Тестирование: Unittest | Я.Шпора

Выражение с ключевым словом `assert` позволяет в любом месте программы

- сделать предположение о выполнении какого-либо условия;
- проверить, выполняется ли это условие;
- в случае, если условие не выполнено — вернуть сообщение об ошибке.

```
x = 5

# Синтаксис: assert <утверждение>, 'Сообщение об ошибке'
# Делаем утверждение: "переменная x равна пяти".
assert x == 5, 'Ошибка: x не равен пяти! Надо что-то чинить!'
# Утверждение вернёт True, сообщение об ошибке не будет показано.

# Ещё одно утверждение:
assert x == 4, 'Ошибка: x не равен четырём! Надо что-то чинить!'
# Утверждение x == 4 вернёт False, и в этом случае в консоль будет выведе
# ...
# AssertionError: Ошибка: x не равен четырём! Надо что-то чинить!
```

Библиотека unittest

Unittest входит в стандартную библиотеку Python.

Именование файлов: названия файлов с тестами должны начинаться с префикса `test_`, например, *test_views.py* или *test_models.py*. Файлы с тестами обычно хранят в директории */tests*.

Порядок написания тестов обычно такой:

- создать тестирующий класс, унаследованный от `unittest.TestCase`. Таких классов можно создать сколько угодно;
- каждый отдельный тест — это метод тестирующего класса. В классе можно объединить несколько тестов. Разработчик сам придумывает имена тестам (методам); имена методов класса должны начинаться с префикса `test_`. В

теле метода разработчик описывает необходимые утверждения и сообщения об ошибках;

- в каждом тесте делается и проверяется предположение, но вместо инструкций `assert` применяются встроенные методы класса `unittest.TestCase`. Названия методов начинаются со слова *assert*; вторая часть названия указывает, какую проверку проводит метод — например, метод `assertEqual(x, y)` проверяет равенство значений `x` и `y`. Таких методов довольно много, общий принцип их работы можно сопоставить с применением инструкции `assert`.

Сравнение методов модуля unittest и выражений с ключевым словом `assert`:

```
x = 5
```

Методы в Unittest	Проверка через assert
`assertEqual(x, 5, 'Ошибка!')`	`assert x == 5, 'Ошибка!'`
`assertIsInstance(x, int, 'Ошибка!')`	`assert isinstance(x, int), 'Ошибка!'`
`assertGreater(x, 2, 'Ошибка!')`	`assert x > 2, 'Ошибка!'`

Методы класса `TestCase`:

Метод Unittest	Какое утверждение проверяется	Утверждаем, что…
`assertTrue(x)`	`bool(x) is True`	x — это True
`assertFalse(x)`	`bool(x) is False`	x — это False
`assertIs(a, b)`	`a is b`	a — тот же объект, что и b
`assertIsNot(a, b)`	`a is not b`	a — иной объект, чем b
`assertIsNone(x)`	`x is None`	x — это None
`assertIsNotNone(x)`	`x is not None`	x — это не None
`assertIn(a, b)`	`a in b`	a принадлежит коллекции b
`assertNotIn(a, b)`	`a not in b`	a не принадлежит коллекции b
`assertNotIsInstance(a, b)`	`not isinstance(a, b)`	a не относится к типу данных b

Запуск тестов в unittest

Для запуска тестов нужно выполнить файл, в котором хранятся тесты:

```
python tests.py
```

Можно запустить тесты, вызвав модуль unittest из директории, где сохранён файл с тестами:

```
python -m unittest
```

Можно выполнить часть тестов. Если в директории хранится несколько файлов с тестами:

```
tests/
├── test_one.py
├── test_two.py
└── test_three.py
```

…можно выполнить только часть из них; для этого в директории */tests* нужно выполнить одну из команд:

```
python -m unittest  # Запуск всех файлов с тестами (всех трёх).
python -m unittest test_one  # Запуск одного файла с тестами.
python -m unittest test_one test_two  # Запуск двух файлов с тестами.
python -m unittest test_one.TestClass  # Запуск отдельного класса с тестам
python -m unittest test_one.TestClass.test_method  # Запуск отдельного те
```

Тестирование может быть автоматически остановлено в случае, если хотя бы один тест упал: для этого тесты надо запустить с ключом `-f`.

```
python -m unittest -f
```

Подробный отчёт о результатах тестирования можно получить, запустив тесты с флагом `-v` (`--verbose`, «подробно»):

```
python tests.py -v
```

При запуске тестирования часть тестов можно проигнорировать. Для этого в библиотеке есть специальные декораторы:

- **@unittest.skip(reason)** — пропустить тест. В параметре `reason` можно описать причину пропуска.
- **@unittest.skipIf(condition, reason)** — пропустить тест, если условие `condition` истинно.
- **@unittest.skipUnless(condition, reason)** — пропустить тест, если условие `condition` ложно.

При необходимости некоторые тесты можно обозначить как ожидаемо провалившиеся.

- **@unittest.expectedFailure** — ставит на тесте отметку «ожидаемое падение»; провалившиеся тесты, обёрнутые этим декоратором, будут обозначены строкой

  `expected failure`

Если ожидается, что тестируемая функция выбросит исключение — при тестировании применяют метод `assertRaises`, его описывают с использованием контекстного менеджера `with`.

```
import unittest


def division_func(a, b):
    """Функция деления одного числа на другое."""
    return a / b


class TestExample(unittest.TestCase):

    ...
    def test_zero_division(self):
        # Используем метод assertRaises как контекстный менеджер
        # (записываем его со словом with); указываем ожидаемый тип исключе
        # "ошибка деления на ноль".
        with self.assertRaises(ZeroDivisionError, msg="Ожидалась ошибка де
            # Передаём в функцию division_func() аргументы 1 и 0. На ноль
```

```
            # поэтому должна быть вызвана ошибка ZeroDivisionError.
            division_func(1, 0)
```

Метод subTest, параметризация тестов

Для выполнения нескольких одинаковых тестов с различными параметрами принято использовать метод `subTest()`.

```
from unittest import TestCase


def get_square(num):
    """Возвращает квадрат полученного аргумента"""
    return num ** 2


class TestExample(TestCase):

    def test_square(self):
        """Тест возведения в квадрат."""
        # Проверим три утверждения: при возведении первого числа в квадра
        # функция вернёт второе число.
        # Исходные данные соберём в кортеж, содержащий в себе другие корте
        values_results = (
            (2, 4),   # С этими параметрами тест вернёт OK.
            (3, 10),  # С этими параметрами тест провалится.
            (4, 20),  # И с этими параметрами - тоже провалится.
        )
        # Цикл, в котором кортежи, вложенные в values_results,
        # распаковываются в переменные value и expected_result:
        for value, expected_result in values_results:
            # Метод subTest в качестве контекстного менеджера.
            with self.subTest():
                result = get_square(value)
                # Тестовое утверждение, которое будет вызвано несколько ра
                # с разными значениями переменных.
                self.assertEqual(result, expected_result)
```

Фикстуры

Метод `setUp()` автоматически вызывается перед запуском каждого теста в классе и применяется для подготовки условий тестов.

Метод `tearDown` вызывается после каждого теста и применяется для очистки или для завершающих действий по окончании тестирования.

```
# test_calculator.py
import unittest

from calc_code.calculator import MadCalculator


class TestCalc(unittest.TestCase):
    """Тестируем MadCalculator."""

    def setUp(self):
        """Подготовка прогона теста. Вызывается перед каждым тестом."""
        # Подготавливаем данные для каждого теста.
        self.calc = MadCalculator()

        def tearDown(self):
                ...
```

Метод `setUpClass` выполняется однократно, перед запуском тестов класса.

Метод `tearDownClass` вызывается однократно после выполнения всех тестов класса.

Для работы этих методов необходим декоратор `@classmethod`.

```
# test_calculator.py
import unittest

from calc_code.calculator import MadCalculator


class TestCalc(unittest.TestCase):
    """Тестируем MadCalculator."""
```

```
    @classmethod  # Декорируем метод класса.
    def setUpClass(cls):
        """Вызывается один раз перед запуском всех тестов класса."""
        # Для создания объекта и обращения к нему вместо self применяем cl
        cls.calc = MadCalculator()
        print(cls.calc)  # Обращаемся к объекту не self.calc, а cls.calc.

    ...
        @classmethod
      def tearDownClass(cls):
                ...
```

Unittest в Django

Модуль для тестирования в Django работает по тому же принципу, что и библиотека unittest:

- разработчик создаёт классы, унаследованные от базового класса **TestCase**;
- в этих классах описываются методы, названия которых должны начинаться с префикса `test_`;
- каждый такой метод — это отдельный тест.

Различные варианты запуска тестов Django:

```
# Запустить все тесты проекта.
python manage.py test

# Запустить только тесты в приложении news.
python manage.py test news

# Запустить только тесты из файла tests/test_trial.py в приложении news.
python manage.py test news.tests.test_trial

# Запустить только тесты из класса Test
# в файле test_trial.py приложения news.
python manage.py test news.tests.test_trial.Test

# Запустить только тест test_example_fails
```

```
# из класса YetAnotherTest в файле tests/test_trial.py приложения news.
python manage.py test news.tests.test_trial.YetAnotherTest.test_example_f
```

Развёрнутый отчёт о результатах теста можно получить, выполнив команду `python manage.py test` с параметром `--verbosity` (или `-v`); значениями этого параметра могут быть числа от 0 до 3: чем больше значение — тем подробнее отчёт.

```
python manage.py test -v 2
```

В Django Unitests есть особенность использования методов `setUp`, `setUpClass`, `tearDown`, `tearDownClass`: необходимо явно вызывать одноименный родительский метод через `super()`

```
from django.test import TestCase


class MyTestCase(TestCase):

    @classmethod
    def setUpClass(cls):
        super().setUpClass()  # Вызов родительского метода.
        ...

    @classmethod
    def tearDownClass(cls):
        ...  # Выполнение необходимых операций.
        super().tearDownClass()  # Вызов родительского метода.
```

Для создания тестовых объектов в классе `django.test.TestCase` есть метод, более удобный, чем `setUp` и `setUpClass` — это метод `setUpTestData()`. Он похож на метод `setUpClass()`, но не требует явного вызова родительского метода.

При тестировании Django через модуль `django.test` рекомендуется работать именно с методом `setUpTestData()`.

```
from django.test import TestCase
```

```
class MyTestCase(TestCase):
    @classmethod
    def setUpTestData(cls):
        ...  # Подготовка тестовых данных для каждого теста.
```

Веб-клиент и пользователи в Django unittest

В каждом тестирующем классе по умолчанию создаётся объект веб-клиента; доступ к нему можно получить через атрибут `self.client`.

Применение:

```
response = self.client.get('/')
```

Подготовка к тестированию запросов от аутентифицированного пользователя:

```
class TestNews(TestCase):

    @classmethod
    def setUpTestData(cls):
        # Создаём пользователя.
        cls.user = User.objects.create(username='testUser')
        # Создаём объект клиента.
        cls.user_client = Client()
        # "Логинимся" в клиенте при помощи метода force_login().
        cls.user_client.force_login(cls.user)
        # Теперь через этот клиент можно отправлять запросы
        # от имени пользователя с логином "testUser".
```

В ответ на любой запрос, отправленный через клиент, возвращается специальный объект класса Response. В нём содержится ответ сервера и дополнительная информация. При тестировании могут быть полезны атрибуты этого объекта:

- `response.status_code` — содержит код ответа запрошенного адреса;
- `response.content` — данные ответа в виде строки байтов;

- `response.context` — словарь переменных, переданный для отрисовки шаблона при вызове функции `render()`;
- `response.templates` — перечень шаблонов, вызванных для отрисовки запрошенной страницы;

URL в тестах

В тестах не следует «хардкодить» — писать адрес страницы в явном виде, например, строкой `'/news/1/'`.

Лучше использовать функцию `reverse()` — она вернёт URL, с которым связано имя `namespace:name`.

```
url = reverse('news:detail', args=(self.news.pk,))
# Или
url = reverse('news:detail', kwargs={'pk': self.news.pk})
```

Практикум