

# Модульное тестирование (Unit Testing)

**Модульное тестирование (Unit Testing)** – это тип тестирования программного обеспечения, при котором тестируются отдельные модули или компоненты программного обеспечения.

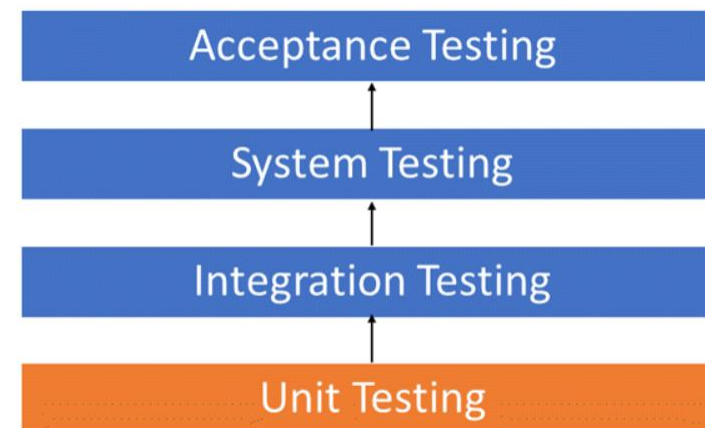
**Цель** модульного тестирования – проверить, что каждая единица программного кода работает должным образом.

Модульное тестирование это метод тестирования WhiteBox, который обычно **выполняется разработчиками** на этапе кодирования приложения.

**Модульные тесты** изолируют часть кода и проверяют его работоспособность.

**Единицей для измерения** может служить отдельная функция, метод, процедура, модуль или объект.

**Модульное тестирование – это первый уровень тестирования, выполняемый перед интеграционным тестированием.**



## Модульные тесты:

- ✓ **позволяют исправить ошибки на ранних этапах разработки** и сэкономить время и усилия, которые могли бы быть потрачены на поиск и исправление проблем в уже сложившемся коде;
- ✓ **помогают предотвратить появление новых ошибок** при внесении изменений в код в будущем;
- ✓ **способствуют повышению надежности кода.** Хороший набор тестов гарантирует, что код будет работать стабильно и предсказуемо даже в сложных ситуациях;
- ✓ **служат документацией к коду.** Описание ожидаемого поведения кода в явном виде делает код более понятным и облегчает его использование и поддержку для других разработчиков;
- ✓ **помогают с миграцией кода.** Просто переносите код и тесты в новый проект и изменяете код, пока тесты не запустятся снова.

# Процесс написания юнит-тестов

Написание юнит-тестов включает несколько шагов:

- ✓ **Определение тестируемых «юнитов».** Выберите функции, методы или классы, которые вы хотите протестировать. Определите, какие части вашего кода должны быть покрыты тестами.
- ✓ **Разработка тестовых случаев.** Определите различные сценарии использования и состояния, которые должны быть протестированы. Напишите тестовые случаи для каждого сценария, где вы проверяете ожидаемые результаты и поведение кода.
- ✓ **Написание тестового кода.** Напишите код для каждого тестового случая, который проверяет ожидаемый результат. Используйте специальные фреймворки и библиотеки для тестирования, чтобы упростить процесс написания и выполнения тестов.
- ✓ **Запуск тестов.** Запустите все написанные тесты и проверьте их результаты. Убедитесь, что все тесты прошли успешно и код работает правильно.
- ✓ **Анализ результатов и исправление ошибок.** Изучите результаты тестов и обратите внимание на любые ошибки или неожиданное поведение. Исправьте проблемы в вашем коде и перезапустите тесты для проверки исправлений.

# Как правильно писать юнит тесты?

Здесь следует придерживаться нескольких принципов:

- ✓ **Напишите тесты сразу.** Лучше писать тесты вместе с кодом или как можно раньше в процессе разработки. Это помогает предотвратить появление ошибок и упростить обнаружение проблем.
- ✓ **Будьте ясны и специфичны.** Называйте ваши тестовые случаи и функции таким образом, чтобы было понятно, что они проверяют. Сделайте каждый тестовый случай максимально специфичным и фокусируйтесь на конкретных аспектах кода.
- ✓ **Используйте ассерты.** Используйте утверждения (assertions) для проверки ожидаемых результатов. Убедитесь, что ваш код возвращает ожидаемые значения и ведет себя так, как вы предполагаете.
- ✓ **Поддерживайте свои тесты.** Обновляйте ваши тесты при внесении изменений в код. Убедитесь, что они все еще актуальны и продолжают проверять правильное поведение вашего кода.
- ✓ **Запускайте тесты регулярно,** чтобы убедиться, что они все еще работают. Это поможет вам быстро обнаружить любые проблемы в вашем коде.

# Структура Unit теста

Любой тест должен содержать:

1. Входные данные.
2. Тестовый сценарий, то есть набор шагов, которые надо выполнить для получения результата.
3. Проверка ожидаемого результата.

## AAA (Arrange, Act, Assert) Паттерн

**Arrange** (настройка) - в этом блоке кода настраиваем тестовое окружение тестируемого юнита; **Act** - выполнение или вызов тестируемого сценария; **Assert** - проверка, что тестируемый вызов ведет себя определенным образом.

# Структура юнит-теста

## 1. Тестовые фикстуры

Тестовые фикстуры – это компоненты модульного теста, отвечающие за подготовку среды, необходимой для выполнения тест-кейса. Они создают начальные условия для тестируемого блока, чтобы лучше контролировать выполнение теста. Тестовые фикстуры очень важны для автоматизированных модульных тестов, поскольку они обеспечивают постоянную тестовую среду для всего процесса тестирования.

Допустим, у нас есть приложение для ведения блога, и мы хотим протестировать модуль создания поста. Тестовые фикстуры должны включать:

- Подключение к базе данных
- Образец поста с заголовком, содержанием, информацией об авторе и т. д.
- Временное хранилище для обработки почтовых вложений
- Настройки конфигурации (видимость сообщений по умолчанию, параметры форматирования и т. д.)
- Тестовую учетную запись пользователя
- Песочницу (Sandbox environment) – среду для изоляции тестирования от продакшен среды и предотвращения вмешательства в реальные данные блога.

# Структура юнит-теста

## 2. Тест-кейс

Тест-кейс – это часть кода, которая проверяет поведение другого участка кода. При помощи тест-кейсов мы можем убедиться, что тестируемый модуль работает так, как ожидается, и дает желаемые результаты. Разработчики также должны писать *asserts*, чтобы конкретно определить, какие именно результаты ожидаются.

## 3. Test Runner

**Test Runner** – это программа, которая управляет запуском модульных тестов и предоставляет отчеты о результатах тестирования. Очень важно, что test runner может запускать тесты по приоритету, а также управлять тестовой средой и настройками для выполнения тестов. Он также помогает изолировать тестируемый модуль от внешних зависимостей.



# Структура юнит-теста

## 4. Тестовые данные

Тестовые данные должны быть тщательно подобраны, чтобы охватить как можно больше сценариев для выбранного блока кода и обеспечить высокое тестовое покрытие. Как правило, предполагается подготовить данные для:

- Обычных случаев: ожидаемые входные данные для тестируемого блока кода.
- Граничных случаев: входные значения находятся на границе допустимого предела.
- Ошибочных случаев: недействительные входные данные, чтобы увидеть, как блок кода реагирует на ошибки (посредством сообщений об ошибках или определенного поведения).
- Угловых случаев: входные данные представляют собой крайние случаи, которые оказывают значительное влияние на блок кода или систему.

# Структура юнит-теста

## 5. Моки и стабы

Моки – это, по сути, заменители реальных зависимостей тестируемого модуля. В модульном тестировании разработчики стараются проверить работу отдельных частей кода независимо друг от друга. Однако иногда для проверки одной части кода может понадобиться использовать другие компоненты программы.

Например, у нас есть класс **User**, который зависит от класса **EmailSender** для отправки уведомлений по электронной почте. В классе **User** есть метод **sendWelcomeEmail()**, который вызывает **EmailSender** для отправки приветственного письма только что зарегистрированному пользователю. Чтобы протестировать метод **sendWelcomeEmail()** без фактической отправки электронных писем, мы можем создать **мок-объект** класса **EmailSender**.

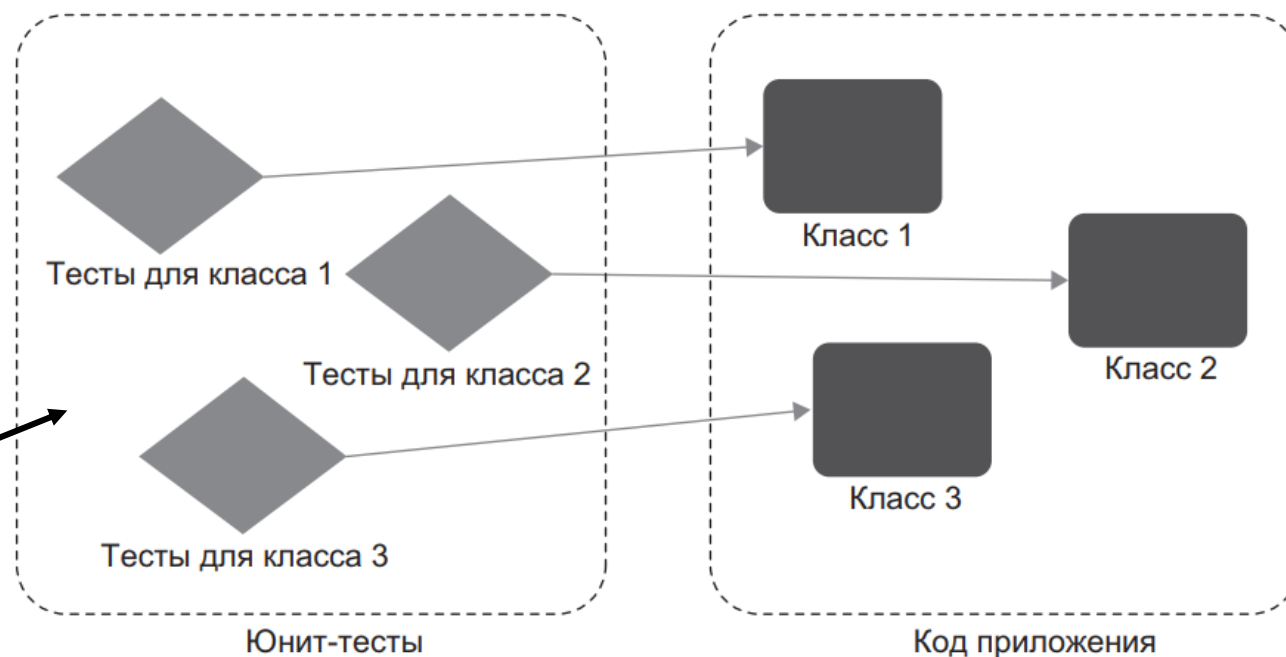
## Что означает «изоляция кода» в юнит-тестировании?

Разработчик может изолировать единицу кода для более качественного тестирования. Эта практика подразумевает копирование кода в собственную среду тестирования. *Изоляция кода помогает выявить ненужные зависимости между тестируемым кодом и другими модулями или пространствами данных в продукте.*

Если класс имеет зависимость от другого класса или нескольких классов, все такие зависимости должны быть заменены на тестовые заглушки (test doubles). Это позволяет сосредоточиться исключительно на тестируемом классе, изолировав его поведение от внешнего влияния.

**Тестовая заглушка (test double)** — объект, который выглядит и ведет себя как его рабочий аналог, но в действительности представляет собой упрощенную версию, более удобную для тестирования.

Изоляции юнит-тестов позволяет установить простые правила тестирования рабочего кода: одному классу соответствует один тест.



# Тестовые заглушки (тестовые двойники)

Тестовый двойник – это объект, который может заменить реальную часть кода для тестирования, примерно как дублер заменяет актера в фильме. Существует пять основных видов тестовых двойников: пустышка (dummy), фейк (fake), заглушка (stub), мок (mock), шпионы (spies).

- ✓ **Пустышки.** Например, пустышки используются для заполнения списка параметров, чтобы код компилировался и компилятор “не ругался”.
- ✓ **Фейки.** Например, объект базы данных, который мы можем использовать исключительно в тестовых сценариях, в то время как в продакшене используются реальные данные из БД.
- ✓ **Заглушки.** Допустим, у нас есть объект, который при вызове метода должен получить данные из базы данных для ответа. Вместо реального модуля мы можем использовать заглушку, которая вернет статичные данные.
- ✓ **Моки.** Моки очень похожи на заглушки, но они более ориентированы на “взаимодействие”. В качестве примера отлично подойдет функциональность, которая вызывает службу отправки сообщений по электронной почте. Нас не интересует отправлено ли нужное письмо. Нам нужно убедиться, что сервис отправки сообщений по электронной почте в принципе вызывался.
- ✓ **Шпионы.** Шпионы – это те же заглушки, но записывающие информацию о том, кто, как и когда их вызвал. Для примера снова подойдет сервис отправки сообщений по электронной почты, который фиксирует количество отправленных сообщений.

## Избегайте команд if в тестах

Наличие в тестах команды if также является антипаттерном.

**Тест (неважно, юнит- или интеграционный) должен представлять собой простую последовательность шагов без ветвлений.**

Присутствие команды if означает, что тест проверяет слишком много всего. Следовательно, такой тест должен быть разбит на несколько тестов. Ветвление в тестах не дает ничего, кроме дополнительных затрат на сопровождение: команды if затрудняют чтение и понимание тестов.



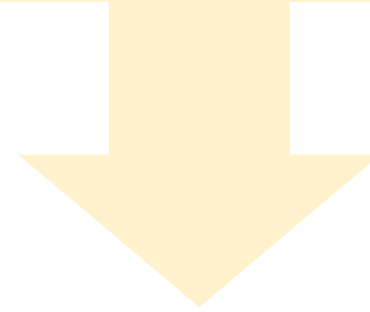
## Снижайте количество тест-кейсов для юнита

В среднем на юнит должно приходиться **от 1 до 9 тест-кейсов**. Это очень хороший индикатор качества юнита - если тест-кейсов больше или хочется их сгруппировать, нам точно нужно разделить его на два и больше независимых юнитов.

Для упрощения написания и запуска тестов существуют специальные фреймворки для разных языков программирования, например:

- ✓ **unittest**: встроенный модуль для тестирования в языке программирования Python. Он предоставляет широкие возможности и включает в себя возможности для создания фейковых объектов (mock objects).
- ✓ **pytest**: популярный фреймворк для тестирования в Python. Обладает простым и понятным синтаксисом и предоставляет множество функций для удобного написания и выполнения тестов.
- ✓ **JUnit**: фреймворк для тестирования в языке Java. Предоставляет средства для написания и запуска юнит-тестов и широко используется в разработке на Java.
- ✓ **NUnit**: фреймворк для тестирования в языке C#. Предоставляет возможности для создания и запуска тестов в C# и обладает богатым функционалом.
- ✓ **Mocha**: фреймворк для тестирования в JavaScript. Позволяет писать тесты с использованием синтаксиса, близкого к естественному языку, и обладает широкой поддержкой различных типов тестов.

Вся последующая информация лекции дана для Python



## Формат кода в Unittest

- ✓ тесты должны быть написаны в классе;
- ✓ класс должен быть унаследован от базового класса `unittest.TestCase`;
- ✓ имена всех функций, являющихся тестами, должны начинаться с ключевого слова **test\_**;
- ✓ вместо `assert` должны использоваться специальные `assertion` методы— именно они будут проверять полученные значения на соответствие заявленным.

# Пример тестирования приложения с использованием unittest

*Модуль calc.py*

```
def add(a, b):  
    return a + b
```

```
def sub(a, b):  
    return a - b
```

```
def mul(a, b):  
    return a * b
```

```
def div(a, b):  
    return a / b
```

```
import unittest  
import calc  
  
class CalcTest(unittest.TestCase):  
    def test_add(self):  
        self.assertEqual(calc.add(1, 2), 3)  
  
    def test_sub(self):  
        self.assertEqual(calc.sub(4, 2), 2)  
  
    def test_mul(self):  
        self.assertEqual(calc.mul(2, 5), 10)  
  
    def test_div(self):  
        self.assertEqual(calc.div(8, 4), 2)  
  
if __name__ == '__main__':  
    unittest.main()
```



# Основные структурные элементы unittest

**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.

**Test case** (прецедент) – это элементарная единица тестирования, в рамках которой проверяется работа компонента тестируемой программы. Для реализации этой сущности используется класс *TestCase*.

**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.

**Test runner** – это компонент, который координирует запуск тестов и предоставляет пользователю результат их выполнения.

# TestCase

TestCase представляет собой базовый (родительский) класс для всех остальных классов, методы которых будут тестировать автономные единицы исходной программы.

При выборе имени класса наследника от TestCase можно руководствоваться следующим правилом:

- ✓ **[ИмяТестируемойСущности]Tests.** [ИмяТестируемойСущности] – это некоторая логическая единица, тесты для которой нужно написать. В нашем случае – это калькулятор, поэтому мы выбрали имя **CalcTests**.
- ✓ Если бы у нашего калькулятора был большой набор поддерживаемых функций, то тестирование простых функций (сложение, вычитание, умножение и деление) можно было бы вынести в отдельный класс и назвать его например так: CalcSimpleActionsTests.

Методы класса TestCase можно разделить на три группы:

- ✓ методы, используемые при запуске тестов;
- ✓ методы, используемые при непосредственном написании тестов (проверка условий, сообщение об ошибках);
- ✓ методы, позволяющие собирать информацию о самом тесте.

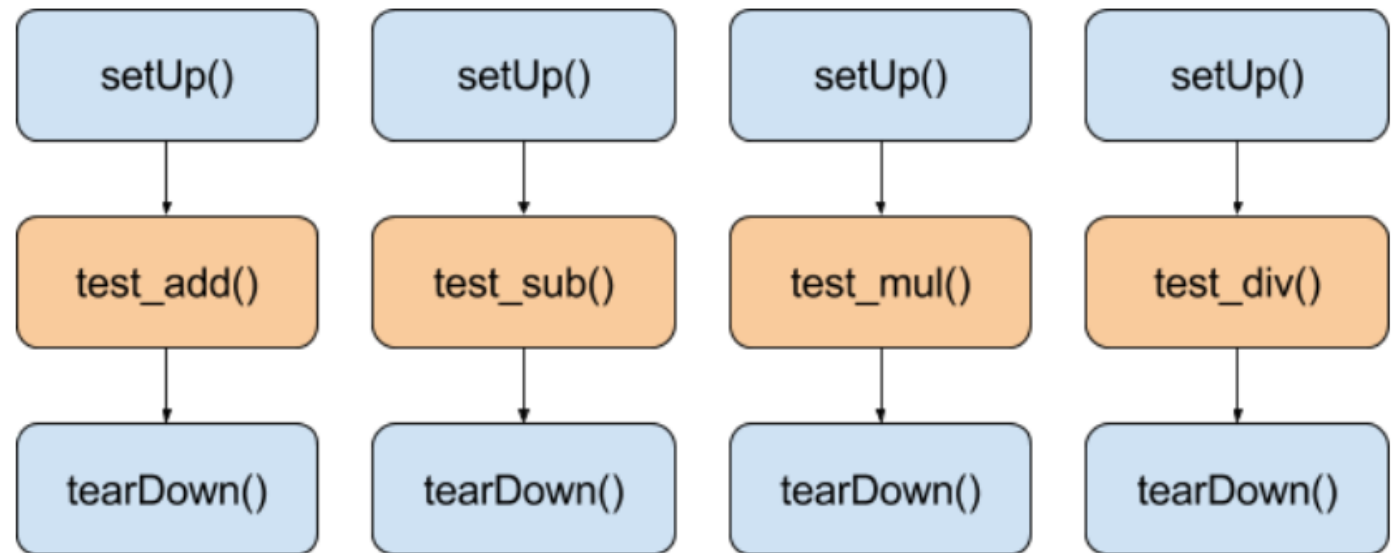
# Методы, используемые при запуске тестов

К этим методам относятся:

- **setUp()** Метод вызывается перед запуском теста. Как правило, используется для подготовки окружения для теста.
- **tearDown()** Метод вызывается после завершения работы теста. Используется для “уборки” за тестом.

Методы `setUp()` и `tearDown()` вызываются для всех тестов в рамках класса, в котором они переопределены.

По умолчанию, эти методы ничего не делают.



# Методы, используемые при запуске тестов

- Методы **setUpClass()** и **tearDownClass()** действуют на уровне класса, т.е. выполняется перед запуском тестов класса и после выполнения всех методов класса соответственно.

- **skipTest(reason)**

Данный метод может использоваться для пропуска теста, если это необходимо.

```
import unittest

class TestExample(unittest.TestCase):

    def test_another_example(self):
        # Предположим, этот тест не работает на определенной платформе
        platform = 'Windows'
        if platform == 'Windows':
            self.skipTest("Этот тест не работает на Windows")

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

# Методы, используемые при непосредственном написании тестов

*(проверка условий,  
сообщение об ошибках)*

наиболее часто  
используемые  
методы

<https://docs.python.org/3/library/unittest.html#assert-methods>

Method	Checks that
<a href="#"><code>assertEqual(a, b)</code></a>	<code>a == b</code>
<a href="#"><code>assertNotEqual(a, b)</code></a>	<code>a != b</code>
<a href="#"><code>assertTrue(x)</code></a>	<code>bool(x) is True</code>
<a href="#"><code>assertFalse(x)</code></a>	<code>bool(x) is False</code>
<a href="#"><code>assertIs(a, b)</code></a>	<code>a is b</code>
<a href="#"><code>assertIsNot(a, b)</code></a>	<code>a is not b</code>
<a href="#"><code>assertIsNone(x)</code></a>	<code>x is None</code>
<a href="#"><code>assertIsNotNone(x)</code></a>	<code>x is not None</code>
<a href="#"><code>assertIn(a, b)</code></a>	<code>a in b</code>
<a href="#"><code>assertNotIn(a, b)</code></a>	<code>a not in b</code>
<a href="#"><code>assertIsInstance(a, b)</code></a>	<code>isinstance(a, b)</code>
<a href="#"><code>assertNotIsInstance(a, b)</code></a>	<code>not isinstance(a, b)</code>

**assertEqual(a, b)** - это метод в модуле unittest, который используется для проверки того, что два объекта равны друг другу.

```
unittest.assertEqual(a, b, msg=None)
```

**msg** (необязательно) - все методы assert принимают аргумент msg, который, используется как сообщение об ошибке при сбое:

```
self.assertEqual(a, b, msg="Значения разные")
```

```
import unittest

class TestExample(unittest.TestCase):

    def test_equal_values(self):
        self.assertEqual(3, 3)

    def test_not_equal_values(self):
        self.assertEqual(3, 4)

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

- **test\_equal\_values()** - тест успешен, так как значения 3 и 3 равны.
- **test\_not\_equal\_values()** - тест неудачен, так как значения 3 и 4 не равны.

Также можно проверить создание исключений, предупреждений и сообщений журнала, используя следующие методы:

Подробнее : <https://docs.python.org/3/library/unittest.html#assert-methods>

Method	Checks that
<a href="#"><code>assertRaises(exc, fun, *args, **kws)</code></a>	<code>fun(*args, **kws)</code> raises <i>exc</i>
<a href="#"><code>assertRaisesRegex(exc, r, fun, *args, **kws)</code></a>	<code>fun(*args, **kws)</code> raises <i>exc</i> and the message matches regex <i>r</i>
<a href="#"><code>assertWarns(warn, fun, *args, **kws)</code></a>	<code>fun(*args, **kws)</code> raises <i>warn</i>
<a href="#"><code>assertWarnsRegex(warn, r, fun, *args, **kws)</code></a>	<code>fun(*args, **kws)</code> raises <i>warn</i> and the message matches regex <i>r</i>
<a href="#"><code>assertLogs(logger, level)</code></a>	The <code>with</code> block logs on <i>logger</i> with minimum <i>level</i>
<a href="#"><code>assertNoLogs(logger, level)</code></a>	The <code>with</code> block does not log on <i>logger</i> with minimum <i>level</i>



**assertRaisesRegex()** - это метод, который используется для проверки того, что вызов функции порождает исключение определенного типа и с заданным сообщением.

```
unittest.assertRaisesRegex(exception, regex, callable, *args, **kwargs)
```

- **exception** - ожидаемый тип исключения.
- **regex** - регулярное выражение, которое должно соответствовать сообщению исключения.
- **callable** - вызываемая функция или метод.
- **args** и **kwargs** - аргументы и ключевые аргументы для вызываемой функции или метода.

```
import unittest

def raise_value_error():
    raise ValueError("Division by zero error")

class TestExample(unittest.TestCase):

    def test_raise_value_error(self):
        with self.assertRaisesRegex(ValueError, r"Division by zero"):
            raise_value_error()

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

- **test\_raise\_value\_error()**  
- тест успешен, так как вызов функции `raise_value_error()` порождает исключение `ValueError` с сообщением, содержащим строку "Division by zero".

Существуют также другие методы, используемые для проведения более конкретных проверок:

[Подробнее : https://docs.python.org/3/library/unittest.html#assert-methods](https://docs.python.org/3/library/unittest.html#assert-methods)

Method	Checks that
<a href="#"><code>assertAlmostEqual(a, b)</code></a>	<code>round(a-b, 7) == 0</code>
<a href="#"><code>assertNotAlmostEqual(a, b)</code></a>	<code>round(a-b, 7) != 0</code>
<a href="#"><code>assertGreater(a, b)</code></a>	<code>a &gt; b</code>
<a href="#"><code>assertGreaterEqual(a, b)</code></a>	<code>a &gt;= b</code>
<a href="#"><code>assertLess(a, b)</code></a>	<code>a &lt; b</code>
<a href="#"><code>assertLessEqual(a, b)</code></a>	<code>a &lt;= b</code>
<a href="#"><code>assertRegex(s, r)</code></a>	<code>r.search(s)</code>
<a href="#"><code>assertNotRegex(s, r)</code></a>	<code>not r.search(s)</code>
<a href="#"><code>assertCountEqual(a, b)</code></a>	<code>a</code> and <code>b</code> have the same elements in the same number, regardless of their order.

**assertRegex()** - это метод в модуле unittest, который используется для проверки того, что строка соответствует заданному регулярному выражению.

```
unittest.assertRegex(text, regex, msg=None)
```

- **text** - строка, которую нужно проверить на соответствие регулярному выражению.
- **regex** - регулярное выражение.
- **msg** (необязательно) - сообщение, которое будет выведено в случае ошибки.

```
import unittest
class TestExample(unittest.TestCase):

    def test_regex_match(self):
        text = "example123"
        regex = r'\w+\d+'
        self.assertRegex(text, regex)

    def test_regex_no_match(self):
        text = "example"
        regex = r'\d+'
        self.assertRegex(text, regex)

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

- **test\_regex\_match()** - тест успешен, так как строка "example123" соответствует регулярному выражению `r'\w+\d+'`.
- **test\_regex\_no\_match()** - тест неудачен, так как строка "example" не соответствует регулярному выражению `r'\d+'`.

# Методы, позволяющие собирать информацию о самом тесте

---

- **countTestCases()**

Возвращает количество тестов в объекте класса-наследника от TestCase.

- **id()**

Возвращает строковый идентификатор теста. Как правило это полное имя метода, включающее имя модуля и имя класса.

- **shortDescription()**

Возвращает описание теста, которое представляет собой первую строку docstring'а метода, если его нет, то возвращает None.

И т.д. см. [документацию](#)

**countTestCases()** - это метод в модуле `unittest.TestSuite`, который используется для подсчета количества тестовых случаев (тестовых методов) в наборе тестов.

```
test_suite.countTestCases()
```

- `test_suite` - объект класса `unittest.TestSuite`, для которого нужно подсчитать количество тестовых случаев.

```
import unittest

class TestExample(unittest.TestCase):

    def test_case_1(self):
        self.assertEqual(1, 1)

    def test_case_2(self):
        self.assertEqual(2, 2)

    def test_case_3(self):
        self.assertEqual(3, 3)

if __name__ == '__main__':
    suite = unittest.TestLoader().loadTestsFromTestCase(TestExample)
    print("Количество тестовых случаев:", suite.countTestCases())
```

**shortDescription()** в модуле unittest используется для получения краткого описания теста.

```
import unittest

class TestExample(unittest.TestCase):

    def test_addition(self):
        """
        Тест на сложение двух чисел.
        """
        description = self.test_addition.shortDescription()
        print(description)  # Выводит: Тест на сложение двух чисел.

    def test_subtraction(self):
        """
        Тест на вычитание двух чисел.
        """
        description = self.test_subtraction.shortDescription()
        print(description)  # Выводит: Тест на вычитание двух чисел.

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

## Unittest: аргументы ЗА

- ✓ Является частью стандартной библиотеки языка Python: не нужно устанавливать ничего дополнительно;
- ✓ Гибкая структура и условия запуска тестов. Для каждого теста можно назначить теги, в соответствии с которыми будем запускаться либо одна, либо другая группа тестов;
- ✓ Быстрая генерация отчетов о проведенном тестировании.

## Unittest: аргументы ПРОТИВ

- ✓ Для проведения тестирования придётся написать достаточно большое количество кода (по сравнению с другими библиотеками);
- ✓ Из-за того, что разработчики вдохновлялись форматом библиотеки JUnit, названия основных функций написаны в стиле camelCase (например setUp и assertEquals);
- ✓ В языке python согласно рекомендациям [pep8](#) должен использоваться формат названий snake\_case (например set\_up и assert\_equal).

# Фреймворк Pytest

## Формат кода

Написание тестов здесь намного проще, нежели в unittest. Вам нужно просто написать несколько функций, удовлетворяющих следующим условиям:

- ✓ Имя файла должно начинаться с **"test"** или заканчиваться **"test.py"**.
- ✓ Имена функций и переменных должны быть написаны в **нижнем** регистре, а слова должны быть разделены подчеркиванием. При этом имя тестовой функции должно начинаться с **"test\_"**, например **"test\_skillfactory"**.
- ✓ Внутри функции должно проверяться логическое выражение при помощи оператора `assert`.



# Asserts PyTest (утверждения в PyTest)

Asserts Pytest — это проверки, которые возвращают статус True или False. В Python Pytest, если в тестовом методе происходит сбой утверждения, то выполнение этого метода останавливается.

Примеры Asserts Pytest:

```
assert "hello" == "Hai" is an assertion failure.  
assert 4==4 is a successful assertion  
assert True is a successful assertion  
assert False is an assertion failure.
```

## Unittest

```
import unittest

class TestAbs(unittest.TestCase):
    def test_abs1(self):
        self.assertEqual(abs(-42), 42, "Should be absolute value of a number")

    def test_abs2(self):
        self.assertEqual(abs(-42), -42, "Should be absolute value of a number")

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

## Pytest

```
def test_abs1():
    assert abs(-42) == 42, "Should be absolute value of a number"

def test_abs2():
    assert abs(-42) == -42, "Should be absolute value of a number"
```

```
def add(a, b):  
    return a + b  
def sub(a, b):  
    return a-b  
def mul(a, b):  
    return a * b  
def div(a, b):  
    return a / b
```

## Unittest

```
import unittest  
import calc  
  
class CalcTest(unittest.TestCase):  
    def test_add(self):  
        self.assertEqual(calc.add(1, 2), 3)  
  
    def test_sub(self):  
        self.assertEqual(calc.sub(4, 2), 2)  
  
    def test_mul(self):  
        self.assertEqual(calc.mul(2, 5), 10)  
  
    def test_div(self):  
        self.assertEqual(calc.div(8, 4), 2)  
  
if __name__ == '__main__':  
    unittest.main()
```

## Pytest

```
# test_calculator.py  
import pytest  
from calculator import add, sub, mul, div  
  
def test_add():  
    assert add(2, 3) == 5  
    assert add(-1, 1) == 0  
  
def test_sub():  
    assert sub(5, 3) == 2  
    assert sub(1, 3) == -2  
  
def test_mul():  
    assert mul(2, 3) == 6  
    assert mul(-1, 3) == -3  
  
def test_div():  
    assert div(6, 3) == 2  
    assert div(10, 2) == 5  
  
# Тест на деление на ноль  
def test_div_by_zero():  
    with pytest.raises(ZeroDivisionError):  
        div(6, 0)
```


# PyTest: правила запуска тестов

```
pytest scripts/selenium_scripts
# найти все тесты в директории scripts/selenium_scripts

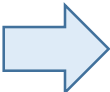
pytest test_user_interface.py
# найти и выполнить все тесты в файле

pytest scripts/drafts.py::test_register_new_user_parametrized
# найти тест с именем test_register_new_user_parametrized в указанном
файле в указанной директории и выполнить
```

- ✓ во всех директориях PyTest ищет файлы, которые удовлетворяют правилу **test\_\*.py** или **\*\_test.py** (то есть начинаются на test\_ или заканчиваются \_test и имеют расширение .py)
- ✓ внутри всех этих файлов находит тестовые методы:
  - все тесты, название которых начинается с **test**, которые находятся вне классов
  - все тесты, название которых начинается с **test** внутри классов, имя которых начинается с **Test** (и без метода `__init__` внутри класса)



```
test_login.py - valid
login_test.py - valid
testlogin.py -invalid
logintest.py -invalid
```



```
def test_file1_method1(): - valid
def testfile1_method1(): - valid
def file1_method1(): - invalid
```

## Pytest: аргументы ЗА

- ✓ Позволяет писать компактные (по сравнению с unittest) наборы тестов;
- ✓ В случае возникновения ошибок выводится гораздо больше информации о них;
- ✓ Позволяет запускать тесты, написанные для других тестирующих систем;
- ✓ Имеет систему плагинов, расширяющую возможности фреймворка. *Примеры таких плагинов: pytest-cov, pytest-django, pytest-bdd;*
- ✓ Позволяет запускать тесты в параллели (при помощи плагина pytest-xdist).

## Pytest: аргументы ПРОТИВ

- ✓ pytest не входит в стандартную библиотеку языка Python. Поэтому его придётся устанавливать отдельно при помощи команды `pip install pytest`;
- ✓ совместимость кода с другими фреймворками отсутствует. Т.е., если напишете код под pytest, запустить его при помощи встроенного unittest не получится.