**Тестовые сценарии**

Созданные тесты нужно сохранить в файле, чтобы его было удобно запускать и хранить в системе контроля версий. Давайте создадим файл**test\_abs\_project.py** и напишем в нём следующий код:

def test\_abs1():

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

if \_\_name\_\_ == "\_\_main\_\_":

test\_abs1()

print("All tests passed!")

Мы поместили тестовый сценарий в функцию для разделения тест-кейсов и возможности их независимого запуска.

Не вдаваясь в подробности, скажем только, что конструкция **if \_\_name\_\_ == "\_\_main\_\_"** служит для подтверждения того, что данный скрипт был запущен напрямую, а не вызван внутри другого файла в качестве модуля. Весь код написанный в теле этого условия будет выполнен только если пользователь запустил файл самостоятельно. Подробнее можно ознакомиться в видео [Олега Молчанова](https://www.youtube.com/watch?v=cW_-zGG4ef4).

В этой конструкции мы вызвали функцию**test\_abs1()**, которая выполняет тестовый сценарий.

С помощью **print("All tests passed!")** мы вывели сообщение, если все тесты прошли успешно.

Чтобы запустить тест, выполните в консоли команду:

python test\_abs\_project.py

Вы должны увидеть в консоли сообщение**"All tests passed!"**.

Если нам нужно добавить еще один тест, мы можем написать его как функцию в этом же файле. В приведенном примере мы уже не увидим сообщение "Everything passed", так как падение любого теста вызывает выход из программы:

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"

if \_\_name\_\_ == "\_\_main\_\_":

test\_abs1()

test\_abs2()

print("Everything passed")

Запустите файл снова. Вы должны увидеть сообщение об упавшем втором тесте:

$ python test\_abs\_project.py

Traceback (most recent call last):

  File "test\_abs\_project.py", line 9, in <module>

    test\_abs2()

  File "test\_abs\_project.py", line 5, in test\_abs2

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

AssertionError: Should be absolute value of a number

**Выбор test runner**

В предыдущих шагах мы научились писать простые тесты и запускать их с помощью Python. Приведём здесь код тестов и результаты запуска из предыдущего шага еще раз.

**test\_abs\_project.py:**

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"

if \_\_name\_\_ == "\_\_main\_\_":

    test\_abs1()

test\_abs2()

    print("Everything passed")

Консоль:

$ python test\_abs\_project.py

Traceback (most recent call last):

  File "test\_project.py", line 9, in <module>

    test\_abs2()

  File "test\_project.py", line 5, in test\_abs2

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

AssertionError: Should be absolute value of a number

Рассмотрим минусы такого подхода к запуску автотестов:

* Когда тестов становится много, сложно становится запускать только тесты из нужных тест-сьютов.
* Для каждого теста нужно создавать тестовые данные и окружение отдельно. Например, если мы захотим для каждого теста запускать браузер, а после завершения теста браузер закрывать, то логику работы с браузером придется дублировать в коде каждого теста.
* Если один из тестов завершится с ошибкой, например, тест упадёт с ошибкой AssertionError, то последующие тесты не запустятся. Мы не узнаем, были ли проблемы в этих тестах, пока не починим упавший тест или пока не запустим эти тесты по отдельности.

Для решения этих проблем и упрощения написания и запуска тестов существуют специальные фреймворки, которые называются test runners (тест-раннеры). Можно выделить три основных тестовых фреймворка для Python: **unittest**, **PyTest** и **nose**. Модуль **unittest** является встроенным инструментом Python — и это его большой плюс. **PyTest** и **nose** устанавливаются дополнительно, они позволяют получить расширенные возможности по сравнению с **unittest**. Мы кратко рассмотрим, как используется **unittest**, а затем изучим возможности **PyTest**, который позволяет писать более простой код тестов по сравнению с **unittest** и гибко настраивать запуск тестов. Еще один плюс использования **PyTest** в том, что для него существует большое количество плагинов, которые позволяют решить практически любую проблему, связанную с запуском автотестов.

## Фиксируем пакеты в requirements.txt

Количество пакетов в нашем проекте растет, а мы тем временем все дальше уходим от учебных кусочков скриптов в сторону настоящего тестового проекта, поэтому в этом шаге давайте зафиксируем все пакеты, которые мы используем. Это стандартная практика, которая позволяет быстро переключаться в свежее виртуальное окружение, а также работать нескольким людям над одним проектом, получая одинаковые результаты.

Откройте терминал, перейдите в директорию, в которой вы работаете с автотестами, и активируйте виртуальное окружение.

После чего выполните в терминале команду:

pip freeze > requirements.txt

Эта команда сохранит все версии пакетов в специальный файл **requirements.txt.**

Как их оттуда достать? Попробуйте создать новое виртуальное окружение (если нужно, вернитесь в [модуль 1](https://stepik.org/lesson/25969/step/3?unit=196192) за инструкциями) и активировать. После чего выполните команду:

pip install -r requirements.txt

В свежем окружении все пакеты установлены одной командой!

## ****PyTest: правила запуска тестов****

В этом шаге мы коротко обсудим важные особенности запуска тестов с помощью PyTest. Когда мы выполняем команду **pytest**, тест-раннер собирает все тесты для запуска по определенным правилам:

* если мы не передали никакого аргумента в команду, а написали просто pytest, тест-раннер начнёт поиск в текущей директории
* как аргумент можно передать файл, путь к директории или любую комбинацию директорий и файлов, например:

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

* дальше происходит рекурсивный поиск: то есть PyTest обойдет все вложенные директории
* 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 функции по следующему правилу:
  + - 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)

Если запустить PyTest с параметром **-v** (**verbose**, то есть подробный), то в отчёт добавится дополнительная информация со списком тестов и статусом их прохождения:



## PyTest — как пишут тесты

PyTest не требует написания дополнительных специфических конструкций в тестах, как того требует unittest.

Мы уже увидели, что PyTest может запускать тесты, написанные в unittest-стиле. Перепишем наши тесты из **test\_abs\_project.py** в более простом формате, который также понимает PyTest. Назовём новый файл test\_abs.py:

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"

Запустим тесты в этом файле:

pytest test\_abs.py

Код тестов стал короче и читабельнее.

## PyTest — проверка ожидаемого результата (assert)

Если вы используете unittest, то для проверки ожидаемых результатов в тестах вам нужно знать и использовать большой набор assert-методов, например, таких: assertEqual, assertNotEqual, assertTrue, assertFalse и [другие](https://docs.python.org/3/library/unittest.html#assert-methods%EF%BB%BF).

В PyTest используется стандартный assert метод из языка Python, что делает код более очевидным.

Давайте сравним два подхода. Проверим, что две переменные равны друг другу.

**unittest:**

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

**PyTest:**

assert a == b, "Значения разные"

С помощью assert можно проверять любую конструкцию, которая возвращает True/False. Это может быть проверка равенства, неравенства, содержания подстроки в строке или любая другая вспомогательная функция, которую вы опишете самостоятельно. Все это делает код проверок приятным и понятным для чтения:

assert user\_is\_authorised(), "User is guest"

Если нужно проверить, что тест вызывает ожидаемое исключение (довольно редкая ситуация для UI-тестов, и вам этот способ, скорее всего, никогда не пригодится), мы можем использовать специальную конструкцию **with pytest.raises()**. Например, можно проверить, что на странице сайта не должен отображаться какой-то элемент:

import pytest

from selenium import webdriver

from selenium.webdriver.common.by import By

from selenium.common.exceptions import NoSuchElementException

def test\_exception1():

try:

browser = webdriver.Chrome()

browser.get("http://selenium1py.pythonanywhere.com/")

with pytest.raises(NoSuchElementException):

browser.find\_element(By.CSS\_SELECTOR, "button.btn")

pytest.fail("Не должно быть кнопки Отправить")

finally:

browser.quit()

def test\_exception2():

try:

browser = webdriver.Chrome()

browser.get("http://selenium1py.pythonanywhere.com/")

with pytest.raises(NoSuchElementException):

browser.find\_element(By.CSS\_SELECTOR, "no\_such\_button.btn")

pytest.fail("Не должно быть кнопки Отправить")

finally:

browser.quit()

В первом тесте элемент будет найден, поэтому ошибка **NoSuchElementException**, которую ожидает контекстный менеджер pytest.raises, не возникнет, и тест упадёт.

test\_3\_3\_9\_pytest\_raises.py:8 (test\_exception1)

E Failed: Не должно быть кнопки Отправить

Во втором тесте, как мы и ожидали, кнопка не будет найдена, и тест пройдет.