



Тестирование ПО

что такое, виды тестов, зачем они
девелоперу, тесты без использования ПО



Тесты

unittest

pytest

Линтеры

Tox

Что такое тестирование


Тестирование программного обеспечения — процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определённым образом.

Тестирование – это процесс, при котором выявляется большинство существующих ошибок (не все).



И3 linkedin



Татьяна Иванченко • 1-й
Менеджер по ИТ – ООО Софтио
3 нед. • 


...

Мы ищем в нашу команду Python разработчика, который понимает зачем под каждую функцию в проекте нужно писать тесты и не пренебрегает этим. З.П. по результатам собеседования.
Опыт коммерческой разработки: не менее 1 – 3 года.
Уровень: Middle.
Занятость: полный день.
Требования:

- Знания и опыт разработки на языке Python (3.x);
- Хорошие знания Django, Django rest framework, docker, docker-compose;
- Опыт работы с GNU/Linux;
- Хорошее понимание клиент-серверной архитектуры, технологий и структур Web-разработки;
- Хорошее понимание баз данных (Postgres, Mysql);
- Опыт ООП, MVC и шаблоны проектирования, RESTFull API, OAuth2.0;
- Написание Unit, API тестов.

Будет плюсом:

- опыт работы с одним или несколькими фреймворками: Bootstrap, Angular, Vue.js, React.js;
- опыт работы с IoT;
- опыт работы с машинным обучением;
- Английский на уровне Intermediate+

 6

 Нравится  Комментировать  Поделиться  Отправить

Будьте первым(ой), кто прокомментировал это



Тесты

unittest

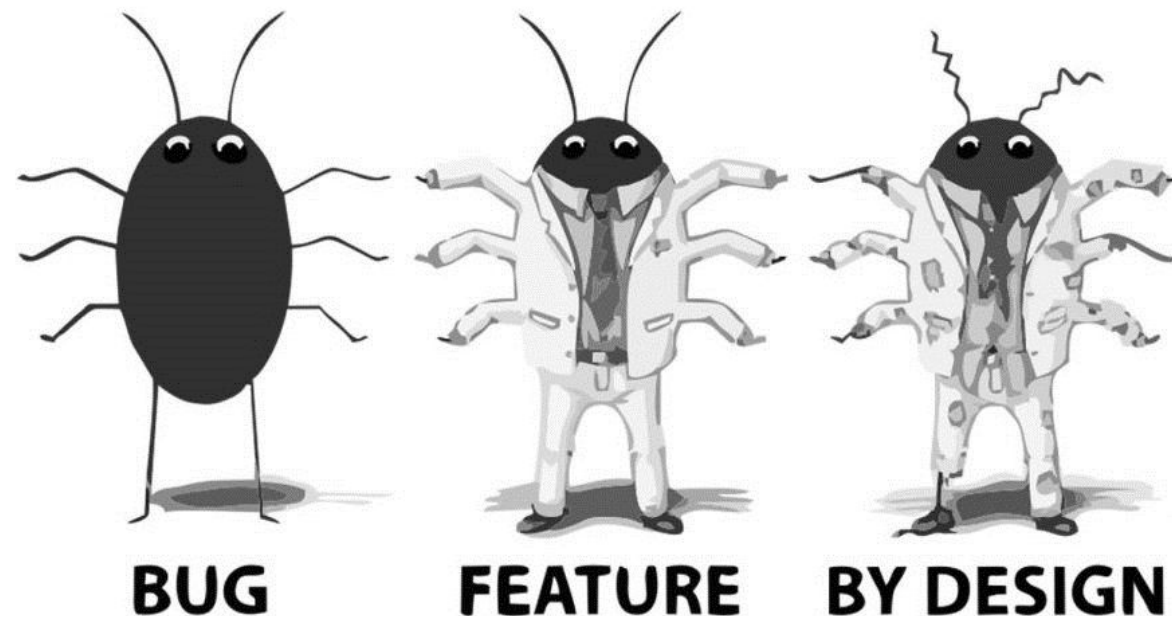
pytest

Линтеры

Tox

Баг

Баг (Bug) – ошибка в программе или в системе, из-за которой программа выдает неожиданное поведение и, как следствие, результат. Может стать результатом некорректной работы или отказа всей системы.



Классификация тестов

coggle

made for free at coggle.it



Тесты

unittest

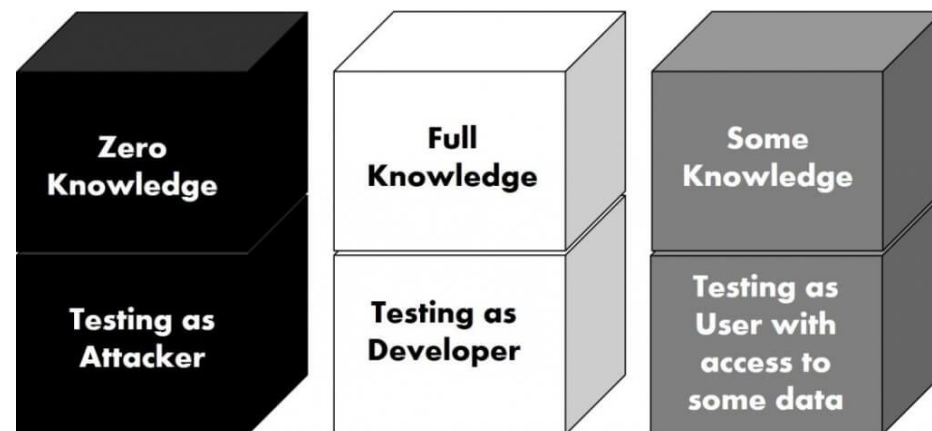
pytest

Линтеры

Tox

Методы тестирования

- **Тестирование черного ящика** (англ. black box testing) – метод тестирования ПО, при котором вся функциональность продукта исследуется без анализа исходного кода. Тестировщики создают логически понятные тест-кейсы, опираясь исключительно на требования из спецификации на проекте
- **Тестирование белого ящика** (англ. white box testing) – метод тестирования ПО, который подразумевает, что внутренняя структура и технические особенности ПО досконально известны проверяющему
- **Тестирование серого ящика** (англ. grey box testing) – специальный метод тестирования программного обеспечения с неполным знанием его внутреннего устройства. Чтобы выполнить подобный вид тестов, не нужно иметь доступ к исходному коду ПО.



Какие тесты пишет девелопер (наверное...)

- **Модульные (unit) тесты** – применяется для тестирования одной логически выделенной и изолированной единицы системы. Чаще всего это метод класса (хотя может и весь класс) или простая функция
- **Функциональные тесты** – тесты, которые призван полностью эмулировать поведение конечного пользователя системы. Проверяют связку модулей системы
- **Интеграционные тесты** – проверка взаимодействия разных систем по принципу “сервис-клиент”. Позволяют проверить, могут ли два наших модуля работать вместе



Плюсы и минусы тестирования ПО

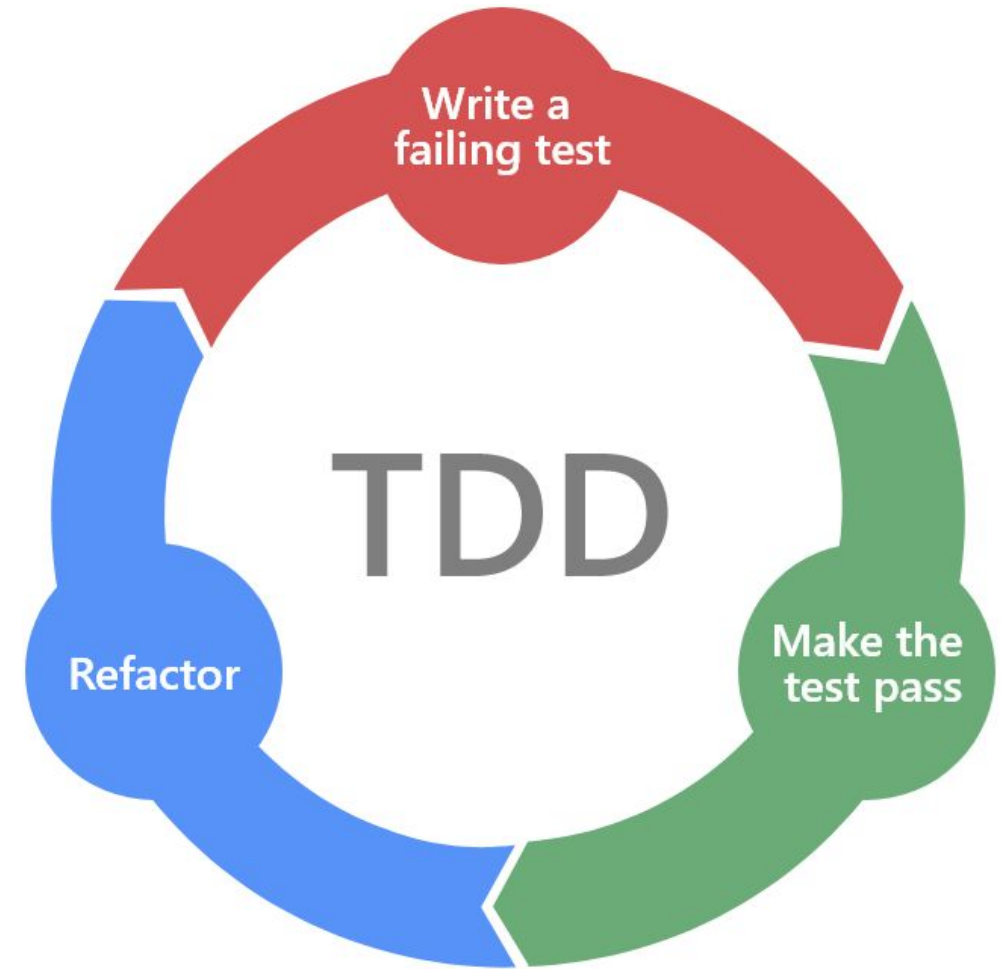
- ✓ Уверенность в функционале, меньше шансов сломать в будущем
- ✓ Проще обновлять зависимости
- ✓ Спокойно спим по ночам, продакшн работает (наверное...)
- ✓ Формирование требований к еще не написанному функционалу (TDD)
- ✓ Заккрытие дыр в функционале, безопасности
- ✓ Не надо проверять ранее написанный функционал при внедрении нового (экономия времени)
- ✗ Тратим время. Более долгий выкат в продакшн, проект дороже (время-деньги)
- ✗ Могут быть написаны бездумно, лишь бы достичь 100% code coverage
- ✗ Человеческий фактор: ошибка все равно может быть пропущена
- ✗ **ОЧЕНЬ СКУЧНО, НО НАДО**
- ✗ Как правило, если код написан плохо, то тестировать его очень трудно



TDD

Разработка через тестирование

(англ. test-driven development, TDD) — техника разработки программного обеспечения, которая основывается на повторении очень коротких циклов разработки: сначала пишется тест, покрывающий желаемое изменение, затем пишется код, который позволит пройти тест, и под конец проводится рефакторинг нового кода к соответствующим стандартам.



Тестирование без стороннего ПО

Рассмотрим прострой пример: у нас есть некоторый модуль, который содержит простые функции и мы хотим его протестировать.

```
1  def add(a, b):  
2      return a + b  
3  
4  def sub(a, b):  
5      return a - b  
6  
7  def mul(a, b):  
8      return a * b  
9  
10 def div(a, b):  
11     return a / b
```



Тестирование без стороннего ПО – функции

Первый способ, которым мы можем это сделать: написать другой модуль, в котором будут описаны тестовые функции, которые будут вызывать функционал из тестируемого модуля и сравнивать результат с ожидаемым.

```
1 import calc
2
3 def test_add():
4     if calc.add(1, 2) == 3:
5         print("Test add(a, b) is OK")
6     else:
7         print("Test add(a, b) is Fail")
8
9 def test_sub():
10    if calc.sub(4, 2) == 2:
11        print("Test sub(a, b) is OK")
12    else:
13        print("Test sub(a, b) is Fail")
14
15 def test_mul():
16    if calc.mul(2, 5) == 10:
17        print("Test mul(a, b) is OK")
18    else:
19        print("Test mul(a, b) is Fail")
20
21 def test_div():
22    if calc.div(8, 4) == 2:
23        print("Test div(a, b) is OK")
24    else:
25        print("Test div(a, b) is Fail")
26
27 test_add()
28 test_sub()
29 test_mul()
30 test_div()
```



Плюсы и минусы такого тестирования

- ✓ Просто, если проект очень мал
- ✓ Нет внешних зависимостей
- ✗ Если есть side effect, то будет сложно
- ✗ Приходиться писать очень много кода
- ✗ Очень слабые возможности
- ✗ Приходится думать над удобством интерфейса для дальнейшего использования



Тестирование без стороннего ПО – assert

Второй способ: написать тестовые функции, внутри которых сравнивать тестируемый функционал с ожидаемым значением с помощью `assert` (или жестко забить `assert` в коде, что я делать не рекомендую)

- **`assert`** – проверяет условие на истинность. Если не истинно, то вызывается исключение `AssertionError`

```
1 import calc
2
3 def test_add():
4     assert calc.add(1, 2) == 3
5
6 def test_sub():
7     assert calc.sub(4, 2) == 2
8
9 def test_mul():
10    assert calc.mul(2, 5) == 10
11
12 def test_div():
13    assert calc.div(8, 4) == 2
14
15
16 test_add()
17 test_sub()
18 test_mul()
19 test_div()
```



Плюсы и минусы такого тестирования

- ✓ Просто, если проект очень мал
- ✓ Нет внешних зависимостей
- ✓ Если все же жестко забили assert в коде, то можно отключить его поведение с помощью флага -O при выполнении файла (python -O file.py)
- ✓ Код стал короче
- ✗ Если есть side effect, то будет сложно
- ✗ Очень слабые возможности
- ✗ Приходится думать над удобством интерфейса для дальнейшего использования





Ваши вопросы

что необходимо прояснить в рамках
данного раздела



Тесты

unittest

pytest

Линтеры

Tox



Модуль unittest

тестирование кода с помощью встроенного модуля unittest



Тесты

unittest

pytest

Линтеры

Tox

Unittest

unittest – модуль, который входит в стандартную библиотеку Python, который позволяет разрабатывать автономные тесты, собирать тесты в коллекции, обеспечивает независимость тестов от framework'а отчетов и т.д..

unittest был перенесен из java jUnit, поэтому он не соответствует PEP8 в плане

именования своих структур данных.

```
1 import unittest
2
3 class TestStringMethods(unittest.TestCase):
4
5     def test_upper(self):
6         self.assertEqual('foo'.upper(), 'FOO')
7
8     def test_isupper(self):
9         self.assertTrue('FOO'.isupper())
10        self.assertFalse('foo'.isupper())
11
12    def test_split(self):
13        s = 'hello world'
14        self.assertEqual(s.split(), ['hello', 'world'])
15        # check that s.split fails when the separator is not a string
16        with self.assertRaises(TypeError):
17            s.split(2)
18
19 if __name__ == '__main__':
20     unittest.main()
```

Основные элементы

Test fixture – обеспечивает подготовку окружения для выполнения тестов, а также организацию мероприятий по их корректному завершению (например очистка ресурсов). Подготовка окружения может включать в себя создание баз данных, запуск необходим серверов и т.п.

```
1 def setUpModule():
2     createConnection()
3
4 def tearDownModule():
5     closeConnection()
```

```
1 import unittest
2
3 class Test(unittest.TestCase):
4     @classmethod
5     def setUpClass(cls):
6         cls._connection = createExpensiveConnectionObject()
7
8     @classmethod
9     def tearDownClass(cls):
10        cls._connection.destroy()
```



Основные элементы

Test case – это элементарная единица тестирования, в рамках которой проверяется работа компонента тестируемой программы (метод, класс, поведение и т.п.). Для реализации этой сущности используется класс `TestCase`.

```
1 import unittest
2 import calc
3
4 class CalcTest(unittest.TestCase):
5     def test_add(self):
6         self.assertEqual(calc.add(1, 2), 3)
7
8     def test_sub(self):
9         self.assertEqual(calc.sub(4, 2), 2)
10
11    def test_mul(self):
12        self.assertEqual(calc.mul(2, 5), 10)
13
14    def test_div(self):
15        self.assertEqual(calc.div(8, 4), 2)
16
17 if __name__ == '__main__':
18     unittest.main()
```



Основные элементы

Test suite – это коллекция тестов, которая может в себя включать как отдельные test case'ы так и целые коллекции (т.е. можно создавать коллекции коллекций). Коллекции используются с целью объединения тестов для совместного запуска.

```
1 import unittest
2 import calc_tests
3
4 calcTestSuite = unittest.TestSuite()
5 calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcBasicTests))
6 calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcExTests))
7 print("count of tests: " + str(calcTestSuite.countTestCases()) + "\n")
8
9 runner = unittest.TextTestRunner(verbosity=2)
10 runner.run(calcTestSuite)
```



Группировка тестов

С помощью класса `TestSuite` и метода `makeSuite` можно группировать различные test cases. Методы класса:

- `addTest(test)` – добавляет текст в коллекцию
- `addTests(tests)` – добавляет все тесты из итерируемого объекта
- `run(result)` – запускает тесты из коллекции и записывает результат в объект `result` (`unittest.TestResult`)

```
1 import unittest
2 import calc_tests
3
4 calcTestSuite = unittest.TestSuite()
5 calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcBasicTests))
6 calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcExTests))
7 print("count of tests: " + str(calcTestSuite.countTestCases()) + "\n")
8
9 runner = unittest.TextTestRunner(verbosity=2)
10 runner.run(calcTestSuite)
```



Основные элементы

Test runner – это компонент, который оркестрирует (координирует взаимодействие) запуск тестов и предоставляет пользователю результат их выполнения. Test runner может иметь графический интерфейс, текстовый интерфейс или возвращать какое-то заранее заданное значение, которое будет описывать результат прохождения тестов.

```
1 import unittest
2 import calc_tests
3
4 calcTestSuite = unittest.TestSuite()
5 calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcBasicTests))
6 calcTestSuite.addTest(unittest.makeSuite(calc_tests.CalcExTests))
7 print("count of tests: " + str(calcTestSuite.countTestCases()) + "\n")
8
9 runner = unittest.TextTestRunner(verbosity=2)
10 runner.run(calcTestSuite)
```



Как именовать что-то с тестами

- **Методы** класса следует именовать в стиле **snake_case_with_underscores**
- **Классы** следует именовать в стиле **CamelCase**, имя должно начинаться с Test
- **Модули** следует именовать в стиле **snake_case_with_underscores**
- Файлы тестов, как правило, размещаются в папке «tests»

```
1 import unittest
2
3 class WidgetTestCase(unittest.TestCase):
4     def setUp(self):
5         self.widget = Widget('The widget')
6
7     def test_default_widget_size(self):
8         self.assertEqual(self.widget.size(), (50,50),
9                             'incorrect default size')
10
11     def test_widget_resize(self):
12         self.widget.resize(100,150)
13         self.assertEqual(self.widget.size(), (100,150),
14                             'wrong size after resize')
```



Цикл работы с unittest

1. Разрабатываем отдельные тесты в рамках test case'ов
2. Собираем test case'ы в модули
3. Если нужно объединить несколько test case'ов, для их совместного запуска, они помещаются в test suite'ы, которые помимо test case'ов могут содержать другие test suite'ы
4. Запускаем тесты



Рекомендации по написанию тестов

При написании тестов следует исходить из следующих принципов:

- ✓ Работа теста не должна зависеть от результатов работы других тестов.
- ✓ Тест должен использовать данные, специально для него подготовленные, и никакие другие.
- ✓ Тест не должен требовать ввода от пользователя
- ✓ Тесты не должны перекрывать друг друга (не надо писать одинаковые тесты 20 раз). Можно писать частично перекрывающиеся тесты.
- ✓ Нашел баг -> напиши тест
- ✓ Тесты надо поддерживать в рабочем состоянии
- ✓ Модульные тесты не должны проверять производительность сущности (класса, функции)
- ✓ Тесты должны проверять не только то, что сущность работает корректно на корректных данных, но и то что ведет себя адекватно при некорректных данных.
- ✓ Тесты надо запускать регулярно



Методы TestCase

- **setUp** – подготовка прогона теста; вызывается перед каждым тестом.
- **tearDown** – вызывается после того, как тест был запущен и результат записан. Метод запускается даже в случае исключения (exception) в теле теста.
- **setUpClass** – метод вызывается перед запуском всех тестов класса.
- **tearDownClass** – вызывается после прогона всех тестов класса.
- **setUpModule** – вызывается перед запуском всех классов модуля.
- **tearDownModule** – вызывается после прогона всех тестов модуля.

```
1 import unittest
2
3
4 class TestUM(unittest.TestCase):
5     def setUp(self):
6         pass
7
8     def tearDown(self):
9         pass
10
11     def test_numbers_3_4(self):
12         self.assertEqual(3 * 4, 12)
13
14     def test_strings_a_3(self):
15         self.assertEqual('a' * 3, 'aaa')
16
17
18 if __name__ == '__main__':
19     unittest.main()
```



Плюсы и минусы unittest

- ✓ Встроен в Python
- ✓ Тем, кто работал с unit тестами в других языках программирования, будет просто включиться
- ✗ Не pythonic way оформление
- ✗ Не такой объемный функционал, как хотелось бы



setUpModule и tearDownModule

- Для организации работы с контекстом тестов в рамках модуля, setUpModule и tearDownModule должны быть оформлены как функции модуля
- Если возникнет исключение в setUpModule, то ни один тест из модуля не будет запущен
- В addModuleCleanup пишется код, который отвечает за очистку контекста

```
1 def setUpModule():  
2     createConnection()  
3  
4 def tearDownModule():  
5     closeConnection()
```



Assert'ы проверок в unittest

Метод	Что делает	Появился в
<code>assertEqual(a, b)</code>	<code>a == b</code>	
<code>assertNotEqual(a, b)</code>	<code>a != b</code>	
<code>assertTrue(x)</code>	<code>bool(x) is True</code>	
<code>assertFalse(x)</code>	<code>bool(x) is False</code>	
<code>assertIs(a, b)</code>	<code>a is b</code>	3.1
<code>assertIsNot(a, b)</code>	<code>a is not b</code>	3.1
<code>assertIsNone(x)</code>	<code>x is None</code>	3.1
<code>assertIsNotNone(x)</code>	<code>x is not None</code>	3.1
<code>assertIn(a, b)</code>	<code>a in b</code>	3.1
<code>assertNotIn(a, b)</code>	<code>a not in b</code>	3.1
<code>assertIsInstance(a, b)</code>	<code>isinstance(a, b)</code>	3.2
<code>assertNotIsInstance(a, b)</code>	<code>not isinstance(a, b)</code>	3.2



Assert'ы исключений и warning'ов в unittest

Метод	Что делает	Появился в
<code>assertRaises(exc, fun, *args, **kwargs)</code>	Функция <code>fun(*args, **kwargs)</code> вызывает исключение <code>exc</code>	
<code>assertRaisesRegex(exc, r, fun, *args, **kwargs)</code>	Функция <code>fun(*args, **kwargs)</code> вызывает исключение <code>exc</code> , сообщение которого совпадает с регулярным выражением <code>r</code>	3.1
<code>assertWarns(warn, fun, *args, **kwargs)</code>	Функция <code>fun(*args, **kwargs)</code> выдает сообщение <code>warn</code>	3.2
<code>assertWarnsRegex(warn, r, fun, *args, **kwargs)</code>	Функция <code>fun(*args, **kwargs)</code> выдает сообщение <code>warn</code> и оно совпадает с регулярным выражением <code>r</code>	3.2
<code>assertLogs(logger, level)</code>	Внутри <code>with</code> логируется запись <code>logger</code> 'ом с минимальным уровнем <code>level</code>	3.4



Assert'ы проверки различных ситуаций в unittest

Метод	Что делает	Появился в
<code>assertAlmostEqual(a, b)</code>	<code>round(a-b, 7) == 0</code>	
<code>assertNotAlmostEqual(a, b)</code>	<code>round(a-b, 7) != 0</code>	
<code>assertGreater(a, b)</code>	<code>a > b</code>	3.1
<code>assertGreaterEqual(a, b)</code>	<code>a >= b</code>	3.1
<code>assertLess(a, b)</code>	<code>a < b</code>	3.1
<code>assertLessEqual(a, b)</code>	<code>a <= b</code>	3.1
<code>assertRegex(s, r)</code>	<code>r.search(s)</code>	3.1
<code>assertNotRegex(s, r)</code>	<code>not r.search(s)</code>	3.2
<code>assertCountEqual(a, b)</code>	а и b имеют одинаковые элементы (порядок неважен)	3.2



Assert'ы проверки различных ситуаций в unittest

Типо-зависимые assert'ы, которые используются при вызове `assertEqual()`.
Приводятся на тот случай, если необходимо использовать конкретный метод.

Метод	Что делает	Появился в
<code>assertMultiLineEqual(a, b)</code>	strings	3.1
<code>assertSequenceEqual(a, b)</code>	sequences	3.1
<code>assertListEqual(a, b)</code>	lists	3.1
<code>assertTupleEqual(a, b)</code>	tuples	3.1
<code>assertSetEqual(a, b)</code>	sets or frozensets	3.1
<code>assertDictEqual(a, b)</code>	dicts	3.1



Пропуск тестов и классов

- **@unittest.skip(reason)** – пропустить тест. reason описывает причину пропуска
- **@unittest.skipIf(condition, reason)** – пропустить тест, если condition ИСТИННО
- **@unittest.skipUnless(condition, reason)** – пропустить тест, если condition ЛОЖНО
- **@unittest.expectedFailure** – пометить тест как ожидаемая ошибка

Для пропущенных тестов не запускаются setUp() и tearDown(). Для пропущенных классов не запускаются setUpClass() и tearDownClass(). Для пропущенных модулей не запускаются setUpModule() и tearDownModule().

```
1 class MyTestCase(unittest.TestCase):
2
3     @unittest.skip("demonstrating skipping")
4     def test_nothing(self):
5         self.fail("shouldn't happen")
6
7     @unittest.skipIf(mylib.__version__ < (1, 3),
8                     "not supported in this library version")
9     def test_format(self):
10         # Tests that work for only a certain version of the library.
11         pass
12
13     @unittest.skipUnless(sys.platform.startswith("win"), "requires Windows")
14     def test_windows_support(self):
15         # windows specific testing code
16         pass
17
18     def test_maybe_skipped(self):
19         if not external_resource_available():
20             self.skipTest("external resource not available")
21         # test code that depends on the external resource
22         pass
23
24
25 @unittest.skip("showing class skipping")
26 class MySkippedTestCase(unittest.TestCase):
27     def test_not_run(self):
28         pass
```



Command line interface (CLI)

Для запуска определенных модулей тестов:

- `python -m unittest test_module1 test_module2`

Запуск конкретного test case:

- `python -m unittest test_module.TestClass`

Запуск метода из некоторого test case:

- `python -m unittest test_module.TestClass.test_method`

Запуск тестов из некоторого модуля:

- `python -m unittest tests/test_something.py`

Запуск тестов из директории tests:

- `python -m unittest discover -s tests`

Запуск всех тестов из директории tests, которые подходят под `*_test.py`:

- `python -m unittest discover -s tests -p “*_test.py”`

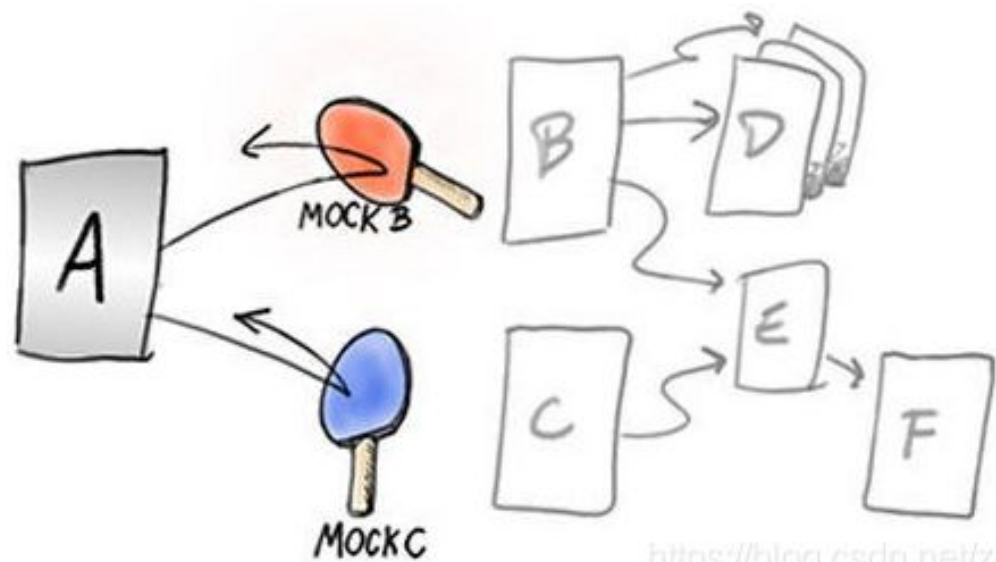


Mock объект

У нас есть модуль A, мы хотим его протестировать и есть другие модули, которые мы тестировать не хотим.

Первый ходит в сеть, другой в базу данных. В таких случаях нам поможет mock (конечно, если мы не пишем интеграционный тест, а то в таком случае мы поднимем тестовую базу данных, напишем тестового клиента).

Mock — это набор объектов, которыми можно подменить настоящий объект. На любое обращение к методам, к атрибутам он возвращает тоже mock.





Ваши вопросы

что необходимо прояснить в рамках
данного раздела



Тесты

unittest

pytest

Линтеры

Tox



Пакет pytest

тестирование кода с помощью мощного
стороннего пакета



Тесты

unittest

pytest

Линтеры

Tox

Pytest

Pytest – сторонний фреймворк, который позволяет разрабатывать автономные тесты, собирать отчеты, рассчитывать покрытие кода тестами, проводить проверку на соответствие стандартам и т.д.

Для установки pytest нужно
выполнить:

- `pip install pytest`

```
(venv33) c:\venv33\Scripts>pytest -v c:\BOOK\bopytest-code\code\ch1\test_one.py
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0 -- c:\venv
33\scripts\python.exe
cachedir: ..\..\cache
rootdir: c:\, inifile:
collecting 0 items
collecting 1 item
collected 1 item

..\..\BOOK\bopytest-code\code\ch1\test_one.py::test_passing PASSED

===== 1 passed in 0.04 seconds =====
(venv33) c:\venv33\Scripts>
```

```
(venv33) c:\BOOK\bopytest-code\code\ch1>pytest
===== test session starts =====
platform win32 -- Python 3.3.5, pytest-3.2.5, py-1.4.34, pluggy-0.4.0
rootdir: c:\BOOK\bopytest-code\code\ch1, inifile:
collecting 0 items
collecting 1 item
collecting 2 items
collecting 4 items
collecting 6 items
collected 6 items

test_one.py .
test_two.py F
tasks\test_four.py ..
tasks\test_three.py ..

===== FAILURES =====
test_failing
def test_failing():
>     assert (1, 2, 3) == (3, 2, 1)
E       assert (1, 2, 3) == (3, 2, 1)
E         At index 0 diff: 1 != 3
E         Use -v to get the full diff
test_two.py:2: AssertionError
===== 1 failed, 5 passed in 0.20 seconds =====
(venv33) c:\BOOK\bopytest-code\code\ch1>
```



Где ищет тесты

pytest исследует текущий каталог и все подкаталоги для тестовых файлов и запустит тестовый код, который найдёт. Если вы передадите pytest имя файла, имя каталога или список из них, то будут найдены там вместо текущего каталога. Каждый каталог, указанный в командной строке, рекурсивно исследуется для поиска тестового кода.

```
1 import pytest
2
3
4 @pytest.fixture
5 def common_fixture():
6     print('Setup common fixture')
7
8
9 @pytest.fixture
10 def rare_fixture(common_fixture):
11     print('Setup rare fixture')
12
13
14 @pytest.fixture
15 def fixture_for_test_a(rare_fixture):
16     print('Setup fixture for test a')
17
18
19 def test_a(rare_fixture):
20     assert True == True
21
22
23 def test_b(fixture_for_test_a):
24     assert True == True
```



Обозначения в отчете

- **PASSED (.)**: Тест выполнен успешно
- **FAILED (F)**: Тест не выполнен успешно (или XPASS + strict)
- **SKIPPED (s)**: Тест был пропущен. Вы можете заставить pytest пропустить тест, используя декораторы `@pytest.mark.skip()` или `pytest.mark.skipif()`
- **xfail (x)**: Тест не должен был пройти, был запущен и провалился. Вы можете принудительно указать pytest, что тест должен завершиться неудачей, используя декоратор `@pytest.mark.xfail()`
- **XPASS (X)**: Тест не должен был пройти, был запущен и прошел
- **ERROR (E)**: Исключение произошло за пределами функции тестирования, либо в фикстуре



Как именовать что-то с тестами

- Функции следует именовать в стиле
snake_case_with_underscores
- Модули следует именовать в стиле
snake_case_with_underscores
- Файлы тестов, как правило, размещаются в папке «tests»

```
1 import pytest
2
3
4 @pytest.fixture
5 def common_fixture():
6     print('Setup common fixture')
7
8
9 @pytest.fixture
10 def rare_fixture(common_fixture):
11     print('Setup rare fixture')
12
13
14 @pytest.fixture
15 def fixture_for_test_a(rare_fixture):
16     print('Setup fixture for test a')
17
18
19 def test_a(rare_fixture):
20     assert True == True
21
22
23 def test_b(fixture_for_test_a):
24     assert True == True
```



Фикстуры

Для определения фикстур в
pytest используется декоратор
`@pytest.fixture`

```
1 import pytest
2
3
4 @pytest.fixture
5 def common_fixture():
6     print('Setup common fixture')
7
8
9 @pytest.fixture
10 def rare_fixture(common_fixture):
11     print('Setup rare fixture')
12
13
14 @pytest.fixture
15 def fixture_for_test_a(rare_fixture):
16     print('Setup fixture for test a')
17
18
19 def test_a(rare_fixture):
20     assert True == True
21
22
23 def test_b(fixture_for_test_a):
24     assert True == True
```



Маркеры

Можно пометить некоторый тест маркером (декоратор `@pytest.mark.<что-то>`), чтобы потом было проще вызвать его из командной строки. Например:

- `pytest -m run_these_please`

```
1 @pytest.mark.run_these_please
2 def test_access():
3     assert user.is_admin == True
```



Параметризация тестов

У pytest есть много мощных встроенных функций. Одна из них – декоратор:

- `@pytest.mark.parametrize("name, name2", [val1, val2])`

```
1  @pytest.mark.parametrize(  
2      "a, b, c, expected", [  
3          (1, -2, -3, (16, 2, 3, -1)),  
4          (1, -3, -4, (25, 2, 4, -1)),  
5          (-1, -2, 15, (64, 2, -5, 3)),  
6          (1, 12, 36, (0, 1, -6, None)),  
7          (11, -32, 123, (-4388, 0, None, None)),  
8      ]  
9  )  
10 def test_formula(a, b, c, expected):  
11     assert resolve_equation(a, b, c) == expected  
12
```



Пропуск тестов и ожидание провала

Для пропуска тестов:

- `@pytest.mark.skip` декоратор для пропуска теста
- `pytest.skip("")` метод для пропуска внутри теста
- `@pytest.mark.skipif` декоратор для пропуска по условию

Когда ожидаем, что тест провалится:

- `@pytest.mark.xfail`

```
1 @pytest.mark.skip(reason="no way of currently testing this")
2 def test_the_unknown():
3     pass
4
5
6 def test_function():
7     if not valid_config():
8         pytest.skip("unsupported configuration")
9
10
11 if not sys.platform.startswith("win"):
12     pytest.skip("skipping windows-only tests", allow_module_level=True)
13
14
15 @pytest.mark.skipif(sys.version_info < (3, 7), reason="requires python3.7 or higher")
16 def test_function():
17     pass
18
19
20 @pytest.mark.xfail
21 def test_function():
22     pass
23
24
25 @pytest.mark.xfail(sys.platform == "win32", reason="bug in a 3rd party library")
26 def test_function():
27     pass
```



Проверка, что код генерирует ошибку

В тесте можно указать в менеджере контекста `pytest.raises()`, чтобы показать, что вы ожидаете исключения. В данном случае, если вызовется `ValueError`, тест пройдет.

У вас должно броситься исключение.

```
1 def test_raises():
2     with pytest.raises(ValueError):
3         some_code(123)
```



Command line interface (CLI)

Выполнение всех тестов:

- `pytest`

Запуск всех тестов из некоторого модуля:

- `pytest tests/test_some.py`

Запуск одного теста из модуля:

- `pytest tests/test_some.py::test_func`

Запуск тестов, которые называются в соответствии с выражением:

- `pytest -k «some or not_some" --collect-only`

Запуск с выводом более подробной информации:

- `pytest -v`

Запуск всех тестов до первой ошибки:

- `pytest -x`

Запуск одного или нескольких неудачных тестов:

- `pytest -lf`

Запуск одного или нескольких неудачных тестов потом всех остальных:

- `pytest -ff`

Показать в traceback локальные переменные:

- `pytest -l`

Выводим ошибки в сокращенном виде:

- `pytest --tb=line`



Конфигурация

В pytest есть основной конфигурационный файл `pytest.ini`. В нем можно изменить поведение pytest по умолчанию.

```
1 # pytest.ini
2 [pytest]
3 minversion = 6.0
4 addopts = -ra -q
5 testpaths =
6     tests
7     integration
```



Плюсы и минусы pytest

- ✓ Поддержкой встроенного утверждения `assert` вместо использования специальных методов `self.assert*()`.
 - ✓ Возможностью повторного запуска с пропущенного теста.
 - ✓ Наличием системы дополнительных плагинов.
 - ✓ Соответствие PEP8
 - ✓ Фикстуры разных уровней: функции, модуля, глобальные
 - ✓ Параметризованные тесты
 - ✓ Много всяких декораторов и хелперов для работы
 - ✓ Выделения цветом
- ✗ Mock не включен в функционал



Бенчмарк с pytest

Если вы решите использовать pytest в качестве исполнителя тестов, обратите внимание на плагин pytest-benchmark. Он предоставляет pytest фикстуру под названием benchmark. Любой вызываемый объект может быть передан benchmark(), он залогирует время вызываемого в результатах pytest.

Установить pytest-benchmark из PyPI можно с помощью pip:

- \$ pip install pytest-benchmark
- Затем можно добавить тест, использующий фикстуру и передающий вызываемый объект на выполнение

```
1 def test_my_func(benchmark):  
2     result = benchmark(test)
```



Покрытие кода тестами

Для определения покрытия кода тестами используется библиотека `pytest-cov`

```
(venv) D:\projects\bh_unittest_example>pytest --cov source
===== test session starts =====
platform win32 -- Python 3.9.0, pytest-6.2.2, py-1.10.0, pluggy-0.13.1
rootdir: D:\projects\bh_unittest_example, configfile: tox.ini, testpaths: pytests
plugins: cov-2.11.1
collected 1 item

pytests\test_calc.py . [100%]

----- coverage: platform win32, python 3.9.0-final-0 -----
Name                Stmts  Miss  Cover
-----
source\__init__.py      0      0   100%
source\calc.py          14      6    57%
-----
TOTAL                   14      6    57%

===== 1 passed in 0.07s =====

(venv) D:\projects\bh_unittest_example>
```





Ваши вопросы

что необходимо прояснить в рамках
данного раздела



Тесты

unittest

pytest

Линтеры

Tox



Линтеры

проверка кода на соответствие стандартам,
безопасности



Тесты

unittest

pytest

Линтеры

Tox

Что такое линтер

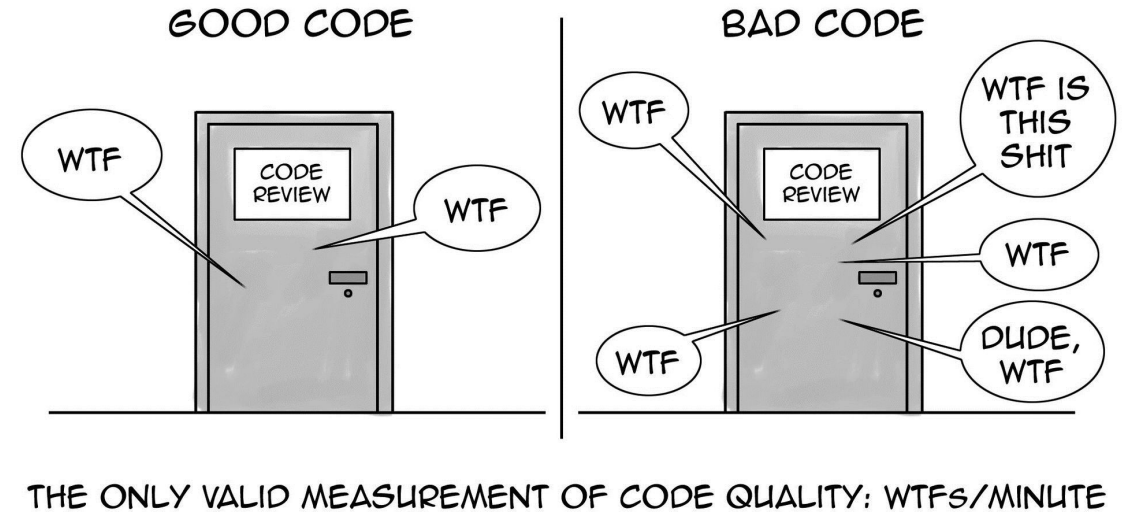
Линтер – программное обеспечение, которое проверяет код на соответствие некоторому стандарту.

Основная задача – единообразие стиля в проекте.



Code review

Code review – систематическая проверка исходного кода программы с целью обнаружения и исправления ошибок, которые остались незамеченными в начальной фазе разработки. Целью обзора является улучшение качества программного продукта и совершенствование навыков разработчика.



pylint

Pylint – статический инструмент для анализа кода Python. Этот мощный, гибко настраиваемый инструмент для анализа кода Python отличается большим количеством проверок и разнообразием отчетов. Это один из самых “придирчивых” и “многословных” анализаторов кода. Анализ нашего тестового скрипта выдает весьма обширный отчет, состоящий из списка найденных в ходе анализа недочетов, статистических отчетов, представленных в виде таблиц, и общей оценки код



Тесты

unittest

pytest

Линтеры

Tox

pylint

- [R]efactor — требуется рефакторинг,
- [C]onvention — нарушено следование стилистике и соглашениям,
- [W]arning — потенциальная ошибка,
- [E]rror — ошибка,
- [F]atal — ошибка, которая препятствует дальнейшей работе программы.
- — Генерация файла настроек (--generate-rcfile). Позволяет не писать конфигурационный файл с нуля. В созданном rcfile содержатся все текущие настройки с подробными комментариями к ним, вам остается только отредактировать его под собственные требования.



Flake8

flake8 — популярный линтер, который оставляет комментарии о стиле вашего кода в соответствии с PEP 8 спецификацией.

Flake8 имеет схожий с pylint основной функционал. Однако она имеет ряд отличий и особенностей:

- Возможности статистических отчетов ограничены подсчетом количества каждой из ошибок (`--statistics`) и их общим количеством (`--count`)
- Для запуска в несколько потоков (`--jobs=<num>`) используется модуль `multiprocessing`, по этой причине многопоточность не будет работать на Windows системах
- Отсутствует возможность генерации отчетов в формате `json`, при вызове с ключом `--bug-report` создается только заголовок для отчета с указанием платформы и версий входящих утилит
- — Комментарии в коде блокирующие вывод. Добавление в строку с ошибкой комментария `# noqa`, уберет ее из отчета.
- — Во время редактирования для подавления отдельных ошибок “на лету” можно перечислить исключаемые ошибки в ключе `--extend-ignore=<errors>`
- Проверка синтаксиса в строках `doctest` (`--doctests`).

Следующие возможности Flake8 можно отнести к его достоинствам. Наличие их сделало Flake8 весьма популярным инструментом среди разработчиков:

- Наличие Version Control Hooks. Интеграция с системами контроля версий происходит буквально с помощью двух команд (поддерживаются `git` и `mercurial`). Приведем пример для `git`. Настройка `hook` позволяет не допускать создания коммита при нарушении каких-либо правил оформления. Подробнее о механизме `git-hook` вы можете узнать в документации `git`



Выполнение анализа flake8

- `$ flake8 --ignore E305 --exclude .git,__pycache__ --max-line-length=90`

```
1  [flake8]
2  ignore = E305
3  exclude = .git,__pycache__
4  max-line-length = 90
```



Black



Black — очень неумолимый форматтер. В нем нет настроек и он очень дотошный. Что делает его отличным инструментом для вставки в ваш тестовый пайплайн.

- игнорирование не модифицированных файлов, программа запоминает, какие файлы она изменяла и при следующем запуске форматирует только файлы с внесенными изменениями;
- возможность запретить изменение отдельных блоков в коде, для этого используются комментарии: `# fmt: off` и `# fmt: on`, обозначающие начало и конец блока;
- длина строки по умолчанию является 88 символов, что не соответствует официальному соглашению PEP8;
- дополнительно можно установить HTTP сервер `blackd`, который позволяет избежать накладных расходов на создание процесса каждый раз, когда мы хотим отформатировать файл. Исходный код передается в теле POST запроса, а флаги управления форматированием в заголовках (флаги идентичны ключам командной строки, используемых при запуске `black`).

Обратите внимание: для `black` требуется Python версии 3.6 и выше.

- `$ black test.py`



bandit

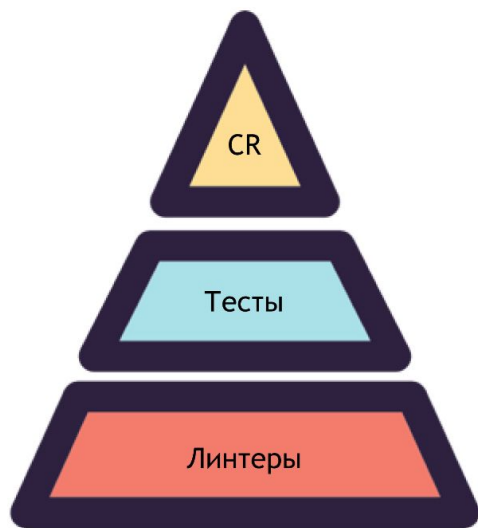
Bandit – утилита, которая проверяет наш код на предмет наличия в нем уязвимостей:

- `pip install bandit`



Плюсы и минусы использования линтеров

- ✓ Ускоряет процесс code review
- ✓ Единообразная кодовая база
- ✓ Проверка на соответствие стандарту, безопасности
- ✗ Дополнительная прослойка в виде ПО, проверяющего код
- ✗ Замедляет прогон автотестов и выкат в продакшн





Ваши вопросы

что необходимо прояснить в рамках
данного раздела



Тесты

unittest

pytest

Линтеры

Tox



Автоматизация тестирования

прогон тестов на различных версиях Python, создания сценария тестирования



Тесты

unittest

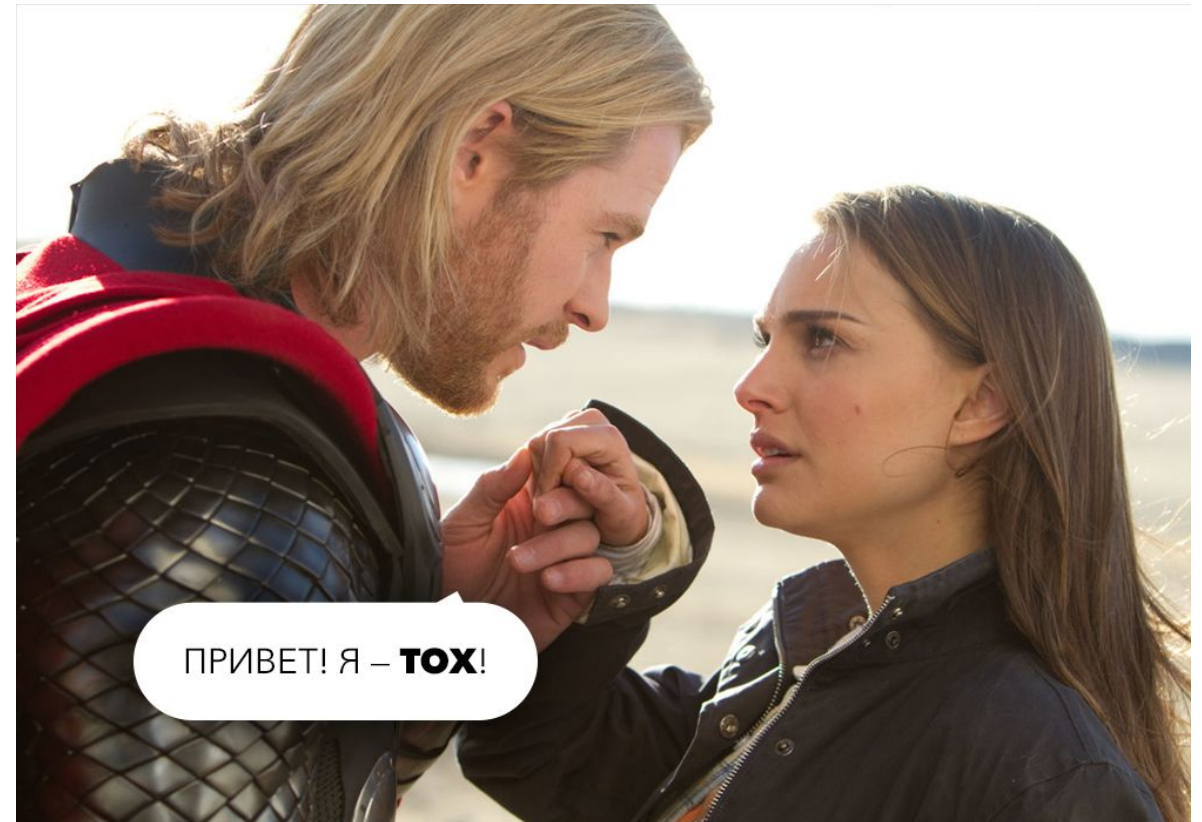
pytest

Линтеры

Tox

Tox

Tox – библиотека, которая автоматизирует тестирование в нескольких средах Python, позволяет создавать сценарии проверки.



Настройка tox

Tox настраивается через файл конфигурации в каталоге проекта. Он содержит следующее (tox.ini):

- Команда запуска для выполнения тестов
- Дополнительные пакеты, необходимые для выполнения
- Разные версии Python для тестирования



Command line interface (CLI)

Команда	Что делает
tox	Запускает все инструкции
tox -e py39	Запустить какой-то environment
tox -r	Повторное создание среды при изменении или повреждении site-packages
tox -q	Меньше информации в выводе
tox -v	Больше информации в выводе

