ЛАБОРАТОРНАЯ РАБОТА №23

Инструкционно-технологическая карта

Тема: Составление тестового сценария для проведения автоматизированного тестирования

Цель работы: освоить навыки создания тестового сценария для проведения автоматизированного тестирования с использованием фреймворка Unittest.

Время выполнения: 4 часа

**Содержание работы**

1. Теоретические сведения для выполнения работы
2. Порядок выполнения работы
3. Контрольные вопросы
4. Литература
5. Теоретические сведения для выполнения работы

**Модульное тестирование в Python**

Фреймворк модульного тестирования [unittest](https://digitology.tech/docs/python_3/library/unittest.html#module-unittest) изначально был вдохновлён JUnit и очень походит на основные фреймворки модульного тестирования на других языках. Он поддерживает автоматизацию тестирования, совместное использование кода настройки и выключения для тестов, объединение тестов в коллекции и независимость тестов от фреймворков отчётности.

**Создание модульного теста**

При написании простого модульного теста необходимо выполнить следующие шаги:

1. Импортировать unittest в файл: import unittest
2. Создать класс, который должен наследоваться от класса TestCase: class SimpleTest (unittest.TestCase):
3. Превратить тестовые функции в методы, добавив ссылку на экземпляр класса self в качестве первого аргумента функции: def test\_add1(self):
4. Изменить assert на self.assertEqual(). Функция assertEquals() сравнивает результат функции add () с аргументом arg2 и выдает assertionError, если сравнение не удается.
5. Заменить строку запуска программы на unittest.main()
6. Вызвать метод main () из модуля unittest.

import unittest

def add(x,y):

return x + y

class SimpleTest(unittest.TestCase):

def test\_add1(self):

self.assertEquals(add(4,5),9)

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

unittest.main()

1. Запустить скрипт из командной строки.

python SimpleTest.py

C:\Python27>python SimpleTest.py

.

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

Ran 1 test in 0.000s

OK

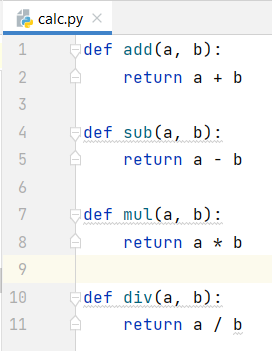
Символ «.» – говорит о том, что тест успешен

Символ «F» – вызывает исключение AssertionError

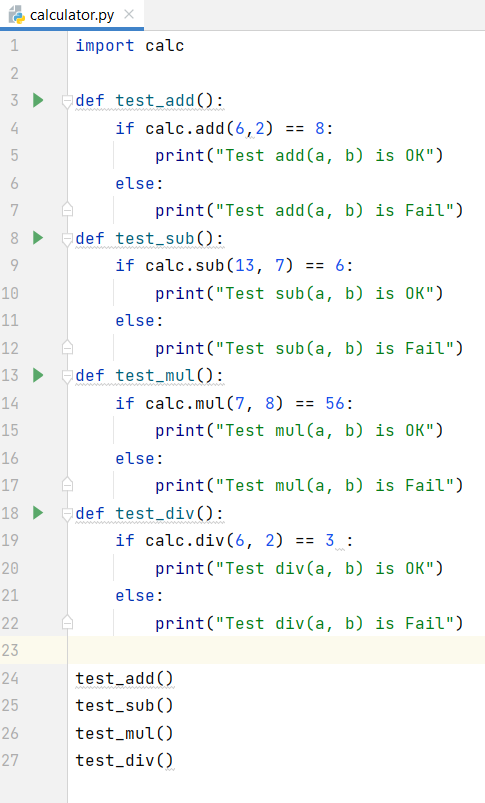
Символ «Е» – вызывает исключение, отличное от AssertionError

Задание 1.

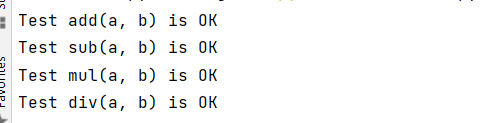
1. Создать файл calc.py и сохранить его.



1. Создать файл calculator.py и сохранить его.



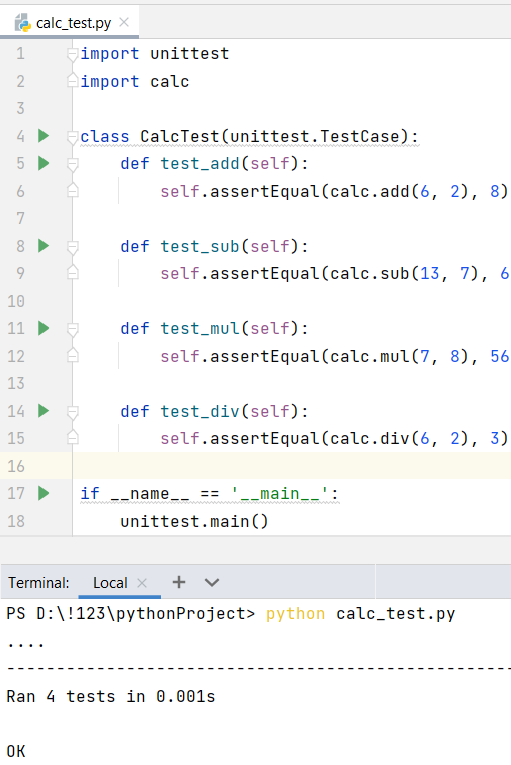
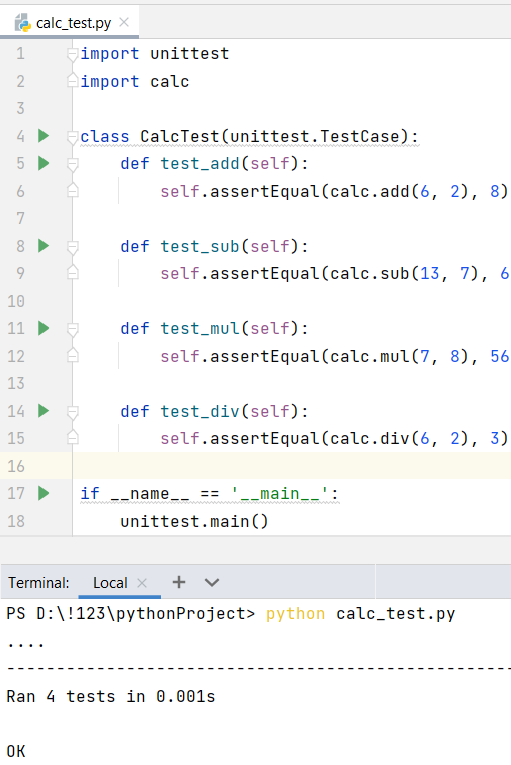
1. Запустить calculator.py в консоли. В результате выведется:



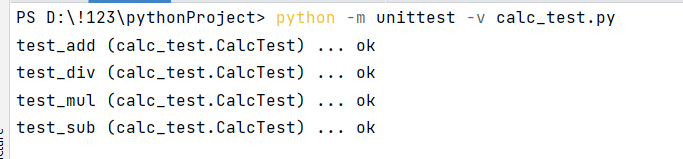
При написании тестов, как обычных программ, возникает ряд неудобств, в первую очередь связанных с унификацией выходной информации о пройденных и не пройденных тестах, сами тесты получаются довольно громоздкими, также необходимо продумывать архитектуру тестирующего приложения и т.д.

Изменить тесты можно с помощью фреймворка *unittest*.

1. Создать файл calc\_test.py. Запустить его в консоли.



Запуск можно сделать с запросом расширенной информации по пройденным тестам, для этого необходимо добавить ключ *-v*:



Для того, чтобы метод класса выполнялся как тест, необходимо, чтобы он начинался с префикса *test\_*.

**Основные структурные элементы** ***unittest***

Основными структурными элемента каркаса *unittest* [являются](https://docs.python.org/3/library/unittest.html):

*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* – 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*.

*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, которые 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.

Вся работа по написанию тестов заключается в том, что разрабатываюся отдельные тесты в рамках *test case’ов*, собираются в модули и запускаются. 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*.

Все 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 «приборки» за тестом.

Методы *setUp()* и *tearDown()* вызываются для всех тестов в рамках класса, в котором они переопределены. По умолчанию, эти методы ничего не делают. Если их добавить в *calc*\_*test.py*, то перед [после] тестов *test\_add()*, *test\_sub()*, *test\_mul()*, *test\_div()* будут выполнены *setUp()* [*tearDown()*].

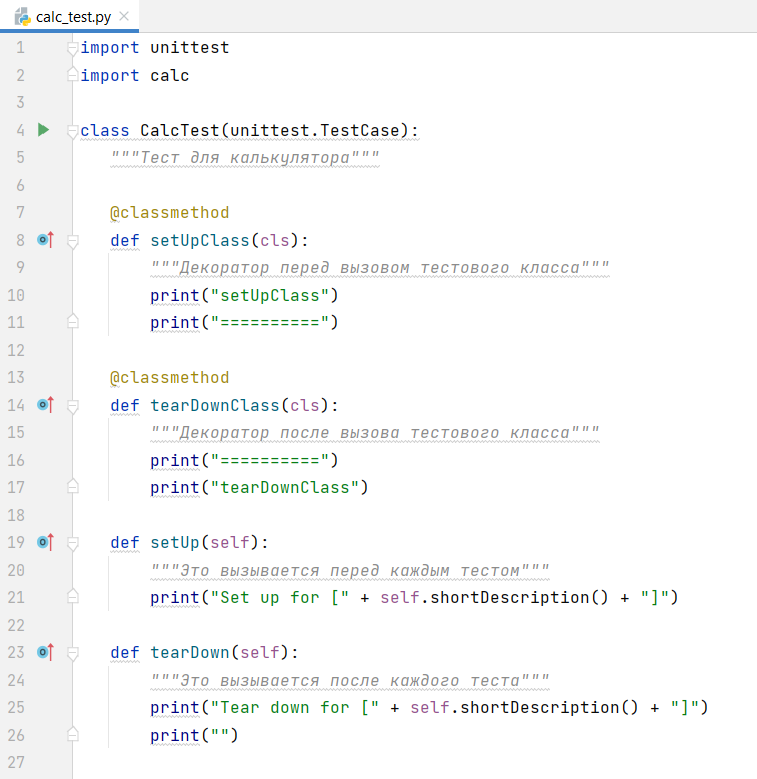
**setUpClass() –** метод действует на уровне класса, т.е. выполняется перед запуском тестов класса. При этом синтаксис требует наличие декоратора *@classmethod*.

**tearDownClass() –** запускается после выполнения всех методов класса, требует наличия декоратора *@classmethod*.

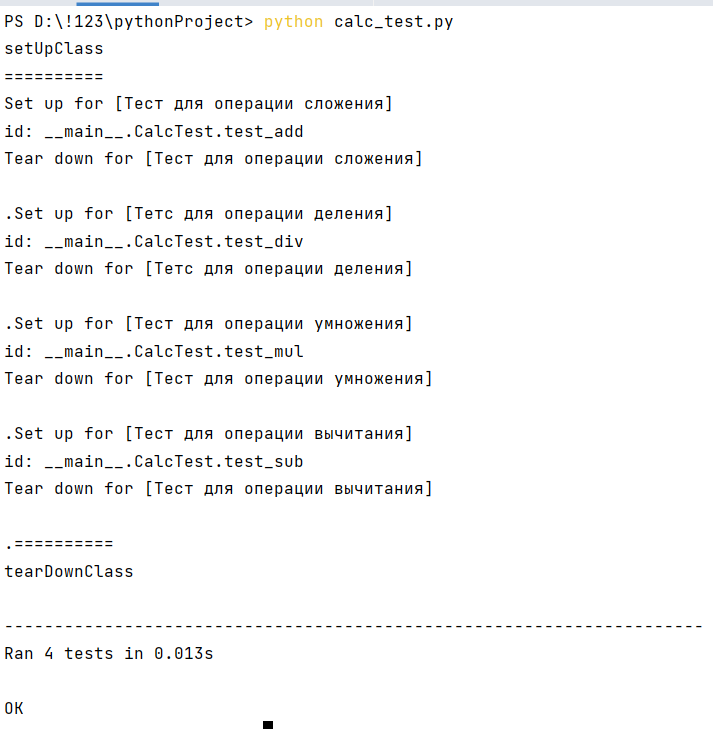
**skipTest(reason) –** данный метод может быть использован для пропуска теста, если это необходимо.

Задание 2.

Изменить код тестового проекта *calc\_test.py*, чтобы рассмотреть возможности, которые предоставляет класс *TestCase*.



При запуске в терминале, должен быть получен следующий результат:



**Класс *TestSuite***

Класс *TestSuite* используется для объединения тестов в группы, которые могут включать в себя как отдельные тесты так и заранее созданные группы. Помимо этого, *TestSuite* предоставляет интерфейс, позволяющий *TestRunner’у,* запускать тесты. Методы класса *TestSuite:*

**addTest(test) –** добавляет *TestCase* или *TestSuite* в группу.

**addTests(tests) –** добавляет все *TestCase* и *TestSuite* объекты в группу, итеративно проходя по элементам переменной *tests*.

**run(result) –** запускает тесты из данной группы.

**countTestCases() –** возвращает количество тестов в данной группе (включает в себя как отдельные тесты, так и подгруппы).

**Класс *TextTestRunner***

Объекты класса *TextTestRunner* используются для запуска тестов. Среди параметров, которые передаются конструктору класса, можно выделить *verbosity*, который определяет формат отображения результатов теста.

Есть 3 уровня:

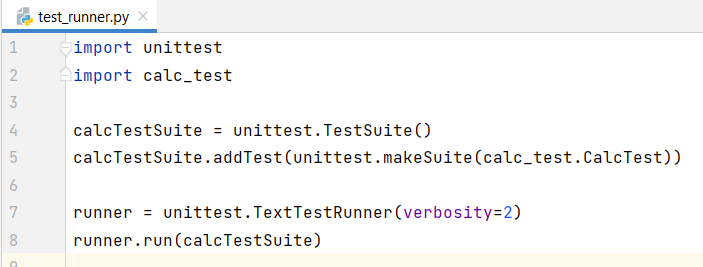
**0** : отображается общее количество выполненных тестов и глобальный результат

**1** **:** отображается то же самое плюс точка для каждого успешного теста, либо F для каждого отказа. Этот параметр используется по умолчанию

**2** : отображается строка справки каждого теста и результат

Задание 3.

1. Создать модуль *test\_runner.py* и добавить в него следующий код



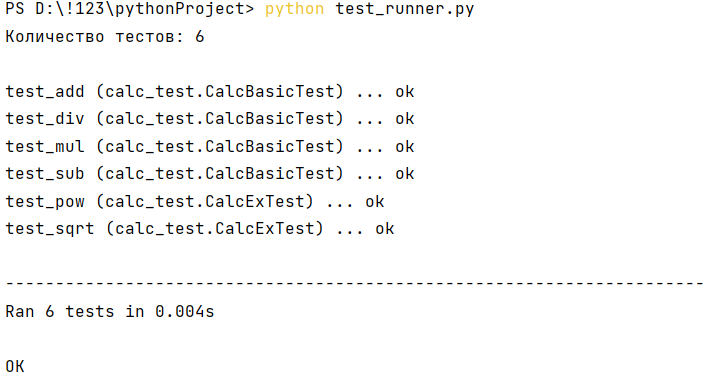
Запустить тест, используя команду:

>python test\_runner.py

1. Самостоятельно добавить еще два метода в модуль calc.py, в которых обеспечить выполнение возведения в степень (def pow(a,b)) и извлечение квадратного корня (def sqrt(a))
2. Добавить тесты для новых функций, создав новый класс с именем *CalcExTests* (расширенные функции калькулятора) с тестами для *sqrt()* и *pow()*, а класс *CalcTest* переименовать в *CalcBasicTests* (базовые функции калькулятора).
3. Изменить содержимое модуля *test\_runner.py*



При запуске в терминале будет получен результат:

В данном случае отсутствовали методы *Setup* и *TearDown*, а также декораторы класса. Если они будут присутствовать, то формат отображения результатов работы теста будут отличаться.

**Класс *TestLoader***

Этот класс используется для создания групп из классов и модулей. Среди методов *TestLoader* можно выделить:

**loadTestsFromTestCase(testCaseClass)** – возвращающий группу со всеми тестами из класса *testCaseClass*. Под тестом понимается модуль, начинающийся со слова *“test”*. Используя этот метод, можно создать список групп тестов, где каждая группа создается на базе классов-наследников от *TestCase*, объединенных предварительно в список.

import unittest

import calc\_tests

testCases = []

testCases.append(calc\_tests.CalcBasicTests)

testCases.append(calc\_tests.CalcExTests)

testLoad = unittest.TestLoader()

suites = []

for tc in testCases:

suites.append(testLoad.loadTestsFromTestCase(tc))

res\_suite = unittest.TestSuite(suites )

runner = unittest.TextTestRunner(verbosity=**2**)

runner.run(res\_suite)

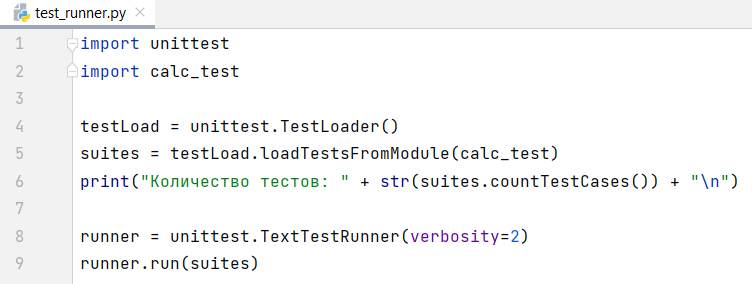
**loadTestsFromModule(module, pattern=None) –** загружает все тесты из модуля *module*. Если модуль поддерживает *load*\_*tests* протокол, то будет вызвана соответствующая функция модуля и ей будет передан в качестве аргумента (третьим по счету) параметр pattern.

**loadTestsFromName(name, module=None) –** загружает тесты в соответствии с параметром *name*. Параметр *name* – это имя, разделенное точками. С помощью этого имени указывается уровень, начиная с которого будут добавляться тесты.

**getTestCaseNames(testCaseClass) –** возвращает список имен методов-тестов из класса *testCaseClass*.

Задание 4.

1. Изменить модуль *test\_runner.py*



1. Запустить тест. Убедиться, что результат выполнения не отличается от предыдущего.
2. В модуле*test\_runner.py* заменить строку

suites = testLoad.loadTestsFromModule(calc\_tests)

на

suites = testLoad.loadTestsFromName(“calc\_tests.CalcBasicTests”)

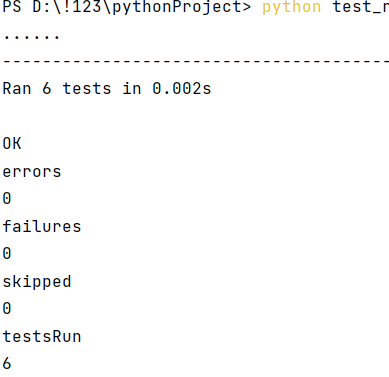
то будут выполнены только тесты из класса *CalcBasicTests*.

**Класс *TestResult***

Класс *TestResult* используется для сбора информации о результатах прохождения тестов.

Задание 5.

1. Изменить содержимое модуля *test\_runner.py*. Запустить и проанализировать результаты выполнения.



**Пропуск отдельных тестов в классе**

**unittest** предоставляет инструменты для удобного управление процессом пропуска тестов. Это может быть ещё полезно в том плане, что информацию о пропущенных тестах (их количестве) можно дополнительно получить через специальный *API*, предоставляемый классом *[TestResult](https://devpractice.ru/unit-testing-in-python-part-3/" \l "p22" \t "_blank)*.

Для пропуска теста используют декораторы, которые пишется перед тестом.

**@unittest.skip(reason)**

Для условного пропуска тестов применяются следующие декораторы:

**@unittest.skipIf(condition, reason)**

Тест будет пропущен, если условие (*condition*) истинно.

**@unittest.skipUnless(condition, reason)**

Тест будет пропущен если, условие (*condition*) не истинно.

Для пропуска классов используется декоратор

**@unittest.skip(reason)**

который записывается перед объявлением класса. В результате все тесты из данного класса не будут выполнены.

Задание 6.

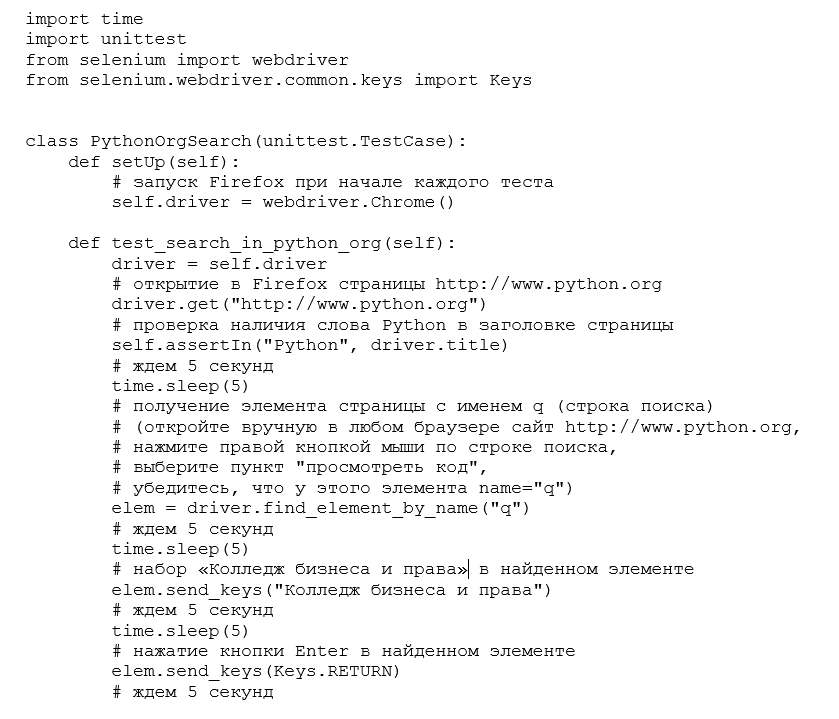
1. Добавить декоратор *@unittest.skip("Временно пропуск теста test\_add")* для теста *test\_add* в файле *calc\_test.py*
2. Запустить *test\_runner.py* и проверить, что тест пропущен.

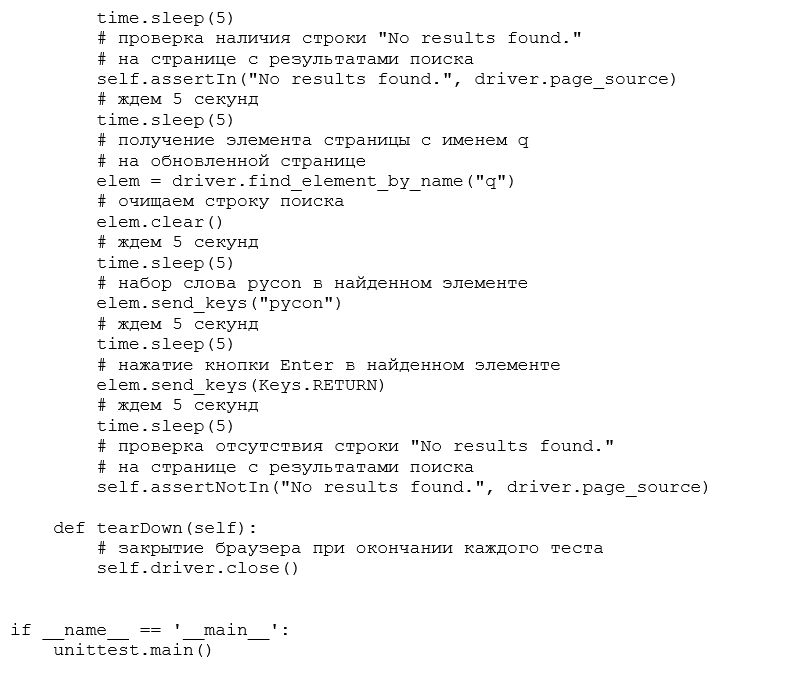
test\_add (calc\_test.CalcBasicTest) ... skipped 'Временно пропуск теста test\_add'

1. Исключить из процесса тестирования методы  *sqrt* и *pow.* Для этого поместить декоратор *skip* перед объявлением класса *CalcExTests* в файле *calc\_test.py.*
2. Запустить *test\_runner.py* и проверить, что тесты пропущен.

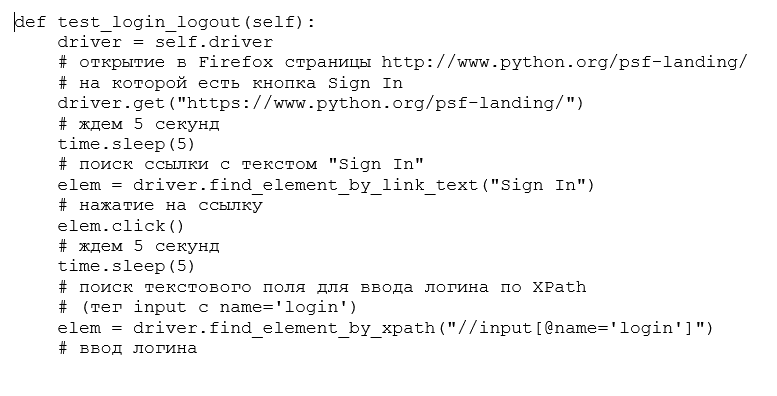
Задание 7.

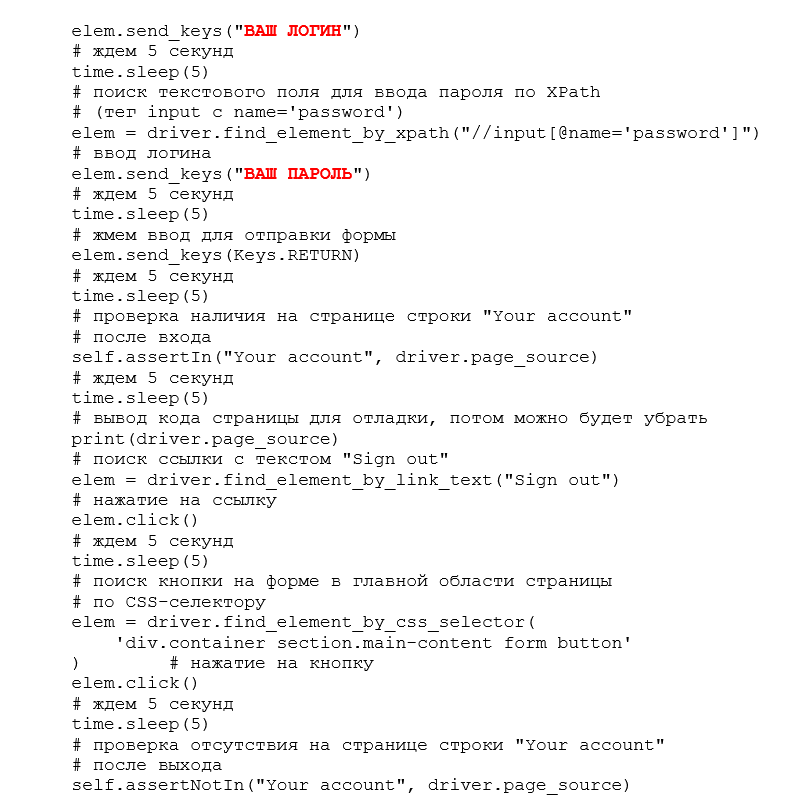
1. Создайте новый проект типа «Pure Python», установите пакет selenium.
2. Будем тестировать сайт python.org. Зайдите на этот сайт, нажмите правой кнопкой мыши по строке поиска, выберите пункт "просмотреть код", убедитесь, что у этого элемента name="q".
3. Введите в строку поиска что-нибудь, например, «Колледж бизнеса и права», убедитесь в появлении сообщения об отсутствии результатов на данном сайте, затем введите что-нибудь по теме сайта, например, «pycon», убедитесь в появлении результатов поиска.
4. Создайте Python unit test для проверки работы Selenium, драйвера Chrome на проделанном вручную сценарии:



**

1. Запустите тест, убедитесь в его успешном выполнении. Учитывайте, что в коде есть задержки по 5 секунд для наглядности выполнения каждого шага в тесте.
2. Зайдите на страницу https://www.python.org/psf-landing/, зарегистрируйтесь **не через социальные сети** (“Sign In” – “Sign Up / Register”), перейдите по ссылке, пришедшей по почте, нажмите кнопку “Confirm”.
3. Войдите на сайт (“Sign In” – “Sign In”), убедитесь в появлении кнопки “Your account”, обязательно **осуществите выход** **с сайта** (“Your account” – “Sign Out”, кнопка “ Sign Out ”).
4. Протестируйте вход и выход автоматически с помощью нового теста внутри существующего класса PythonOrgSearch:





1. Запустите тест, убедитесь в возникновении ошибки при обнаружении ссылки "Sign out", нажмите на окно «Run», нажмите **Ctrl + F** для поиска, введите "Sign out", убедитесь, что в верстке страницы такая ссылка есть, но у ее родительского элемента некорректные атрибуты:

<li class="tier-2 element-5" **"=""** role="treeitem&quot;">

1. Таким образом, составленный тест обнаружил проблему. Для продолжения тестирования до ее исправления замените поиск и нажатие ссылки Sign out на переход по соответствующему адресу:

# заменить

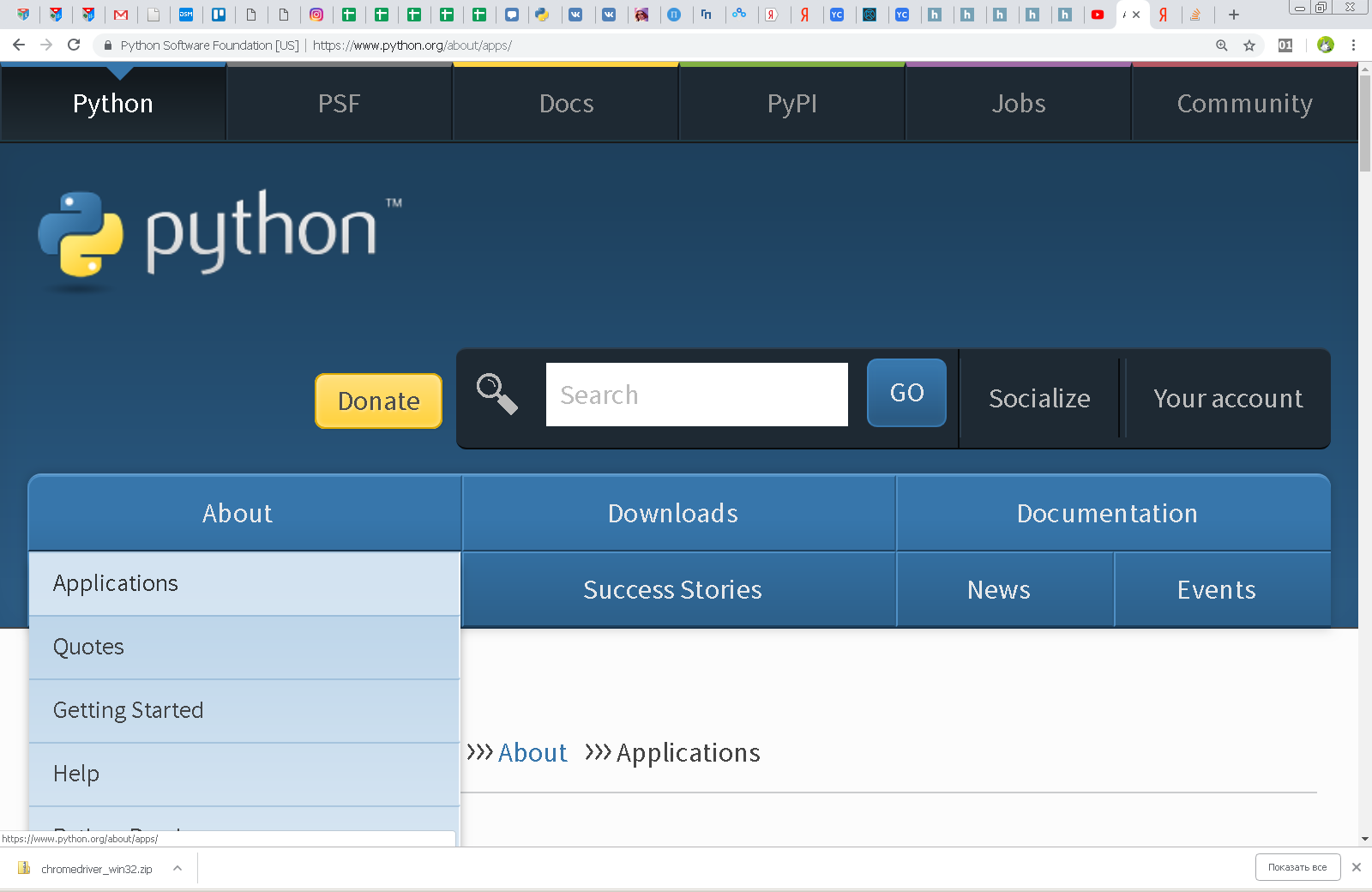
# elem = driver.find\_element\_by\_link\_text("Sign out")

# elem.click()

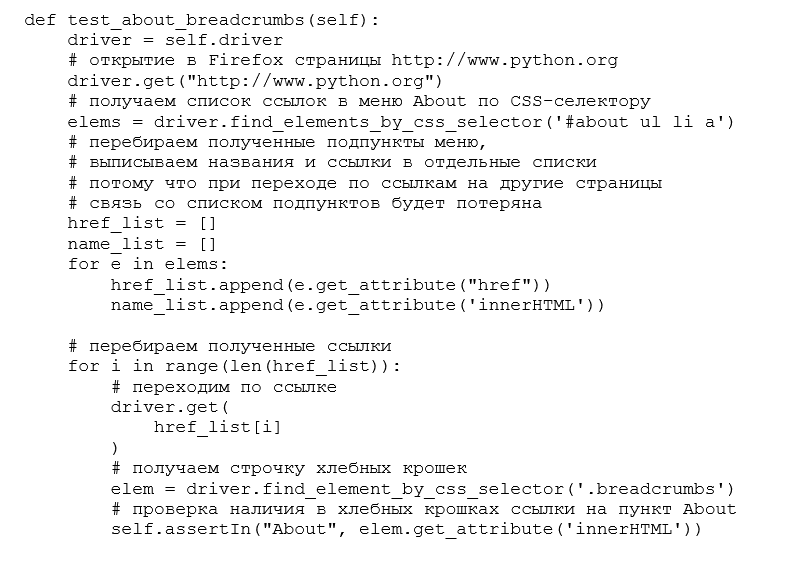
# на

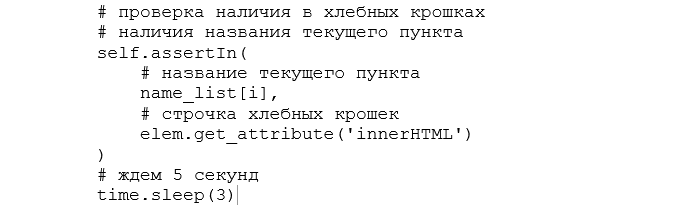
driver.get("https://www.python.org/accounts/logout/")

1. Запустите оба теста, убедитесь в их успешном выполнении.
2. Протестируем отображение «хлебных крошек» на страницах из выпадающего списка About – строчки, отображающей путь от корня сайта до текущей страницы через иерархию промежуточных пунктов:



1. Допишем новый тест для хлебных крошек в существующий класс PythonOrgSearch:





1. Запустите тест, убедитесь, что последний пункт Python Brochure не загружается. Проблема выявлена, для продолжения теста до ее устранения будем выполнять цикл по всем пунктам, кроме этого последнего – уменьшим число итераций во втором цикле:

# заменить

# for i in range(len(href\_list)):

# на

for i in range(len(href\_list)-1):

1. **Самостоятельно добавить два-три теста, осуществляющих проверку текста на странице, проверку результатов поиска, наличие ссылки на странице или самостоятельно продумать проверку.**

**При написании тестов использовать методы, позволяющие собирать информацию о самом тесте.**

1. Порядок выполнения работы:
2. Изучить предлагаемый теоретический материал.
3. Выполнить задания, приведенные в лабораторной работе.
4. Выполнить индивидуальное задание.
5. Подготовить отчет по практической работе, в отчете дать ответы на контрольные вопросы.
6. Контрольные вопросы:
7. Что представляет собой unittest?
8. Описать основные структурные элементы unittest.
9. Какие методы используются при запуске тестов?
10. Для чего используется класс TestResult?
11. Каким образом организовать пропуск отдельных тестов, пропуск классов?
12. ЛИТЕРАТУРА

[unittest](https://digitology.tech/docs/python_3/library/unittest.html#module-unittest) – Фреймворк модульного тестирования. [Электронный ресурс] – Режим доступа: https://digitology.tech/docs/python\_3/library/unittest.html#module-unittest