIsNotNone(x)`	`x is not None`
`assertIn(a, b)`	`a in b`
`assertNotIn(a, b)`	`a not in b`
`assertIsInstance(a, b)`	`isinstance(a, b)`
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`
`assertRaises(exc, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc*
`assertRaisesRegexp(exc, r, fun, *args, **kwds)`	`fun(*args, **kwds)` raises *exc* and the message matches regex *r*

Smoke testing

Наиболее примитивный вид теста. Можно проверить саму возможность вызова функции или загрузки модуля без проверки того, что результат несет какой-то смысл.

```
class CompanyControllerTestCase(TestCase):

    records = [AdminUser]

    def test_smoke_with_permission(self):
        login(AdminUser)
        company_list_url = url.admin.absolute(
            controller='company', action='list'
        )
        resp = app.get(company_list_url)
        self.assertEqual(resp.status_int, 200)
```

Unit testing

Процесс тестирования, при котором проверяется отдельный модуль или функция. Проверять можно как на корректность ожидаемого результата, так и на ожидаемые ошибки.

```python
class TestIsPaidPackage(TestCase):

    def test_package(self):
        for service_id in PremiumService.PAID_PACKAGES:
            self.assertTrue(
                is_paid_package(service_id),
                msg=service_id,
            )

    def test_non_paid_package(self):
        services = PremiumService.SERVICES + [
        PremiumService.PACKAGE_FREE,
        PremiumService.PACKAGE_TEST,
    ]
    for service_id in services:
        self.assertFalse(
            is_paid_package(service_id),
            msg=service_id,
        )
```

Integration testing

Тестирование работы нескольких модулей / компонентов системы в связке.

Например, при вызове некой функции, мы создаем фикстуры в базе. Проверяется работа алгоритма в связке с persistence слоем или другими сторонними сервисами.

```python
from uaprom.lib.testing.testcase import TestCase
from uaprom.lib.fixtures.base.agency import BaseAgencyRate
from uaprom.lib.fixtures.base.company import BaseActiveCompany


class AgencyCompany(Record):
    __model__ = Agency

    id = unique(rand.int, min=2 ** 30, max=2 ** 31 - 1)
    agency_rate = BaseAgencyRate
    company_name = rand.unicode(250)
    owner = BaseActiveCompany
```

```python
class GetAgencyNameHelperTestCase(TestCase):

    records = [AgencyCompany]

    def test_name_agency(self):
        agency = Agency.query.get(AgencyCompany.id)
        agency_name = h.get_agency_name(agency)
        self.assertEqual(agency.company_name, agency_name)
```

Regression testing

Написание тестов, на основе выявленных ошибок.
Проверка, что подобные ошибки не появятся в будущем.

Запуск тестов

- python -m unittest my.module1 my.module2.TestClass
- python -m unittest discover
- nose (http://nose.readthedocs.org/en/latest/)
- pytest (http://pytest.org/latest/)

nose

- Более гибко позволяет указывать где искать тесты
- Плагины (coverage, performance, pylons)

Вывод

Чтобы протестировать какой-либо код, нам нужно

- некая функция, которая вызовет этот код и проверит корректность результата (unittest.TestCase.test_*)
- метод, с помощью которого функция сможет сообщить о результатах проверки (unittest.TestCase.assert*())
- способ найти в нашем проекте все функции, которые выполняют тестирование, запустить их и показать нам сводный результат (uniitest.discover, nose, pytest)

mock

https://docs.python.org/3/library/unittest.mock.html
(from 3.3)

mock.sentinel

- Генератор уникальных объектов. Объекты создаются динамически во время доступа к атрибутам sentinel.

```
obj = sentinel.obj1
assert obj is sentinel.obj1
assert obj is not sentinel.obj2
```

mock.Mock

- mock.Mock(spec, return_value, side_effect, wraps)
- called / assert_called_with /assert_called_once_with
- called_count

```
from unittest import TestCase
from mock import Mock, sentinel
from my.module import get_company_id

class MyTest(TestCase):
    def test_get_company_id(self):
        company = Mock(id=sentinel.company_id)
        company_id = get_company_id(company)
        self.assertIs(company_id, sentinel.company_id)
```

mock.patch

```
from mock import patch

class MyTest(TestCase):
    @patch("my.module.method1")
    def test_one(self, method1):
        method1.return_value = True
        ...

    def test_two(self):
        with patch("my.module.method2") as method2:
            method2.return_value = True
            ...

    def test_tree(self):
        method3_patch = patch("my.module.method3")
        method3 = method3_patch.start()
        method3.return_value = True
        ...
        method3_patch.stop()
```

Еще о mock.patch

- patch.object
- patch.dict
- patch.multiple

Банальности.
Почему тестировать хорошо.

- Простые smoke тесты могут спасти от банальных ошибок, вызванных мержами или опечатками.
- Regression тесты часто могут быть написаны быстрее, чем получится воспроизвести ошибку через web-интерфейс. И время экономит, и позволяет не повторить эту ошибку в будущем.
- При написании теста функции, можно понять, что какие-то связи мешают тестированию. Это лишний повод задуматься над архитектурой, не слишком ли много берет на себя эта функция.

Тестирование часто улучшает качество кода

- Когда вы собираетесь тестировать свой код, чаще всего вы будете писать pure функции, которые поменьше зависят от внешнего мира и по возможности не делают side-эффектов.
- Походы во внешний мир могут выражаться в стандартизированных интерфейсах, которые проще заменить Mock'ами.

Еще о unittest.TestCase

- @classmethod
  setUpClass(cls):
- @classmethod
  tearDownClass(cls):
- setUpModule
- tearDownModule

Варианты указания теста

- uaprom.tests.lib.test_helpers — все тесткейсы из модуля test_helpers
- uaprom.tests.lib.test_helpers:TestHelpersTitle — все тесты из тесткейса TestHelpersTitle из модуля test_helpers
- uaprom.tests.lib.test_helpers:TestHelpersTitle.test_product_page_title_portal — тестировать только тест test_product_page_title_portal
- uaprom/tests/lib/test_helpers.py — все тесткейсы из файла

Links

- http://www.voidspace.org.uk/python/mock/ — библиотека позволяет создавать фиктивные объекты, которые могут себя вести по аналогии с трестируемыми, подменять куски кода, ассертить разные ситуации.
- https://nose.readthedocs.org/en/latest/ — nosetests. Ищет и запускает тесты в нашем проекте