# Модуль unittest: тестируем свои программы

Представьте, что вы написали какую-либо программу, а теперь хотите проверить, правильно ли она работает. Что вы для этого сделаете? Скорее всего, вы запустите её несколько раз с различными входными данными, и убедитесь в правильности выдаваемого ответа.

А теперь вы что-то поменяли и снова хотите проверить корректность программы. Запускать ещё несколько раз? А если потом снова что-то поменяется? Нельзя ли как-то автоматизировать это дело?

Оказывается, можно. В Python встроен модуль unittest, который поддерживает автоматизацию тестов, использование общего кода для настройки и завершения тестов, объединение тестов в группы, а также позволяет отделять тесты от фреймворка для вывода информации.

Для автоматизации тестов, unittest поддерживает некоторые важные концепции:

* **Испытательный стенд** (test fixture) - выполняется подготовка, необходимая для выполнения тестов и все необходимые действия для очистки после выполнения тестов. Это может включать, например, создание временных баз данных или запуск серверного процесса.
* **Тестовый случай** (test case) - минимальный блок тестирования. Он проверяет ответы для разных наборов данных. Модуль unittest предоставляет базовый класс TestCase, который можно использовать для создания новых тестовых случаев.
* **Набор тестов** (test suite) - несколько тестовых случаев, наборов тестов или и того и другого. Он используется для объединения тестов, которые должны быть выполнены вместе.
* **Исполнитель тестов** (test runner) - компонент, который управляет выполнением тестов и предоставляет пользователю результат. Исполнитель может использовать графический или текстовый интерфейс или возвращать специальное значение, которое сообщает о результатах выполнения тестов.

Модуль unittest предоставляет богатый набор инструментов для написания и запуска тестов. Однако достаточно лишь некоторых из них, чтобы удовлетворить потребности большинства пользователей.

Вот короткий скрипт для тестирования трех [методов строк](https://pythonworld.ru/tipy-dannyx-v-python/stroki-funkcii-i-metody-strok.html):

**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())

**def** test\_split(self):

s = 'hello world'

self.assertEqual(s.split(), ['hello', 'world'])

*# Проверим, что s.split не работает, если разделитель - не строка*

**with** self.assertRaises(TypeError):

s.split(2)

**if** \_\_name\_\_ == '\_\_main\_\_':

unittest.main()

Тестовый случай создаётся путём [наследования](https://pythonworld.ru/osnovy/inkapsulyaciya-nasledovanie-polimorfizm.html) от unittest.TestCase. 3 отдельных теста определяются с помощью методов, имя которых начинается на test. Это соглашение говорит исполнителю тестов о том, какие методы являются тестами.

Суть каждого теста - вызов assertEqual() для проверки ожидаемого результата; assertTrue() или assertFalse() для проверки условия; assertRaises() для проверки, что метод порождает [исключение](https://pythonworld.ru/tipy-dannyx-v-python/isklyucheniya-v-python-konstrukciya-try-except-dlya-obrabotki-isklyuchenij.html). Эти методы используются вместо обычного assert для того, чтобы исполнитель тестов смог взять все результаты и оформить отчёт.

Методы setUp() и tearDown() (которые в данном простом случае не нужны) позволяют определять инструкции, выполняемые перед и после каждого теста, соответственно.

Последние 2 строки показывают простой способ запуска тестов. unittest.main() предоставляет интерфейс командной строки для тестирования программы. Будучи запущенным из командной строки, этот скрипт выводит отчёт, подобный этому:

...

----------------------------------------------------------------------

Ran 3 tests **in** 0.000s

OK

## Интерфейс командной строки

unittest может быть использован из командной строки для запуска модулей с тестами, классов или даже отдельных методов:

python -m unittest test\_module1 test\_module2

python -m unittest test\_module.TestClass

python -m unittest test\_module.TestClass.test\_method

Можно также указывать путь к файлу:

python -m unittest tests/test\_something.py

С помощью флага -v можно получить более детальный отчёт:

python -m unittest -v test\_module

Для нашего примера подробный отчёт будет таким:

test\_isupper (\_\_main\_\_.TestStringMethods) ... ok

test\_split (\_\_main\_\_.TestStringMethods) ... ok

test\_upper (\_\_main\_\_.TestStringMethods) ... ok

----------------------------------------------------------------------

Ran 3 tests **in** 0.001s

OK

-b (--buffer) - вывод программы при провале теста будет показан, а не скрыт, как обычно.

-c (--catch) - Ctrl+C во время выполнения теста ожидает завершения текущего теста и затем сообщает результаты на данный момент. Второе нажатие Ctrl+C вызывает обычное исключение KeyboardInterrupt.

-f (--failfast) - выход после первого же неудачного теста.

--locals (начиная с Python 3.5) - показывать локальные переменные для провалившихся тестов.

## Обнаружение тестов

unittest поддерживает простое обнаружение тестов. Для совместимости с обнаружением тестов, все файлы тестов должны быть модулями или пакетами, импортируемыми из директории верхнего уровня проекта (см. подробнее о [правилах наименования модулей](https://pythonworld.ru/osnovy/rabota-s-modulyami-sozdanie-podklyuchenie-instrukciyami-import-i-from.html#id3) ).

Обнаружение тестов реализовано в TestLoader.discover(), но может быть использовано из командной строки:

cd project\_directory

python -m unittest discover

-v (--verbose) - подробный вывод.

-s (--start-directory) directory\_name - директория начала обнаружения тестов (текущая по умолчанию).

-p (--pattern) pattern - шаблон названия файлов с тестами (по умолчанию test\*.py).

-t (--top-level-directory) directory\_name - директория верхнего уровня проекта (по умолчанию равна start-directory).

## Организация тестового кода

Базовые блоки тестирования это тестовые случаи - простые случаи, которые должны быть проверены на корректность.

Тестовый случай создаётся путём наследования от unittest.TestCase.

Тестирующий код должен быть самостоятельным, то есть никак не зависеть от других тестов.

Простейший подкласс TestCase может просто реализовывать тестовый метод (метод, начинающийся с test). Вымышленный пример:

**import** **unittest**

**class** **DefaultWidgetSizeTestCase**(unittest.TestCase):

**def** test\_default\_widget\_size(self):

widget = Widget('The widget')

self.assertEqual(widget.size(), (50, 50))

Заметьте, что для того, чтобы проверить что-то, мы используем один из assert\\*() методов.

Тестов может быть много, и часть кода настройки может повторяться. К счастью, мы можем определить код настройки путём реализации метода setUp(), который будет запускаться *перед* каждым тестом:

**import** **unittest**

**class** **SimpleWidgetTestCase**(unittest.TestCase):

**def** setUp(self):

self.widget = Widget('The widget')

**def** test\_default\_widget\_size(self):

self.assertEqual(self.widget.size(), (50,50),

'incorrect default size')

**def** test\_widget\_resize(self):

self.widget.resize(100,150)

self.assertEqual(self.widget.size(), (100,150),

'wrong size after resize')

Мы также можем определить метод tearDown(), который будет запускаться *после* каждого теста:

**import** **unittest**

**class** **SimpleWidgetTestCase**(unittest.TestCase):

**def** setUp(self):

self.widget = Widget('The widget')

**def** tearDown(self):

self.widget.dispose()

Можно разместить все тесты в том же файле, что и сама программа (таком как widgets.py), но размещение тестов в отдельном файле (таком как test\_widget.py) имеет много преимуществ:

* Модуль с тестом может быть запущен автономно из командной строки.
* Тестовый код может быть легко отделён от программы.
* Меньше искушения изменить тесты для соответствия коду программы без видимой причины.
* Тестовый код должен изменяться гораздо реже, чем программа.
* Протестированный код может быть легче переработан.
* Тесты для модулей на C должны быть в отдельных модулях, так почему же не быть последовательным?
* Если стратегия тестирования изменяется, нет необходимости изменения кода программы.

## Пропуск тестов и ожидаемые ошибки

unittest поддерживает пропуск отдельных тестов, а также классов тестов. Вдобавок, поддерживается пометка теста как "не работает, но так и надо".

Пропуск теста осуществляется использованием [декоратора](https://pythonworld.ru/osnovy/dekoratory.html) skip() или одного из его условных вариантов.

**class** **MyTestCase**(unittest.TestCase):

**@unittest**.skip("demonstrating skipping")

**def** test\_nothing(self):

self.fail("shouldn't happen")

**@unittest**.skipIf(mylib.\_\_version\_\_ < (1, 3),

"not supported in this library version")

**def** test\_format(self):

*# Tests that work for only a certain version of the library.*

**pass**

**@unittest**.skipUnless(sys.platform.startswith("win"), "requires Windows")

**def** test\_windows\_support(self):

*# windows specific testing code*

**pass**

test\_format (\_\_main\_\_.MyTestCase) ... skipped 'not supported in this library version'

test\_nothing (\_\_main\_\_.MyTestCase) ... skipped 'demonstrating skipping'

test\_windows\_support (\_\_main\_\_.MyTestCase) ... skipped 'requires Windows'

----------------------------------------------------------------------

Ran 3 tests **in** 0.005s

OK (skipped=3)

Классы также могут быть пропущены:

**@unittest**.skip("showing class skipping")

**class** **MySkippedTestCase**(unittest.TestCase):

**def** test\_not\_run(self):

**pass**

Ожидаемые ошибки используют декоратор expectedFailure():

**class** **ExpectedFailureTestCase**(unittest.TestCase):

**@unittest**.expectedFailure

**def** test\_fail(self):

self.assertEqual(1, 0, "broken")

Очень просто сделать свой декоратор. Например, следующий декоратор пропускает тест, если переданный объект не имеет указанного атрибута:

**def** skipUnlessHasattr(obj, attr):

**if** hasattr(obj, attr):

**return** **lambda** func: func

**return** unittest.skip("*{!r}* doesn't have *{!r}*".format(obj, attr))

Декораторы, пропускающие тесты или говорящие об ожидаемых ошибках:

@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().

## Различение итераций теста с помощью подтестов

Когда некоторые тесты имеют лишь незначительные отличия, например некоторые параметры, unittest позволяет различать их внутри одного тестового метода, используя [менеджер контекста](https://pythonworld.ru/osnovy/with-as-menedzhery-konteksta.html) subTest().

Например, следующий тест:

**class** **NumbersTest**(unittest.TestCase):

**def** test\_even(self):

*"""*

*Test that numbers between 0 and 5 are all even.*

*"""*

**for** i **in** range(0, 6):

**with** self.subTest(i=i):

self.assertEqual(i % 2, 0)

даст следующий отчёт:

======================================================================

FAIL: test\_even (\_\_main\_\_.NumbersTest) (i=1)

----------------------------------------------------------------------

Traceback (most recent call last):

File "subtests.py", line 32, **in** test\_even

self.assertEqual(i % 2, 0)

AssertionError: 1 != 0

======================================================================

FAIL: test\_even (\_\_main\_\_.NumbersTest) (i=3)

----------------------------------------------------------------------

Traceback (most recent call last):

File "subtests.py", line 32, **in** test\_even

self.assertEqual(i % 2, 0)

AssertionError: 1 != 0

======================================================================

FAIL: test\_even (\_\_main\_\_.NumbersTest) (i=5)

----------------------------------------------------------------------

Traceback (most recent call last):

File "subtests.py", line 32, **in** test\_even

self.assertEqual(i % 2, 0)

AssertionError: 1 != 0

Без использования подтестов, выполнение будет остановлено после первой ошибки, и ошибку будет сложнее диагностировать, потому что значение i не будет показано:

======================================================================

FAIL: test\_even (\_\_main\_\_.NumbersTest)

----------------------------------------------------------------------

Traceback (most recent call last):

File "subtests.py", line 32, **in** test\_even

self.assertEqual(i % 2, 0)

AssertionError: 1 != 0

## Проверки на успешность

Модуль unittest предоставляет множество функций для самых различных проверок:

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) порождает исключение exc

assertRaisesRegex(exc, r, fun, \*args, \*\*kwds) — fun(\*args, \*\*kwds) порождает исключение exc и сообщение соответствует регулярному выражению r

assertWarns(warn, fun, \*args, \*\*kwds) — fun(\*args, \*\*kwds) порождает предупреждение

assertWarnsRegex(warn, r, fun, \*args, \*\*kwds) — fun(\*args, \*\*kwds) порождает предупреждение и сообщение соответствует регулярному выражению r

assertAlmostEqual(a, b) — round(a-b, 7) == 0

assertNotAlmostEqual(a, b) — round(a-b, 7) != 0

assertGreater(a, b) — a > b

assertGreaterEqual(a, b) — a >= b

assertLess(a, b) — a < b

assertLessEqual(a, b) — a <= b

assertRegex(s, r) — r.search(s)

assertNotRegex(s, r) — not r.search(s)

assertCountEqual(a, b) — a и b содержат те же элементы в одинаковых количествах, но порядок не важен