**Тема 4.3. Использование модулей Unittest и Pytest для автоматизации тестов**

**Вопросы**:

1. Фреймворк Unittest на Python для проведения автоматизированного тестирования.
2. Фреймворк Pytest на Python для проведения автоматизированного тестирования

**Фреймворк Unittest на Python для проведения автоматизированного тестирования**

Для 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.

**PyTest** и **nose** устанавливаются дополнительно, они позволяют получить расширенные возможности по сравнению с **unittest**.

Тест-раннеры сами находят тестовые методы в указанных при запуске файлах, но для этого нужно следовать общепринятым правилам. Общее правило для всех фреймворков: *название тестового метода должно начинаться со слова "****test\_****"*.  Дальше может идти любой текст, который является уникальным названием для теста:

def test\_name\_for\_your\_test():

Для **unittest** существуют собственные дополнительные правила:

1. Тесты обязательно должны находиться в специальном тестовом классе.
2. Вместо **assert** должны использоваться специальные assertion методы.

Для запуска тестов с помощью **unittest** понадобится выполнить следующие шаги:

1. Импортировать unittest в файл:

**import unittest**

1. Создать класс, который должен наследоваться от класса TestCase:

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

1. Превратить тестовые функции в методы, добавив ссылку на экземпляр класса self в качестве первого аргумента функции:

**def test\_abs1(self):**

1. Изменить assert на **self.assertEqual()**
2. Заменить строку запуска программы на **unittest.main()**

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

**Test fixture**

*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**

*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**

*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**

*Test runner* – это компонент, которые 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.

Вся работа по написанию тестов заключается в том, что разрабатываются 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*.

Работа с ***TestCase***

Основным элементом при написании тестов с использованием *unittest* является *TestCase*. Он представляет собой класс, который должен являться базовым для всех остальных классов, методы которых будут тестировать те или иные автономные единицы исходной программы.

import unittest

import calc

*class* CalcTests(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()

Для того, чтобы была возможность использовать компоненты unittest (в том числе и TestCase), в самом начале программы нужно импортировать модуль unittest стандартным образом.

При выборе имени класса наследника от TestCase можно руководствоваться следующим правилом: [ИмяТестируемойСущности]Tests. [ИмяТестируемойСущности] – это некоторая логическая единица, тесты для которой нужно написать.

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.

***setUpClass()***

Метод действует на уровне класса, т.е. выполняется перед запуском тестов класса. При этом синтаксис требует наличие декоратора *@classmethod.*

*@classmethod*

*def* setUpClass(cls):

...

***tearDownClass()***

Запускается после выполнения всех методов класса, требует наличия декоратора *@classmethod*.

*@classmethod*

*def* tearDownClass(cls):

...

***skipTest(reason)***

Данный метод может быть использован для пропуска теста, если это необходимо.

**Методы, используемые при непосредственном написании тестов (проверка условий, сообщение об ошибках).**

*TestCase* класс предоставляет набор *assert*-методов для проверки и генерации ошибок:

|  |  |
| --- | --- |
| [assertEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertEqual) | a == b |
| [assertNotEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertNotEqual) | a != b |
| [assertTrue(x)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertTrue) | bool(x) is True |
| [assertFalse(x)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertFalse) | bool(x) is False |
| [assertIs(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIs) | a is b |
| [assertIsNot(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIsNot) | a is not b |
| [assertIsNone(x)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIsNone) | x is None |
| [assertIsNotNone(x)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIsNotNone) | x is not None |
| [assertIn(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIn) | a in b |
| [assertNotIn(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertNotIn) | a not in b |
| [assertIsInstance(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertIsInstance) | isinstance(a, b) |
| [assertNotIsInstance(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertNotIsInstance) | not isinstance(a, b) |

*Assert’ы* для контроля выбрасываемых исключений и warning’ов:

|  |  |
| --- | --- |
| [assertRaises(exc, fun, \*args, \*\*kwds)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaises) | Функция fun(\*args, \*\*kwds) вызывает исключение exc |
| [assertRaisesRegex(exc, r, fun, \*args, \*\*kwds)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRaisesRegex) | Функция fun(\*args, \*\*kwds) вызывает исключение exc, сообщение которого совпадает с регулярным выражением r |
| [assertWarns(warn, fun, \*args, \*\*kwds)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertWarns) | Функция fun(\*args, \*\*kwds) выдает сообщение warn |
| [assertWarnsRegex(warn, r, fun, \*args, \*\*kwds)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertWarnsRegex) | Функция fun(\*args, \*\*kwds) выдает сообщение warn и оно совпадает с регулярным выражением r |

*Assert’ы* для проверки различных ситуаций:

|  |  |
| --- | --- |
| [assertAlmostEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertAlmostEqual) | round(a-b, 7) == 0 |
| [assertNotAlmostEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertNotAlmostEqual) | round(a-b, 7) != 0 |
| [assertGreater(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertGreater) | a > b |
| [assertGreaterEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertGreaterEqual) | a >= b |
| [assertLess(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertLess) | a < b |
| [assertLessEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertLessEqual) | a <= b |
| [assertRegex(s, r)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertRegex) | r.search(s) |
| [assertNotRegex(s, r)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertNotRegex) | not r.search(s) |
| [assertCountEqual(a, b)](https://docs.python.org/3/library/unittest.html#unittest.TestCase.assertCountEqual) | a и b имеют одинаковые элементы (порядок неважен) |

Дополнительно хотелось бы отметить метод *fail().*

**fail(msg=None)**

Этот метод сигнализирует о том, что произошла ошибка в тесте.

**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.*

**Фреймворк Pytest на Python для проведения автоматизированного тестирования**

Преимущества использования **PyTest**:

1)PyTest полностью обратно совместим с фреймворками **unittest**. Это означает, что если изначально писали тесты, используя unittest, то перейти на **PyTest** можно буквально в ту же минуту. Для этого в виртуальном окружении должен быть установлен пакет **PyTest**.

*pip install pytest*

Теперь можно запустить тесты с помощью **PyTest**, не изменяя сам файл. **PyTest** сам найдёт тесты в папке, в которой вы их запускаете, и выполнит их:

*pytest test\_abs\_project.py*

2) Подробный отчёт с поддержкой цветовых схем из коробки.

3) **PyTest** не требует написания дополнительных специфических конструкций в тестах, как того требует **unittest** (no boilerplate).

4) Для проверок используется стандартный **assert** из **Python**.

5) Возможность создания динамических фикстур (специальных функций, которые настраивают тестовые окружения и готовят тестовые данные).

6) Дополнительные возможности по настройке фикстур.

7) Параметризация тестов – для одного теста можно задать разные параметры (тест запустится несколько раз с разными тестовыми данными).

8) Наличие маркировок (**marks**), которые позволяют маркировать тесты для их выборочного запуска.

9) Возможность передавать дополнительные параметры через командную строку для настройки тестовых окружений.

10) Большое количество плагинов, которые расширяют возможности **PyTest** и позволяют решать узкоспециализированные проблемы, что может сэкономить много времени.

Рассмотрим минусы **PyTest**:

1) **PyTest** требуется устанавливать дополнительно, так как он не входит в стандартный пакет библиотек **Python**, в отличие от **unittest**. Нужно не забывать об этом, когда вы будете настраивать автоматический запуск тестов с помощью CI-сервера.

2) Использование **PyTest** требует более глубокого понимания языка **Python**, чтобы разобраться, как применять фикстуры, параметризацию и другие возможности **PyTest**.

**Запуск тестов с помощью PyTest**

Когда выполняют команду **pytest**, тест-раннер собирает все тесты для запуска по определенным правилам:

* тест-раннер начнёт поиск в текущей директории и 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)

**PyTest – проверка ожидаемого результата (assert)**

Для проверки ожидаемых результатов в **unittest** тестах 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.

**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, можно использовать специальную конструкцию **with pytest.raises()**.

**Классические фикстуры (fixtures)**

Важной составляющей в использовании **PyTest** является концепция фикстур. Фикстуры в контексте **PyTest** — это вспомогательные функции для наших тестов, которые не являются частью тестового сценария.

Назначение фикстур может быть самым разным. Одно из распространенных применений фикстур – это подготовка тестового окружения и очистка тестового окружения и данных после завершения теста. Фикстуры можно использовать для самых разных целей: для подключения к базе данных, с которой работают тесты, создания тестовых файлов или подготовки данных в текущем окружении с помощью API-методов.

Классический способ работы с фикстурами – создание setup- и teardown-методов в файле с тестами

from selenium import webdriver

link = http://test123.com/

class TestMainPage():

def setup\_method(self):

print("start browser for test..")

self.browser = webdriver.Chrome()

def teardown\_method(self):

print("quit browser for test..")

self.browser.quit()

def test\_guest\_should\_see\_login\_link(self):

self.browser.get(link)

self.browser.find\_element\_by\_css\_selector("#login\_link")

PyTest предлагает продвинутый подход к фикстурам, когда фикстуры можно задавать глобально, передавать их в тестовые методы как параметры, а также имеет набор встроенных фикстур. Это более гибкий и удобный способ работы со вспомогательными функциями.

**Возвращаемое значение**

Фикстуры могут возвращать значение, которое затем можно использовать в тестах. Давайте перепишем предыдущий пример с использованием **PyTest** фикстур. Для этого создадим фикстуру **browser**, которая будет создавать объект WebDriver. Этот объект сможем использовать в тестах для взаимодействия с браузером. Для этого напишем метод browser и укажем, что он является фикстурой с помощью декоратора **@pytest.fixture**. После этого можем вызывать фикстуру в тестах, передав ее как параметр. По умолчанию фикстура будет создаваться для каждого тестового метода, то есть для каждого теста запустится свой экземпляр браузера.

import pytest from selenium

import webdriver

link = http://test123.com/

@pytest.fixture

def browser():

print("\nstart browser for test..")

browser = webdriver.Chrome()

return browser

class TestMainPage1(): # вызываем фикстуру в тесте, передав ее как параметр

def test\_guest\_should\_see\_login\_link(self, browser):

browser.get(link)

browser.find\_element\_by\_css\_selector("#login\_link")

**Финализаторы – закрываем браузер**

В этом примере не использована команду browser.quit(). Это привело к тому, что несколько окон браузера оставались открыты после окончания тестов, а закрылись только после завершения всех тестов. Надо явно закрывать браузеры после каждого теста, для этого надо воспользоваться финализатором Python: **yield**. После завершения теста, который вызывал фикстуру, выполнение фикстуры продолжится со строки, следующей за строкой со словом **yield**:

import pytest from selenium

import webdriver

link = http://test123.com/

@pytest.fixture

def browser():

print("\nstart browser for test..")

browser = webdriver.Chrome()

yield browser # этот код выполнится после завершения теста

print("\nquit browser..")

browser.quit()

class TestMainPage1(): # вызываем фикстуру в тесте, передав ее как параметр

def test\_guest\_should\_see\_login\_link(self, browser):

browser.get(link)

browser.find\_element\_by\_css\_selector("#login\_link")

Для фикстур можно задавать область покрытия фикстур. Допустимые значения: “**function**”, “**class**”, “**module**”, “**session**”. Соответственно, фикстура будет вызываться один раз для тестового метода, один раз для класса, один раз для модуля или один раз для всех тестов, запущенных в данной сессии.

Запустим все тесты из класса **TestMainPage** в одном браузере для экономии времени, задав scope="class" в фикстуре browser:

import pytest

from selenium import webdriver

link = http://test123.com/

@pytest.fixture(scope="class")

def browser():

print("\nstart browser for test..")

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

class TestMainPage1():

# вызываем фикстуру в тесте, передав ее как параметр

def test\_guest\_should\_see\_login\_link(self, browser):

print("start test1")

browser.get(link)

browser.find\_element\_by\_css\_selector("#login\_link")

print("finish test1")

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

print("start test2")

browser.get(link)

browser.find\_element\_by\_css\_selector(".basket-mini .btn-group > a")

print("finish test2")

В данном примере браузер открылся один раз и тесты последовательно выполнились в этом браузере.

Для запуска используется параметр **-s**, чтобы увидеть текст, который выводится командой print().

pytest -s test\_fixture.py

**Маркировка тестов**

Когда тестов становится много, хорошо иметь способ разделять тесты не только по названиям, но также по каким-нибудь заданным нами категориям. Например, можем выбрать небольшое количество критичных тестов (smoke), которые нужно запускать на каждый коммит разработчиков, а остальные тесты обозначить как регрессионные (regression) и запускать их только перед релизом. Или могут быть тесты, специфичные для конкретного браузера (internet explorer 11), и хотим запускать эти тесты только под данный браузер. Для выборочного запуска таких тестов в **PyTest** используется маркировка тестов или **метки (marks)**. Для маркировки теста нужно написать декоратор вида **@pytest.mark.mark\_name**, где mark\_name — произвольная строка.

class TestMainPage1():

@pytest.mark.smoke

def test\_guest\_should\_see\_login\_link(self, browser):

browser.get(link)

browser.find\_element\_by\_css\_selector("#login\_link")

@pytest.mark.regression

def test\_guest\_should\_see\_basket\_link\_on\_the\_main\_page(self, browser):

browser.get(link)

browser.find\_element\_by\_css\_selector(".basket-mini .btn-group > a")

Чтобы запустить тест с нужной маркировкой, нужно передать в командной строке параметр -m и нужную метку:

pytest -s -v -m smoke test\_fixture.py

Если всё сделано правильно, то должен запуститься только тест с маркировкой smoke.

При этом вы увидите warning, то есть предупреждение:

PytestUnknownMarkWarning: Unknown pytest.mark.smoke - is this a typo? You can register custom marks to avoid this warning - for details, see https://docs.pytest.org/en/latest/mark.html

PytestUnknownMarkWarning,

Это предупреждение появилось потому, что в последних версиях PyTest настоятельно рекомендуется регистрировать метки явно перед использованием. Это, например, позволяет избегать опечаток, когда можете ошибочно пометить  тест несуществующей меткой, и он будет пропускаться при прогоне тестов.

**Как же регистрировать метки?**

Создайте файл pytest.ini в корневой директории вашего тестового проекта и добавьте в файл следующие строки:

[pytest]

markers =

smoke: marker for smoke tests

regression: marker for regression tests

Текст после знака ":" является поясняющим — его можно не писать.

Снова запустите тесты:

pytest -s -v -m smoke test\_fixture.py

Теперь предупреждений быть не должно.

**Conftest.py — конфигурация тестов**

Ранее добавили фикстуру browser, которая создает экземпляр браузера для тестов в данном файле. Когда файлов с тестами становится больше одного, приходится в каждом файле с тестами описывать данную фикстуру. Это очень неудобно. Для хранения часто употребимых фикстур и хранения глобальных настроек нужно использовать файл**conftest.py,** который должен лежать в директории верхнего уровня в проекте с тестами. Можно создавать дополнительные файлы conftest.py в других директориях, но тогда настройки в этих файлах будут применяться только к тестам в под-директориях.

Создадим файл **conftest.py** в корневом каталоге нашего тестового проекта и перенесем туда фикстуру **browser**.

**conftest.py:**

import pytest

from selenium import webdriver

@pytest.fixture(scope="function")

def browser():

print("\nstart browser for test..")

browser = webdriver.Chrome()

yield browser

print("\nquit browser..")

browser.quit()

Теперь, сколько бы файлов с тестами ни создали, у тестов будет доступ к фикстуре browser. Фикстура передается в тестовый метод в качестве аргумента. Таким образом можно удобно переиспользовать одни и те же вспомогательные функции в разных частях проекта.

**test\_conftest.py:**

link = http://test123.com/

def test\_guest\_should\_see\_login\_link(browser):

browser.get(link)

browser.find\_element\_by\_css\_selector("#login\_link")