Geekbrains

Факультет тестирования ПО

Автоматизация тестирования UI с использованием Selenium и Pytest

**Дипломный проект**

студента 1 года обучения

специальности “Тестирование ПО”

Конюхов А.С.

Руководитель

Шафигуллин И.К.

Радужный

2023

**Дипломный проект “Автоматизация тестирования UI с использованием Selenium и Pytest”**

**Содержание**

[**Введение 5**](#_o0o8d1yn7sb)

[**Глава 1. Теория тестирования 7**](#_k7otkrrlald)

[1.1 Что такое тестирование? 7](#_i1cz2v7uc96b)

[1.1.1 Определение тестирования ПО 7](#_am5i5sjfr31n)

[1.1.2 Кто такой тестировщик? 8](#_zdsda10ok02)

[1.2 Классификация тестирования 10](#_ja8wecsdh7pf)

[1.3. Тестирование UI 11](#_xufwncu0e7td)

[**Глава 2. Базовые знания web-технологий 13**](#_9lfmweifvtnp)

[2.1 Основы HTML 13](#_4ijs5tq8h7ob)

[2.1.1 Что такое HTML? 13](#_e8xpfxj5vwvu)

[2.1.2 HTML элемент 14](#_dxs5lbpcmc91)

[2.2 Основы CSS 16](#_kmce6om0d8wn)

[2.2.1 Так что же такое CSS? 16](#_lghxldlpxm5n)

[2.2.2 Анатомия набора правил CSS 17](#_mlnnlcmdp6f6)

[2.3 Основы JavaScript 18](#_xsc0kzumnswy)

[**Глава 3. Знакомство с системой контроля версий Git 20**](#_58by4zyusqxv)

[3.1 Что такое система контроля версий? 20](#_bgfwvmetn2j5)

[3.2. Файл README.md и разметка Markdown 21](#_6hjibu5npcfg)

[**Глава 4. Основы Python 23**](#_pk9bsm8klmlj)

[4.1 Знакомство с языком программирования Python 23](#_rnptsg8csuan)

[4.2 Базовые принципы ООП 25](#_mviikck6y7vv)

[**Глава 5. Знакомство с Selenium 27**](#_8qs3mx5h6tbo)

[5.1 Что такое Selenium? 27](#_y2pr7nfhzhtq)

[5.2 Настройка первого запуска браузера с помощью Selenium Webdriver 29](#_flelyck9lxuq)

[5.3 Поиск элементов 33](#_eal5zjp8xxsf)

[5.3.1 Поиск элементов с помощью CSS-селекторов 33](#_q0vs17ochf91)

[5.3.1.1 Поиск по id 35](#_aa7wyumhtwg2)

[5.3.1.2 Поиск по class 36](#_gjhxxuqgcb95)

[5.3.1.3 Поиск по tag 37](#_o4bl5ys6ahdx)

[5.3.1.4 Поиск по значению атрибута 38](#_s40kgnew8on0)

[5.3.1.5 Поиск элементов с помощью составных CSS-селекторов 39](#_xbu25kav8lr3)

[5.3.1.5.1 Использование порядкового номера дочернего элемента 39](#_llhrp27ms458)

[5.3.1.5.2 Использование дочерних элементов 40](#_fmgtm2g51ydz)

[5.3.1.5.3 Использование потомков 41](#_jwqft9c8f3wb)

[5.3.2 Поиск элементов с помощью XPath 42](#_7l8cshdu18xv)

[5.4 Поиск элементов с помощью Selenium WebDriver 44](#_4395n3xwjgyl)

[5.4.1 Поиск элементов с помощью Selenium 44](#_3xhkbcvc8nq3)

[5.4.2 Работа с браузером в Selenium 46](#_u92rjppkbppx)

[5.4.2.1 Поиск всех необходимых элементов. 47](#_b2h2vkhg0noc)

[**Глава 6. Полезные методы Selenium 49**](#_eagukfwknf0q)

[6.1 Как работать с элементами типа checkbox и radiobutton? 49](#_mhbise86w59b)

[6.2 Метод get\_attribute() 51](#_1ga3rbyfryki)

[6.3 Работа с файлами, списками и js-скриптами 54](#_w67thlxdhqax)

[6.3.1 Работа со списками 54](#_pz0mp4w9o8r2)

[6.3.2 Метод execute\_script() 56](#_hz5bymoxkc92)

[6.3.3 Загрузка файлов 58](#_9c3syho3trm)

[6.3.4 Работа с окнами 60](#_leino1pqpiyj)

[6.3.4.1 Alerts 60](#_x150y77205kh)

[6.3.4.2 Переход на новую вкладку в браузере 62](#_b8grkl97ckiv)

[6.4 Настройка ожиданий 63](#_v0hn1upeeln)

[6.4.1 Работа методов get() и find\_element() 63](#_ljf25a71ev2t)

[6.4.2. Использование time.sleep() 65](#_3bnc18xtzhgw)

[6.4.3 Использование Selenium Waits (Implicit Waits) 66](#_zgs0ygg5t5b)

[6.4.4 Использование Explicit Waits (WebDriverWait и expected\_conditions) 68](#_fcsdz8k11xaa)

[**Глава 7. Тестовые фреймворки 69**](#_bamwhb7cxdar)

[7.1 Тестирование web-приложений и тестовые фреймворки 69](#_y95yokvbq7m)

[7.1.1 Автоматизированное и ручное тестирование 69](#_1oetv3kvafbv)

[7.1.2 Структура автотеста 72](#_kmyuvn5l52gw)

[7.1.3 Проверка ожидаемого результата 73](#_ajokadaavu1)

[7.1.4 Тестовые сценарии 75](#_2f7bywh3ei4y)

[7.1.5 Выбор test runner 77](#_19njiymcq7mv)

[7.1.5.1 unittest 79](#_pi6heavrd7tg)

[7.2 Тестирование с помощью Pytest 82](#_fekzbappoxje)

[7.2.1 Pytest - преимущества и недостатки 82](#_x4olfohdo8gq)

[7.2.2 Фиксация пакетов 84](#_1mdoza2gz1qd)

[7.2.3 Pytest: правила запуска тестов 86](#_rtmlcmwu0i29)

[7.2.4 Pytest - отчёты 88](#_4qwxxp2k6twq)

[7.2.5 Pytest - как пишут тесты. 90](#_xve276ybz9i8)

[7.3 Использование фикстур в Pytest 91](#_gmt8l5kxx0rp)

[7.3.1 Классические фикстуры 91](#_9cxadnwszjoz)

[7.3.2 Фикстуры, возвращающие значение 94](#_kcq96jj0kxz2)

[7.3.3 Финализаторы, закрываем браузер 96](#_tg1478dbv9q9)

[7.3.4 Область видимости 98](#_dy212aikmei)

[7.3.5 Автоиспользование фикстур 100](#_b3pr4u2934mb)

[7.4 Pytest - маркировка 101](#_7kf0rlxwqya3)

[7.4.1 Маркировка тестов часть первая 101](#_8f1xogog6siq)

[7.4.2 Как регистрировать метки? 103](#_jl1su4w7fmi1)

[7.4.3 Маркировка тестов часть вторая 104](#_vzyplr9bi039)

[7.4.3.1 Инверсия 104](#_av59q6arxrva)

[7.4.3.2 Объединение тестов с разной маркировкой 105](#_y6n7pvz3ou81)

[7.4.3.3 Запуск тестов, имеющих несколько маркировок 106](#_afbnm8a5bc0t)

[7.4.4 Пропуск тестов 108](#_un9nru2gad3e)

[7.4.5 XFail: отметить тест как ожидаемо падающий 110](#_9bwa4himf6tp)

[7.5 Pytest - параметризация, конфигурирование, плагины 112](#_xplk8cts9sh)

[7.5.1 Conftest.py - конфигурирование тестов 112](#_qi2jlejs0xfs)

[7.5.2 Параметризация тестов 115](#_cbvc3jjuor28)

[7.5.3 Установка Firefox и Selenium-драйвера geckodriver 117](#_769uxacahugw)

[7.5.4 Conftest.py и передача параметров в командной строке 118](#_59sp2k4t0fwj)

[7.5.5 Плагины и перезапуск тестов 120](#_13j8i95vob9v)

[**Глава 8. Использование Page Object Model 122**](#_67t2eslbz0nf)

[8.1 Что такое Page Object? 122](#_hi5l1ob28l8v)

[8.1.1 Code Style 122](#_ieyp80fi8vpq)

[8.1.1.1 Отступы 123](#_xspfjtwy8z)

[8.1.1.2 Имена переменных и финкций 124](#_a3qfhcla4zc0)

[8.1.1.3 Максимальная простота кода 126](#_k0uzn1m3fxs1)

[8.1.2 Code Style в автотестах 127](#_cibfgtl8srqn)

[8.1.3 Подготовка окружения 130](#_sxcvykhrmkz6)

[8.1.4 Что такое Page Object Model? 132](#_d231j7b29ay)

[8.1.5 Почему именно Page Object? 134](#_ua7nwisztaue)

[8.2 Первые тесты с применением Page Model 136](#_vydtjztfmq9m)

[8.2.1 Базовая страница для проекта: BasePage 136](#_9yk6lt3gxpvk)

[8.2.2 Page Object для главной страницы сайта 138](#_4js7tsu1alus)

[8.2.3 Первый тест на основе Page Object 140](#_p8f4gaaog9qo)

[8.2.4 Методы-проверки в Page Object 141](#_rodqrpaka5za)

[8.2.5 Проверка элемента на странице 143](#_uav9vl6v7fgc)

[8.2.6 Реализация LoginPage 147](#_ym7dh71djg03)

[8.2.7 Переходы между страницами 149](#_hcderrxmqyjr)

[8.2.7 Удобство поддержки тестов — инкапсуляция бизнес-логики в методах 150](#_gar3y8e5zt3p)

[8.3 Улучшаем дизайн тестов 151](#_sdzvg6fmbo2v)

[8.3.1 Задание: добавление в корзину со страницы товара 152](#_q0crk49od8w9)

[8.3.2 Задание: независимость от данных 154](#_yk406n6vrbem)

[8.3.3 Задание: независимость контента, ищем баг 157](#_hh2g3o4vh6lh)

[8.3.4 Отрицательные проверки: как проверить отсутствие элемента 159](#_ningsvx8t06w)

[8.3.4.1 Почему нужно писать такие проверки с осторожностью? 160](#_tpakczcxvymb)

[8.3.4.2 Как же тогда реализовывать такие проверки? 161](#_2wtoph6i376e)

[8.3.5 Задание: отрицательные проверки 163](#_fkcfbk81hdyc)

[8.3.6 Плюсы наследования: магия ООП 164](#_ax2b122wnh9l)

[8.3.7 Плюсы наследования: пример 165](#_reghfutootbk)

[8.3.8 Задание: наследование и отрицательные проверки 167](#_1q4uk6ck50y3)

[8.3.9 Группировка тестов: setup 168](#_t1clulf099bl)

[8.3.10 Задание: группировка тестов и setup 170](#_heclfij43oui)

[8.4 Создание отчётов о тестировании с помощью фреймворка Allure 171](#_9pdxrvp2oh5g)

[8.4.1 Что такое отчёт о тестировании и зачем он нужен? 171](#_p6azonekeujh)

[8.4.2 Стоит ли использовать инструменты формирования отчётов? 172](#_dtqzn1zo55j)

[8.4.3 Использование Allure 174](#_34c09m2fwtpf)

[8.4.3.1 Установка зависимостей 174](#_539htk11zq5)

[8.4.3.2 Получение скриншотов экрана 177](#_43ftc8hvzbjs)

[8.4.3.3 Готовый отчёт о тестировании 178](#_xygiw0iyrcjf)

[**Библиографический список литературы 179**](#_xjv3y4q1ubnu)

# Введение

Актуальность

Автоматизация проникает во все сферы деятельности человека, избавляет от рутин и тестирование ПО не исключение. Автоматизация позволяет разгрузить тестировщика от рутинных проверок во время ретеста и регрессивного тестирования. А также для повышения качества разрабатываемого продукта путем встраивания в цикл непрерывной интеграции и непрерывной доставки. Актуальность темы дипломной работы связана со значительным распространением автоматизации тестирования и языка программирования Python и заключается в необходимости разработки рекомендаций по изучению данного направления для начинающих и практикующих специалистов по обеспечению качества.

Цели и задачи

Цель дипломной работы является разработка дорожной карты развития начинающего специалиста по обеспечению качества разработки ПО. Познакомить с инструментами и сформировать компетенции для успешного развития в направлении автоматизации тестирования на языке программирования Python.

Задачами дипломной работы в связи с указанной целью являются:

* изучить основные понятия теории тестирования;
* исследовать наиболее распространенные web-технологии;
* изучить систему контроля версий Git;
* изучить базовые возможности языка программирования Python;
* изучить основные возможности и методы Selenium;
* исследовать расширение возможностей тестирования с помощью фреймворков;
* раскрыть необходимость использования паттерна Page Object Model;
* изучить различные методы формирования отчетов о тестировании.

Объект и предмет исследования

Объектом дипломной работы является изучение начинающим специалистом автоматизации на языке Python для тестирования пользовательского интерфейса сайтов, и способы ускорения профессионального развития в данном направлении.

Предметом дипломной работы являются открытые источники в сети интернет, книги и курсы посвященные тестированию ПО.

Научная и практическая значимость

Научная новизна заключается в нахождении нового подхода к обучению специалистов в направлении автоматизации на языке программирования Python.

Практическая значимость заключается в снижении времени, необходимого для получения навыков, необходимых для использования в автоматизации тестирования UI web-приложений.

Инструменты и технологии

В дипломном проекте будут использованы следующие инструменты: DevTools, PyCharm, Python, Selenium WebDriver, Pytest, Allure.

Будут использованы следующие технологии: ChromeDriver, geckodriver, pytest-html, pytest-rerunfailures, allure-pytest, Page Object Model.

Рекомендации по изучению

Дипломный проект представляет из себя курс по освоению (или расширению уже имеющихся) знаний в области автоматизации тестирования на языке программирования Python.

Рекомендуется проходить курс последовательно и самостоятельно изучать дополнительные материалы. Ссылки на некоторые материалы будут приведены по ходу курса, но рекомендуется проводить самостоятельную работу по поиску актуальной информации.

# Глава 1. Теория тестирования

## 1.1 Что такое тестирование?

### 1.1.1 Определение тестирования ПО

Чтобы погрузиться в автоматизацию тестирования необходимо разобраться с тем что такое тестирование, для чего оно необходимо и как оно осуществляется.

Есть несколько разных трактовок о том, чем же является тестирование ПО, рассмотрим некоторые из них и вычленим общие черты тестирования.

Тестирование программного обеспечения - процесс исследования, испытания программного продукта, имеющий своей целью проверку соответствия между реальным поведением программы и её ожидаемым поведением на конечном наборе тестов, выбранных определенным образом.[[1]](#footnote-0)

Святослав Куликов дает такое определение тестирования ПО в своей книге «Тестирование программного обеспечения. Базовый курс.»:

Тестирование программного обеспечения — процесс анализа программного средства и сопутствующей документации с целью выявления дефектов и повышения качества продукта.[[2]](#footnote-1)

Международный совет по тестированию программного обеспечения (ISTQB, International Software Testing Qualifications Board) содержит только определение тестирования:

The process consisting of all lifecycle activities, both static and dynamic, concerned with planning, preparation and evaluation of a component or system and related work products to determine that they satisfy specified requirements, to demonstrate that they are fit for purpose and to detect defects.[[3]](#footnote-2)

Итак мы узнали, что тестирование - это поиск дефектов (багов) с целью повышения качества продукта. Как же эти баги обнаруживают?

1. Используем источники ожидаемого поведения системы (тестовые оракулы) для понимания как должна работать система.

2. Производим манипуляции с системой, чтобы она проявила свое истинное поведение.

3. Сравниваем ожидаемый результат с фактическим.

### 1.1.2 Кто такой тестировщик?

Тестировщик - специалист заботящийся о качестве системы в виде обнаружения дефектов до того, как их обнаружат пользователи системы.

Тестирование служит для повышения качества системы через обнаружение дефектов и их устранение.

Вы наверное слышали такие термины как Tester, QC, QA. Их довольно часто путают между собой или смешивают, поэтому давайте разберемся в терминологии.

* Тестирование — это уже непосредственно процесс проверки результатов работы на соответствие установленным требованиям.
* QC — Контроль качества продукта — анализ результатов тестирования и качества новых версий выпускаемого продукта.
* QA — Обеспечение качества продукта — изучение возможностей по изменению и улучшению процесса разработки, улучшению коммуникаций в команде, где тестирование является только одним из аспектов обеспечения качества.

Без понимания основ теории тестирования невозможно стать хорошим автоматизатором тестирования. Она необходима для понимания важности работы тестировщика и его участии в развитии продукта и работе в команде.

Заострим внимание ещё на одном важном аспекте теории тестирования - принципах тестирования.

Выделяют семь принципов тестирования:

1. Исчерпывающее тестирование невозможно (Exhaustive testing is impossible).
2. Тестирование демонстрирует наличие дефектов, а не их отсутствие (Testing shows presence of defects).
3. Тестирование зависит от контекста (Testing is context depending).
4. Раннее тестирование (Early testing).
5. Скопление или кластеризация дефектов (Defects clustering).
6. Парадокс пестицида (Pesticide paradox).
7. Заблуждение об отсутствии ошибок (Absence-of-errors fallacy).

Подробнее о принципах тестирования вы можете ознакомиться в статьях на Habr “Фундаментальная теория тестирования[[4]](#footnote-3)” и “Принципы тестирования: нас 7[[5]](#footnote-4)”.

## 1.2 Классификация тестирования

Существует много различных классификаций тестирования и каждый автор обладает своим мнением по классификации тестирования. Поэтому говоря о классификации тестирования лучше уточнить согласно какому источнику от вас хотят услышать ответ.

Проанализировав различные источники можно разделить классификацию тестирования по различным способам классифицировать само тестирование.

По уровням тестирования: модульное, интеграционное, системное, End-to-End, приёмочное.

По принципу работы с приложением: позитивное, негативное.

По доступности: черный ящик, серый ящик, белый ящик.

По запуску кода: статическое, динамическое.

По степени автоматизации: ручное, полуавтоматизированное, автоматизированное.

По зависимости от целей тестирования: функциональное, нефункциональное.

В свою очередь каждая классификация содержит в себе входящие в себя подкатегории, которые на стыке друг с другом образуют новые виды тестирования. Например, на пересечении конфигурационного и UI тестирования возникает тестирование совместимости, направленное на проверку того, что интерфейс корректно работает при различных конфигурациях ПО и платформ. А на срезе тестирования UI и безопасности возникает тестирование локализации и интернационализации.

## 1.3. Тестирование UI

Важно понимать на каких аспектах работы ПО сосредоточено UI тестирование для успешного изучения автоматизации UI тестирования.

UI тестирование, также известное как тестирование пользовательского интерфейса, представляет собой тип тестирования программного обеспечения, которое фокусируется на графическом пользовательском интерфейсе приложения. Тестирование UI проводится, чтобы удостовериться, что интерфейс приложения работает должным образом, а также что им легко пользоваться и ориентироваться.

Тестирование пользовательского интерфейса важно для выявления проблем, таких как неработающие ссылки, неправильное форматирование или некорректное поведение при взаимодействии с интерфейсом, работу элементов взаимодействия с пользователем таких как кнопки, поля ввода, модальные окна, выпадающие списки, меню и прочее.

Несколько примеров UI тестирования:

1. Функциональное тестирование. Этот тип тестирования гарантирует, что компоненты пользовательского интерфейса, такие как кнопки и формы, работают должным образом. Например, если пользователь нажимает кнопку, UI должен выполнить ожидаемое действие.

2. Юзабилити-тестирование. Этот тип тестирования гарантирует, что пользовательский интерфейс прост в использовании и навигации. Например, проверяется подходящий размер шрифта и визуально привлекательная цветовая схема.

3. Тестирование производительности. Этот тип тестирования гарантирует, что пользовательский интерфейс отзывчив и хорошо работает в различных условиях. Например, проверяется насколько быстро загружается интерфейс и как он работает при одновременном доступе к нему нескольких пользователей.

4. Тестирование совместимости. Этот тип тестирования гарантирует, что пользовательский интерфейс хорошо работает на различных устройствах и платформах. Например, проверяется правильность отображения приложения на экранах разных размеров и разрешений, работу приложения в разных браузерах.

5. Тестирование доступности. Этот тип тестирования гарантирует, что пользовательский интерфейс доступен для пользователей с ограниченными возможностями. Например, он проверяет, можно ли использовать приложение с программой чтения с экрана и что используемые цвета доступны пользователям с цветовой слепотой.

Это всего лишь несколько примеров типов тестирования пользовательского интерфейса, которые можно выполнить, чтобы убедиться, что пользовательский интерфейс приложения работает должным образом.

Наибольшее распространение получило именно автоматизированное функциональное тестирование пользовательского интерфейса, которое мы и будем изучать на стеке Selenium + Pytest.

Подведём небольшой итог этой главы: без знания основ тестирования невозможно стать хорошим автоматизатором тестирования. Требуется гораздо более глубокое понимание того как, когда и зачем производится определённый вид тестирования. Только в сочетании знания теории тестирования и навыков программирования возможно добиться высоких показателей качества продукта, который вы разрабатываете.

# Глава 2. Базовые знания web-технологий

## 2.1 Основы HTML

### 2.1.1 Что такое HTML?

HTML — это не язык программирования, а язык разметки, используемый для структурирования и представления веб-страниц и их содержимого. Он позволяет организовывать текст в абзацы, создавать маркированные списки, вставлять изображения и отображать данные в таблицах. Понимание основ HTML и его особенностей имеет важное значение для любого тестировщика. Но для автоматизации тестирования является неотъемлемой часть знаний.

HTML использует теги, которые представляют собой метки, используемые для разметки и классификации информации, что упрощает для браузеров интерпретацию и форматирование контента на веб-сайте. Проще говоря, теги служат ключевыми словами, которые указывают браузеру, как обрабатывать и отображать содержимое сайта.

### 2.1.2 HTML элемент

Давайте рассмотрим структуру HTML элемента.

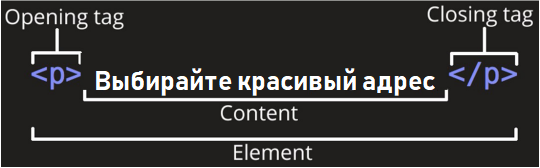


Рисунок 1. HTML элемент.

Главными частями нашего элемента являются:

1. Открывающий тег (Opening tag): Состоит из имени элемента (в данном случае, "p"), заключённого в открывающие и закрывающие угловые скобки. Открывающий тег указывает, где элемент начинается или начинает действовать, в данном случае — где начинается абзац.
2. Закрывающий тег (Closing tag): Это то же самое, что и открывающий тег, за исключением того, что он включает в себя косую черту перед именем элемента. Закрывающий элемент указывает, где элемент заканчивается, в данном случае — где заканчивается абзац. Отсутствие закрывающего тега является одной из наиболее распространённых ошибок начинающих и может приводить к странным результатам.
3. Контент (Content): Это контент элемента, который в данном случае является просто текстом.
4. Элемент(Element): Открывающий тег, закрывающий тег и контент вместе составляют элемент.[[6]](#footnote-5)

Элементы могут иметь различные атрибуты, например класс:

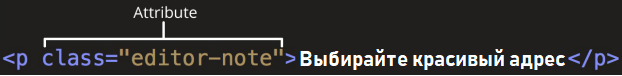


Рисунок 2. Атрибут элемента.

Атрибуты предоставляют дополнительную информацию об элементе, которая не предназначена для отображения как часть самого содержимого. В этом примере “class” — это имя атрибута, а “editor-note” — это значение атрибута. Атрибут “class” позволяет присвоить элементу описательное имя, которое впоследствии можно использовать для ссылки на элемент в целях стилизации или других функций.

Атрибуты часто используются для поиска элементов на странице, поэтому необходимо обратить на них внимание при изучении HTML.

## 2.2 Основы CSS

### 2.2.1 Так что же такое CSS?

CSS (Cascading Style Sheets) — это код, используемый для настройки внешнего вида вашей веб-страницы. Понимание основ CSS позволяет управлять такими аспектами, как цвет текста, расположение контента на экране и добавление фоновых изображений и цветов. Для автоматизации тестирования необходимо знать как меняются свойства отображения элемента, такие как цвет, размер и активное состояние.

Подобно HTML, CSS считается не языком программирования, а скорее языком таблиц стилей. Его цель — выборочное применение стилей к определенным элементам в документах HTML.

### 2.2.2 Анатомия набора правил CSS

Давайте посмотрим на CSS более подробно:

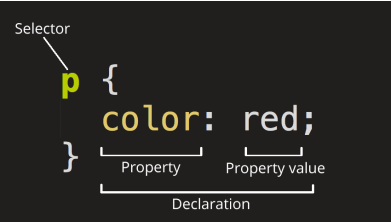


Рисунок 3. Набор правил CSS.

Вся структура называется набором правил (но зачастую для краткости "правило"). Отметим также имена отдельных частей:

Селектор (Selector)

Имя HTML-элемента в начале набора правил. Он выбирает элемент(ы) для применения стиля (в данном случае, элементы p ). Для стилизации другого элемента, просто измените селектор.

Объявление (Declaration)

Единственное правило, например color: red; указывает, какие из свойств элемента вы хотите стилизовать.

Свойства (Properties)

Способы, которыми вы можете стилизовать определённый HTML-элемент (в данном случае, color является свойством для элементов [<p>](https://developer.mozilla.org/ru/docs/Web/HTML/Element/p)). В CSS вы выбираете, какие свойства вы хотите затронуть в вашем правиле.

Значение свойства (Property value)

Справа от свойства, после двоеточия, у нас есть значение свойства, которое выбирает одно из множества возможных признаков для данного свойства (существует множество значений color, помимо red).[[7]](#footnote-6)

## 2.3 Основы JavaScript

JavaScript — это язык программирования, который повышает интерактивность вашего веб-сайта. Он включает различные функции, такие как интерактивные игры, отклики на нажатие кнопок или ввод данных в формы, динамический стиль и анимацию.

Вы можете ознакомиться с синтаксисом языка самостоятельно. Здесь же мы поговорим о наиболее часто встречаемых элементах взаимодействиях: alert, confirm и prompt.

Alert - функция, которая показывает сообщение и ждёт, пока пользователь нажмёт кнопку “ОК”.

Например:

“alert("Привет");”



Рисунок 4. Alert JS.

Окно, появляющееся поверх элементов интерфейса, называется модальным. Пользователь не сможет взаимодействовать с интерфейсом, пока не закроет модальное окно. В случае alert это нажатие на кнопку “OK”.

Код “let test = prompt("Test");“ отобразит модальное окно с полем ввода текста и кнопки “OK”, “Cancel”.

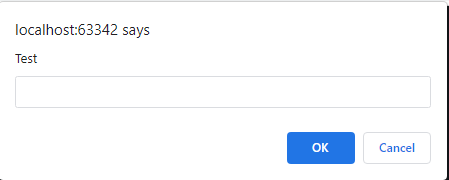


Рисунок 5. Prompt JS.

Функция confirm отображает окно с текстом вопроса и двумя кнопками: “OK”, “Cancel”. Результат - истина, если пользователь выбрал “OK” и ложь в противном случае.

Например:

“let right = confirm("Ты всегда прав?");“

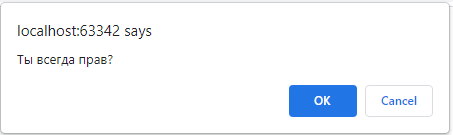


Рисунок 6. Confirm JS.

В автоматизации тестирования Java Script часто используется как для тестирования UI, так и для тестирования баз данных и API. Для автоматизации тестирования на Python вам понадобится базовое знание синтаксиса, так как некоторые вещи, чтобы сэкономить время и не писать самостоятельно, мы используем из уже реализованных методов браузера. А они в свою очередь реализованы на языке Java Script.

Подводя итог под знакомством с web-технологиями, которые сейчас активно используются для взаимодействия приложений с пользователями, нужно отметить, что знание основ этих технологий позволяет эффективно производить тестирование пользовательского интерфейса.

# Глава 3. Знакомство с системой контроля версий Git

## 3.1 Что такое система контроля версий?

Система контроля версий — это программное обеспечение, которое позволяет хранить историю изменений в любом документе или файле. Таким образом, вы можете легко просматривать и откатывать изменения, сделанные каким-либо участником проекта. Это очень полезно, когда необходимо восстановить старую версию документа или сравнить две разные версии файла. Кроме того, она позволяет отслеживать, кто и что изменил в документе или файле и помогает избежать конфликтов при одновременном редактировании одного и того же файла.

Система контроля версий Git является распределенной, она используется для отслеживания изменений в файлах и директориях проекта. Позволяет разработчикам контролировать и отслеживать изменения в проекте.

Чтобы работать с Git, необходимо инициализировать репозиторий в папке с проектом. После этого можно добавлять изменения в репозиторий, создавая различные версии проекта. Можно также просматривать историю изменений и откатываться к любой предыдущей версии. Также можно объединять работы различных разработчиков в один проект.

Я рекомендую начать изучение Git в игровой форме на сайте LearnGitBranching[[8]](#footnote-7), которое позволит вам изучить все основные возможности и команды.

Большинство современных сред разработки поддерживают интуитивно понятный интерфейс для работы с системой контроля версий Git и вам не надо знать наизусть все команды.

## 3.2. Файл README.md и разметка Markdown

Большинство open source проектов размещаются в GitHub и я рекомендую поступать также. Это не только даёт вам возможность поделиться кодом с коллегами, но и работать совместно над проектом со специалистами из разных стран.

Стоит заострить внимание на аспекте README.md файла, который создаётся для каждого репозитория. Некоторые разработчики пренебрегают созданием этого файла потому что не осознают до конца всей важности его наличия и влияния на сообщество.

README.md — это файл в репозитории GitHub, который служит файлом документации для репозитория. Он написан в формате Markdown и предоставляет информацию о назначении репозитория, инструкции по установке, примеры использования и любую другую информацию касательно содержимого проекта, например, примеры оформления баг-репортов или способы связаться с автором для совместной разработки.

Файл README.md важен, потому что он помогает пользователям понять назначение и использование репозитория. Он также предоставляет важную информацию, которая может помочь пользователям быстро и легко начать работу с репозиторием. Кроме того, наличие хорошо написанного файла README.md может помочь привлечь участников в репозиторий и улучшить совместную работу.

Часто вам потребуется читать и составлять подобные файлы, а также поддерживать их в актуальном состоянии. Поэтому важно понимать, что должен содержать такой файл.

* Название: название вашего проекта или репозитория.
* Описание: краткое описание того, что делает ваш проект или репозиторий.
* Оглавление: список разделов в вашем файле README.md, если файл получился объёмным.
* Установка: инструкции по установке и настройке вашего проекта или репозитория.
* Использование: инструкции по использованию вашего проекта или репозитория.
* Участие: рекомендации о том, как внести свой вклад в ваш проект или репозиторий (опционально).
* Лицензия: информация о лицензии вашего проекта или репозитория.

Вы можете использовать синтаксис Markdown для форматирования контента и добавления заголовков, списков, изображений, ссылок и других элементов.

Я перевел на русский язык удобную и информативную инструкцию по применению разметки Markdown[[9]](#footnote-8) с оригинальной Markdown Cheatsheet by Adam Pritchard[[10]](#footnote-9) специально для тех, кому комфортнее читать актуальную документацию на родном языке. Обязательно советую ознакомиться с ней или другими источниками.

Отметим, что автоматизация тестирования тесно связана с разработкой, поскольку является продуктом в продукте (ваш тестовый фреймворк можно считать полноценным приложением). Поэтому без уверенного знания системы контроля версий работать практически невозможно. А без надлежащего оформления сопутствующей документации проекта вы уже через месяц забудете какие команды нужно использовать, чтобы получить необходимые результаты. Кроме того, отсутствие актуальной документации не только приведёт к трате времени на объяснение того как работает ваш проект, но и в целом ухудшает потенциальное качество проекта.

# Глава 4. Основы Python

## 4.1 Знакомство с языком программирования Python

Программирование - это, по сути, процесс предоставления компьютеру инструкций для выполнения конкретной задачи. Для этого нам нужно использовать язык программирования, который, по сути, представляет собой способ общения с компьютером таким образом, чтобы он мог его понять.

Python - популярный язык программирования, известный своей простотой в изучении и использовании. Некоторые из основных концепций программирования, применимых к Python, включают:

1. Синтаксис. Синтаксис языка программирования относится к правилам и соглашениям по написанию кода на этом языке. В Python, например, мы используем отступы для обозначения блоков кода, и мы используем определенные ключевые слова, такие как “if”, “else” и “while”, чтобы управлять потоком программы.
2. Типы и структуры данных. В Python мы можем работать с различными типами данных, включая числа, строки и списки. Мы также можем использовать структуры данных, такие как словари и множества, для организации данных и управления ими.
3. Структуры управления и ветвления. Структуры управления используются для управления потоком программы. В Python мы можем использовать операторы “if” для выполнения определенного кода, только если выполняется определенное условие, циклы “while” для повторения блока кода, пока определенное условие выполняется, и циклы “for” для повторения последовательности элементов.
4. Функции. Функции — это повторно используемые блоки кода, которые выполняют определенную задачу. В Python мы можем определить наши собственные функции с помощью ключевого слова “def”, а также можем использовать встроенные функции, такие как “print” и “len”.
5. Объектно-ориентированное программирование. Объектно-ориентированное программирование — это парадигма программирования, в которой особое внимание уделяется использованию объектов для представления данных и поведения. В Python мы можем определять наши собственные классы и объекты, которые позволяют нам организовывать наш код и создавать более сложные программы.

Это всего лишь несколько основных концепций программирования, применимых к Python.

Базовый синтаксис языка программирования Python достаточно прост и рекомендуется самостоятельно его изучить.

## 4.2 Базовые принципы ООП

Важным элементом для автоматизатора тестирования на любом языке программирования является понимание объектно-ориентированной парадигмы, которая сейчас очень распространена в промышленной разработке.

Объектно-ориентированное программирование (ООП) — это парадигма программирования, основанная на концепции объектов, которые могут содержать данные (атрибуты) и функции (методы), работающие с этими данными. ООП — это мощная парадигма программирования, которую можно использовать для создания более модульного, повторно используемого и поддерживаемого кода.

Однако количество кода растёт по экспоненте с количеством классов, которые вы внедряете по мере разработки.

Существует четыре основных принципа ООП:

1. Инкапсуляция. Относится к практике сокрытия внутренних деталей объекта от внешнего мира и раскрытия только общедоступного интерфейса, который можно использовать для взаимодействия с объектом. Это помогает защитить целостность объекта и упрощает поддержку и изменение кода с течением времени.
2. Абстракция. Относится к практике представления сложных систем или понятий с использованием более простых и общих понятий. Абстракция позволяет нам сосредоточиться на основных характеристиках объекта и игнорировать детали, которые не имеют отношения к нашей текущей задаче.
3. Наследование. Относится к практике создания новых классов, которые наследуют атрибуты и методы существующих классов. Наследование позволяет нам повторно использовать код и создавать более специализированные классы, основанные на функциональности более общих классов.
4. Полиморфизм. Относится к практике использования одного интерфейса для представления нескольких типов объектов. Полиморфизм позволяет нам писать более общий код, который может работать с широким спектром объектов, без необходимости знать конкретные детали каждого объекта.

В Python мы можем использовать принципы ООП для создания классов и объектов, которые инкапсулируют данные и функциональность. Мы можем определить классы, используя ключевое слово “class”, а затем создать экземпляры этих классов, используя имя класса, за которым следуют круглые скобки. Например:

1. ***class*** *Person:*
2. ***def*** *\_\_init\_\_(self, name, age):*
3. *self.name = name*
4. *self.age = age*
6. ***def*** *say\_hello(self):*
7. ***print****(f'Hello, my name is {self.name} and I am {self.age} years old.')*

10. *person1 = Person('John', 30)*
11. *person2 = Person('Jane', 25)*
13. *person1.say\_hello() # prints "Hello, my name is John and I am 30 years old."*
14. *person2.say\_hello() # prints "Hello, my name is Jane and I am 25 years old."*

В этом примере мы определяем класс “Person”, который имеет два атрибута (“name” и “age”) и один метод (“say\_hello()”). Затем мы создаем два экземпляра класса “Person” (“person1” и “person2”) и вызываем метод “say\_hello()” для каждого экземпляра.

В целом, принципы ООП являются важным инструментом для создания более модульного, повторно используемого и поддерживаемого кода в Python. Освоив использование ООП, вы сможете поднять свои навыки программирования на новый уровень и создавать более мощные, гибкие и эффективные программы. К тому же, ООП повсеместно используется в автоматизации тестирования и наибольшее распространение получила Page Model Object, о которой мы поговорим позже.

# Глава 5. Знакомство с Selenium

## 5.1 Что такое Selenium?

Selenium — это среда тестирования программного обеспечения с открытым исходным кодом, которая используется для автоматизации веб-приложений. Она предоставляет набор инструментов и API, которые позволяют QA-инженерам автоматизировать тестирование и решать задачи, такие как взаимодействие с пользователем, ввод данных и проверка вывода. Selenium поддерживает различные языки программирования, такие как Java, Python, C#, Ruby, JavaScript и другие.

Selenium работает, имитируя действия пользователя в веб-браузере. Он использует компонент драйвера, который взаимодействует с веб-браузером и позволяет тестам автоматизации взаимодействовать с веб-страницей. Драйвер Selenium WebDriver является наиболее часто используемым драйвером и поддерживает популярные браузеры, такие как Chrome, Firefox,

Safari и Edge.

Вот несколько способов использования Selenium для автоматизированного тестирования.

Функциональное тестирование: Selenium можно использовать для автоматизации функциональных тестов, чтобы убедиться, что веб-приложения работают должным образом. Это включает в себя проверку того, что элементы пользовательского интерфейса отображаются правильно, что ссылки и кнопки работают должным образом, а поля формы принимают допустимый ввод.

Регрессионное тестирование: Selenium можно использовать для автоматизации регрессионных тестов, чтобы гарантировать, что изменения, внесённые в веб-приложение, не привели к появлению новых ошибок или проблем. Регрессионные тесты можно запускать несколько раз, чтобы убедиться, что приложение продолжает функционировать должным образом после каждого внесения изменений. Тут стоит отметить тот факт, что именно регрессионное тестирование является первым кандидатом на автоматизацию.

Интеграционное тестирование: Selenium можно использовать для автоматизации интеграционных тестов, чтобы убедиться, что различные части веб-приложения работают вместе правильно. Это включает в себя тестирование взаимодействия между интерфейсными и внутренними компонентами веб-приложения. Стоит упомянуть, что большинство авторов относят автоматизацию тестирования UI к End-to-end тестам, так как они проходят через всё приложение начиная от ввода в пользовательский интерфейс и заканчивая выводом конечных данных в нём же.

Кросс-браузерное тестирование: Selenium можно использовать для автоматизации кросс-браузерного тестирования, чтобы убедиться, что веб-приложения работают согласованно в разных браузерах и версиях. Это помогает выявлять и устранять проблемы, связанные с браузерами.

В целом, Selenium — это мощный инструмент для автоматизации тестирования веб-приложений, который может помочь QA-инженерам сэкономить время и силы за счёт автоматизации повторяющихся и трудоёмких задач тестирования.

## 5.2 Настройка первого запуска браузера с помощью Selenium Webdriver

На этом этапе я рекомендую создать свой учебный репозиторий в GitHub и повторять и анализировать действия в среде разработке PyCharm[[11]](#footnote-10). Создайте свой проект, настройте виртуальное окружение и не забудьте про README.md файл. Рекомендую добавить README.md и .gitignore для Python сразу при создании репозитория в GitHub.

Мы будем с вами изучать автоматизацию на языке Python, поэтому первое, что вы должны сделать - это скачать версию 3.11.2[[12]](#footnote-11) (на момент написания - это последняя версия языка Python).

В виртуальном окружении устанавливаем библиотеку Selenium: “pip install selenium==4.\*”.

В этом дипломном проекте мы будем работать с драйвером для Chrome, так как на данный момент это самый популярный браузер согласно ресурсу statcounter[[13]](#footnote-12), и в первую очередь следует убедиться, что веб-приложение работает для большинства пользователей.

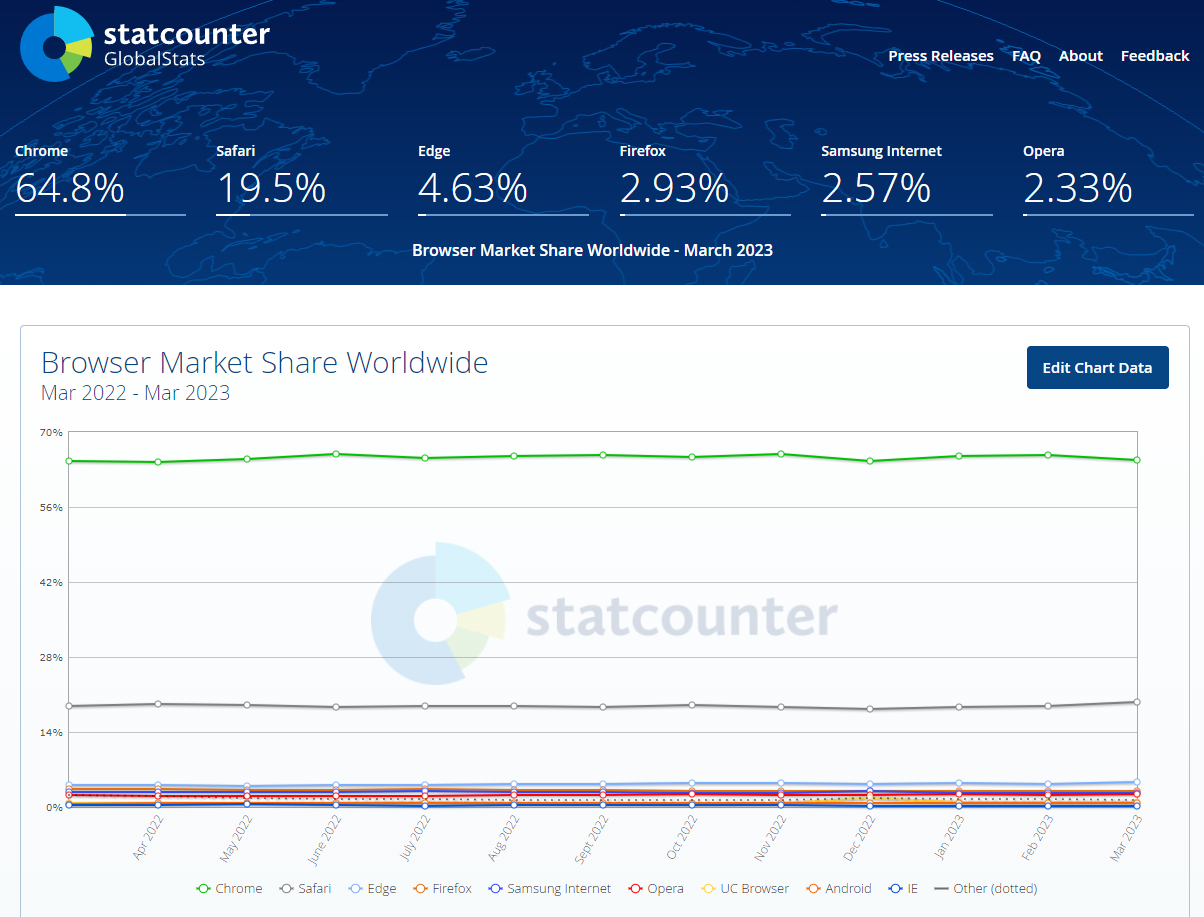


Рисунок 7. Статистика использования браузеров.

Драйвер для Chrome разрабатывается командой браузера и носит название ChromeDriver. Скачать нужную версию можно с официального сайта.[[14]](#footnote-13)

Подробности установки драйвера хорошо описаны на сайте selenium-python.com[[15]](#footnote-14).

Далее мы создадим скрипт, который открывает веб-страницу и заполняет форму. Я буду использовать сайты demoqa.com[[16]](#footnote-15) и testpages.herokuapp.com[[17]](#footnote-16) для демонстрации возможностей Selenium.

Рекомендую попробовать запускать команды по отдельности или пользоваться отладчиком среды разработки, чтобы получить более глубокое понимание работы программы.

1. **import** time
3. *# webdriver - это набор команд для управления браузером.*
4. **from** selenium **import** webdriver
6. *# By - класс, который позволяет выбрать способ поиска элемента.*
7. **from** selenium.webdriver.common.by **import** By
9. *# Инициализация браузера Chrome.*
10. driver = webdriver.Chrome()
12. *# Задержка в 5 секунд, чтобы вы увидили, что происходит в браузере.*
13. *# Попробуйте её закомментировать и посмотреть, что из этого выйдет.*
14. time.sleep(5)
16. *# Метод get открывает страницу в браузере.*
17. driver.get('https://testpages.herokuapp.com/styled/basic-html-form-test.html')
18. time.sleep(5)
20. *# Метод find\_element() позволяет найти нужный элемент на сайте по указанному локатору (селектору).*
21. *# Метод принимает в качестве аргументов способ поиска элемента и значение, по которому будет осуществляться поиск.*
22. *# Ищем поле ввода.*
23. username\_field = driver.find\_element(By.CSS\_SELECTOR, '[name="username"]')
25. *# Вводим текст в поле, используя метод send\_keys().*
26. username\_field.send\_keys('Alexander')
27. time.sleep(5)
29. *# Найдем кнопку, которая отправляет введённый текст.*
30. submit\_button = driver.find\_element(By.CSS\_SELECTOR, '[value="submit"]')
32. *# Скажем драйверу, что нужно кликнуть по кнопке.*
33. *# После этой команды мы должны увидеть ввёденные значения в форму.*
34. submit\_button.click()
35. time.sleep(5)
37. *# Мы должны не забыть закрыть окно браузера, но если этого не сделать, то за нас это сделает сборщик мусора Python.*
38. driver.quit()

Пронаблюдайте за работой скрипта, а затем еще раз пробегитесь по коду глазами. Одна из особенностей языка Python в том, что он пишется почти как предложения на английском языке. Такая высокая читаемость кода позволяет гораздо легче изучать язык.

Следует обязательно добавить к проекту файл “requirements.txt”. В этот файл следует занести все зависимости, которые используются для работы вашей программы, чтобы любой мог склонировать себе ваш репозиторий и запустить его на своей локальной машине.

Записать такой файл можно вручную или воспользоваться командой: “pip freeze > requirements.txt”.

## 5.3 Поиск элементов

Тема поиска элементов критично важна для автоматизации тестирования UI и мы остановимся на ней подробнее.

### 5.3.1 Поиск элементов с помощью CSS-селекторов

Ниже приведены части элементов HTML-страницы, по которым можно найти элемент:

* id;
* значение атрибута;
* name;
* class;
* tag.

Использование тега для поиска элемента используется в сочетание с другими частями элемент, потому что важно выбрать единственный объект для взаимодействия кроме некоторый исключений из этого правила. И, как правило, на странице присутствует несколько элементов с одинаковым тегом, а когда таких элементов нет все равно не рекомендуется использовать исключительно тег элемента, так как страница может измениться: могут быть добавлены новые элементы, изменено наполнение страницы.

Давайте откроем страницу Basic Web Page Title[[18]](#footnote-17) и попробуем найти элемент первого параграфа.

Чтобы это сделать вам потребуется использовать DevTools вашего браузера.

Инструменты разработчика (от англ. "development tools" или сокращённо "DevTools") - это программы, которые позволяют создавать, тестировать и отлаживать (debug) программное обеспечение.[[19]](#footnote-18)

Я рекомендую вам использовать Google Chrome.

Для включения DevTools можно нажать F12 на клавиатуре.

Ниже приведён кусок html-кода страницы:

1. <**div** class="centered">
2. <**p** id="para1" class="main">A paragraph of text</**p**>
3. <**p** id="para2" class="sub">Another paragraph of text</**p**>
4. </**div**>

#### 5.3.1.1 Поиск по id

У элемента с первым параграфом есть атрибут id=”para1”, а значит, мы однозначно можем найти его с помощью селектора “#para1” (символ “#” означает, что мы ищем по “id” со значением “para1”).

Нужно всегда проверять правильность подобранного локатора в консоли разработчика. Откройте консоль разработчика и перейдите в ней в таб “Elements”. Затем нажмите “ctrl+F” и в открывшейся внизу поисковой строке введите “#para1”. Если селектор написан правильно,

то вы увидите подсвеченный элемент на веб-странице, а также элемент будет подсвечен жёлтым цветом в html-коде. Еще в поисковой строке вы увидите количество найденных элементов.

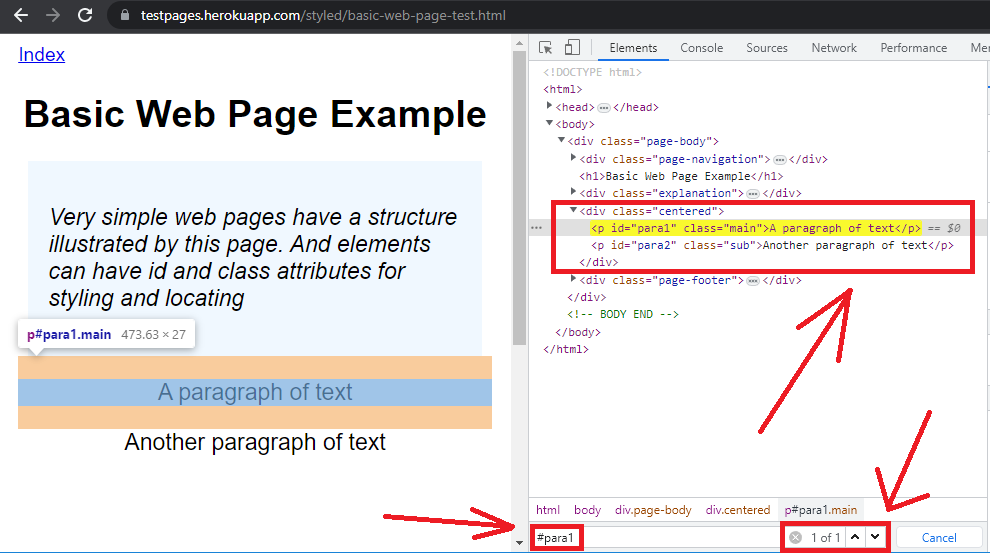


Рисунок 8. Поиск по id.

#### 5.3.1.2 Поиск по class

Рассмотрим поиск второго параграфа по классу. Поскольку классы используются повсеместно для задания стилей элементов, то у них есть краткая форма поиска: “.sub”. Необходимо указать на использование класса для поиска использовав точку, а затем ввести значение класса.

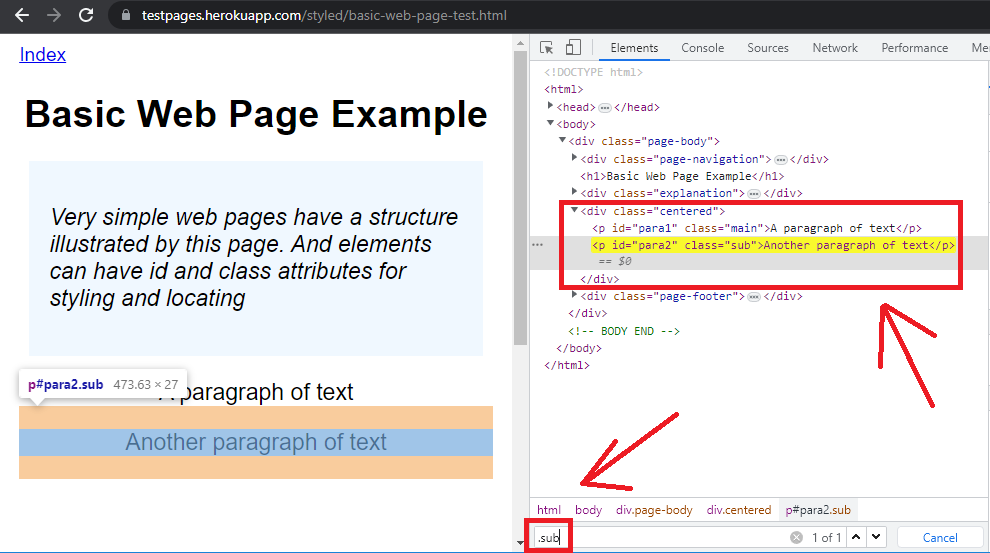


Рисунок 9. Поиск по class.

Важно отметить, что поиск по классу чувствителен к регистру и поиск “.Sub” уже не найдёт нужный элемент.

#### 5.3.1.3 Поиск по tag

Чтобы найти элемент по тегу, просто напишите название тега в поисковой строке. Поиск по тегам не очень удобен, т.к. разработчики используют небольшое количество тегов для разметки страниц, и скорее всего, одному тегу будет соответствовать несколько элементов. Но есть некоторые теги, которые почти наверняка будут присутствовать в единичном экземпляре: html, title, h1.

Давайте попробуем найти заголовок нашей страницы Basic Web Page Title[[20]](#footnote-19).

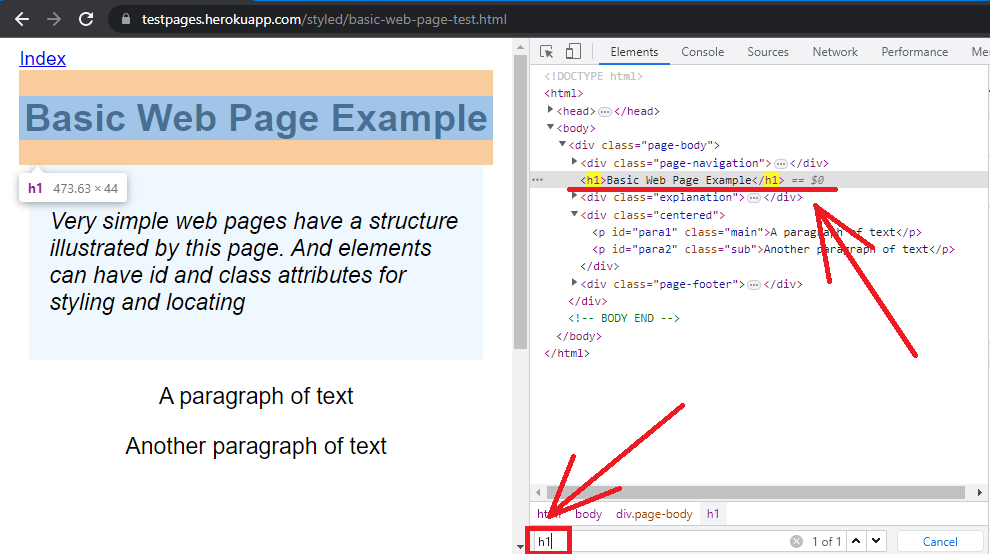


Рисунок 10. Поиск по tag.

#### 5.3.1.4 Поиск по значению атрибута

Можно найти элемент, указав название атрибута и его значение. Например, можно переписать поиск по “id” в следующем виде “[id="para1"]” вместо “#para1” и поиск по “class” в следующем виде “[class="sub"]” вместо “.sub”.

Вариант с квадратными скобками принято использовать для тех видов поиска, у которых отсутствует короткая поисковая команда.

Рассмотрим пример параграфа с атрибутами (This paragraph has attributes) и попробуем найти содержащий его элемент на странице Test Page For Element Attributes[[21]](#footnote-20).

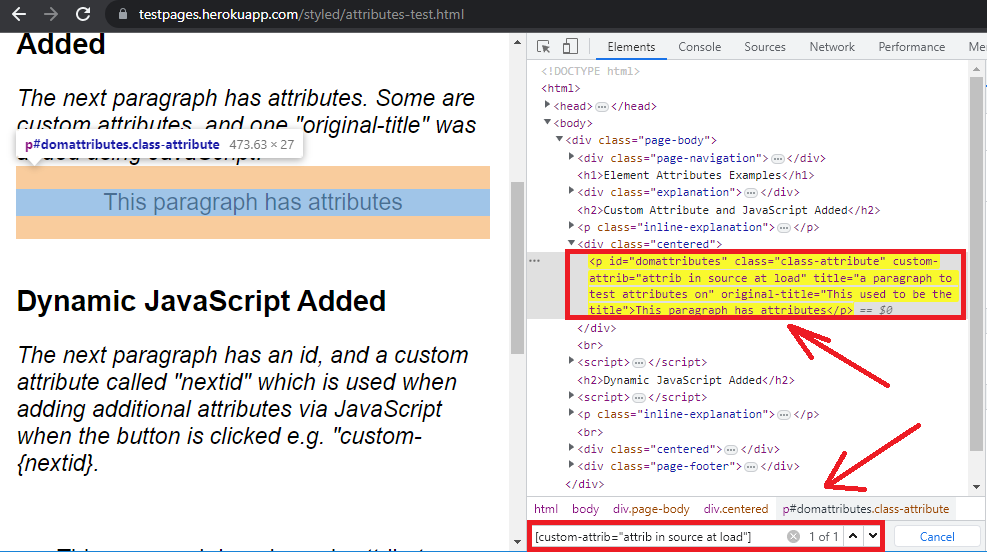


Рисунок 11. Поиск по значению атрибута.

Поиск по “name” является разновидностью поиска по значению атрибута и выделяется отдельно в Selenium WebDriver.

#### 5.3.1.5 Поиск элементов с помощью составных CSS-селекторов

Допустим мы не можем найти необходимый нам элемент на странице используя простой селектор, так как он находит сразу несколько элементов сразу. Что делать в такой ситуации?

Вот тут нам и понадобится умение комбинировать селекторы и иерархическая структура страницы.

##### 5.3.1.5.1 Использование порядкового номера дочернего элемента

Например, нам необходимо получить только текст пункта “The Attributes window, has no name.” на странице Windows Example Test[[22]](#footnote-21).

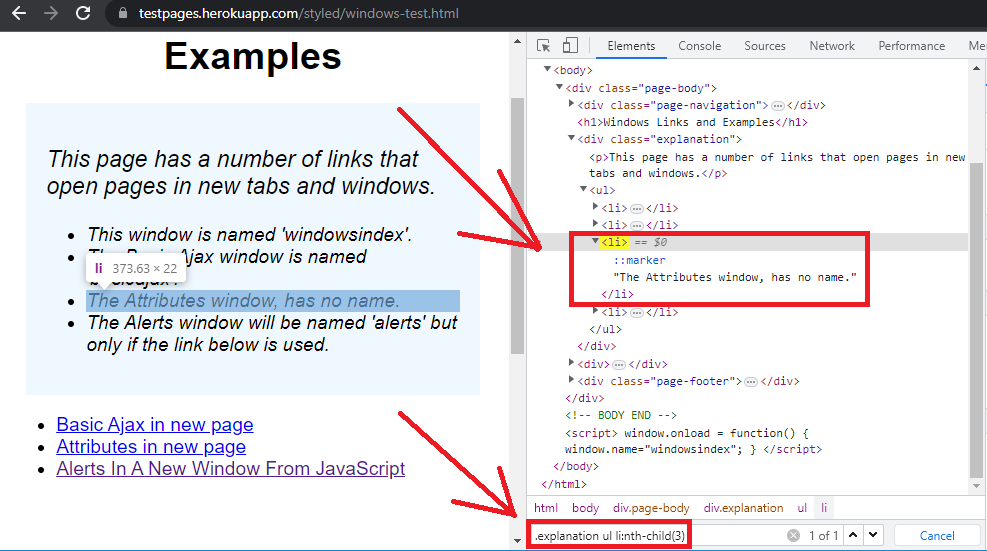


Рисунок 12. Поиск с использованием порядкового номера дочернего элемента.

Псевдокласс “:nth-child(3)” позволяет выбрать третий дочерний элемент тега “ul” с тегом “li”.

##### 5.3.1.5.2 Использование дочерних элементов

Допустим нам нужно содержимое параграфа “This is a paragraph text” на странице Welcome to the Find By Playground[[23]](#footnote-22), заключённое в тег “a”. Для этого можно использовать следующую команду для поиска элемента: “.specialDiv > p.normal #a1”.

Символ “>” означает, что нужно взять элемент строго на один уровень вложенности ниже элемента с классом specialDiv. Этот элемент должен иметь тег “p” и класс “normal”, а уже далее от этого элемента производится поиск по “id”.

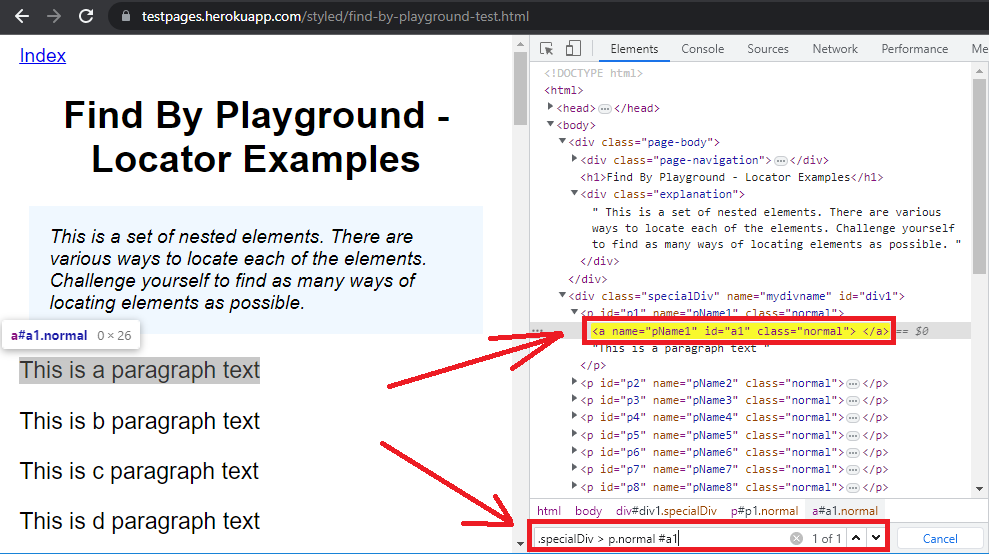


Рисунок 13. Поиск с использованием дочернего элемента.

##### 5.3.1.5.3 Использование потомков

Попробуем найти ввод цвета (Colour) на странице HTML5 Form Elements Test Page[[24]](#footnote-23).

Можно написать следующий локатор элемента: “.page-body .centered .form-label [name="colour"]”.

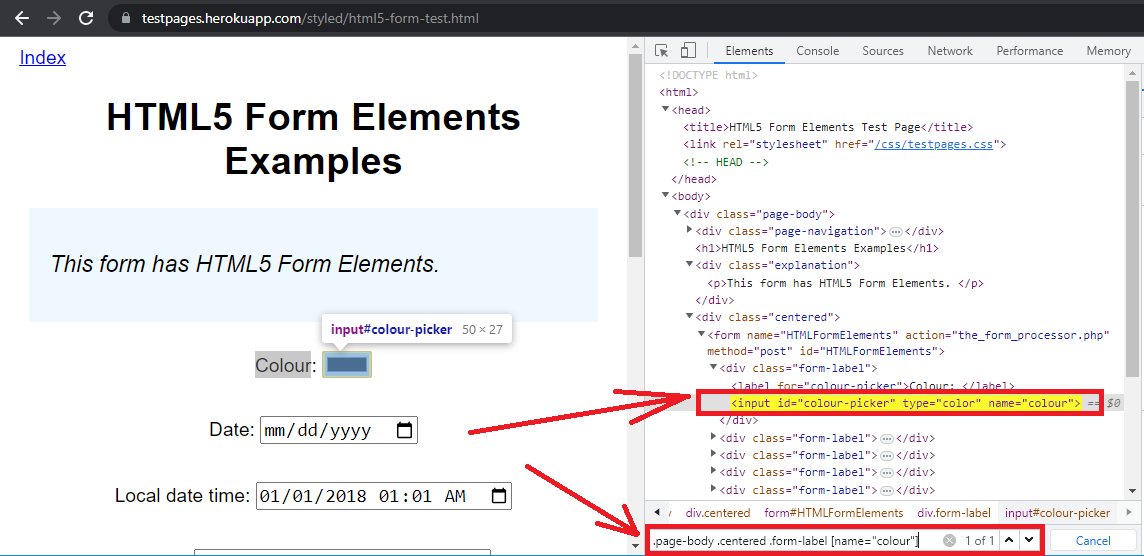


Рисунок 14. Поиск с использованием потомков.

Пробел является значащим в CSS-селекторах, это разделитель предка и потомка. Потомок может находиться на любом уровне вложенности, так например все элементы с классом “centered” являются потомками класса “page-body”.

### 5.3.2 Поиск элементов с помощью XPath

XPath (XML Path Language) это язык запросов, который использует древовидную структуру документа.[[25]](#footnote-24) Это мощный и гибкий инструмент, который позволяет использовать сложные поисковые запросы для нахождения нужного элемента.

Разберём кратко особенности синтаксиса XPath.

1. XPath запрос всегда начинается с символа “/” или “//”.

Символ “/” аналогичен символу “>” в CSS-селекторе, а символ “//” - пробелу.

Разница в том, что когда мы используем символ “/” в XPath локаторе, то мы всегда обязаны указывать корень документа. У html страницы корнем всегда будет элемент с тегом “<html>”.

Мы можем начать XPath запрос с символа “//” - это будет означать, что мы хотим найти всех потомков корневого элемента без его указания.

2. Квадратные скобки “[ ]” - это команда фильтрации в XPath.

Некоторые правила фильтрации, которые могут вам пригодится в составлении XPath-селекторов рассмотрим на примере страницы DEMOQA Practice Form[[26]](#footnote-25).

* По любому атрибуту будь то class, name, type или любой другой. Например, нам нужна кнопка “Submit”: “//button[@type="submit"]”.
* По порядковому номеру. Допустим нам необходимо найти радиобаттон женского пола в форме для активации. Для этого подойдет следующий локатор: “//div[@id="genterWrapper"]/div/div[2]/input”, где “[2]” будет означать, что из списка элементов, которые подходят под данный локатор необходимо выбрать второй по счёту.
* По частичному совпадению. Мы можем найти элемент по внутреннему тексту, который расположен внутри него используя функцию “text()”. Давайте найдём таким методом кнопку “Submit”: “//button[contains(text(), "Submit")]”. Также этот метод подойдет для поиска частичного совпадения атрибутов, например, для поиска поля ввода текущего адреса: “//textarea[contains(@class, "form")]”. Обратите внимание, что класса “form” нет у поля ввода адреса, но есть класс “form-control”, который и нашёл наш селектор.

3. Символ “\*” - команда выбора всех элементов.

Может быть удобна для выбора всех интересующих нас элементов или когда точно неизвестен тег элемента для поиска: “//div/\*[@class="form-control"]”

4. Поиск по классам с помощью XPath регистрозависим также как и в CSS-селекторах.

Поиск по локатору “//div/\*[@class="Form-control"]” не даст результатов.

Мы рассмотрели базовые селекторы, которых будет достаточно для написания простых UI-тестов.

## 5.4 Поиск элементов с помощью Selenium WebDriver

### 5.4.1 Поиск элементов с помощью Selenium

Для поиска элементов на странице в Selenium WebDriver используются несколько методов, позволяющих искать по CSS-селекторам и XPath-селекторам. Для поиска Selenium предоставляет метод “find\_element”, который принимает два аргумента - тип локатора и значение локатора.

Существуют следующие методы поиска элементов:

* “find\_element(By.ID, value)” - поиск по уникальному атрибуту “id” элемента;
* “find\_element(By.CSS\_SELECTOR, value)” - поиск элемента с помощью правил на основе CSS;
* “find\_element(By.XPATH, value)” - поиск с помощью языка запросов XPath;
* “find\_element(By.NAME, value)” - поиск по атрибуту name элемента;
* “find\_element(By.TAG\_NAME, value)” - поиск элемента по названию тега элемента;
* “find\_element(By.CLASS\_NAME, value)” - поиск по значению атрибута class;
* “find\_element(By.LINK\_TEXT, value)” - поиск ссылки на странице по полному совпадению;
* “find\_element(By.PARTIAL\_LINK\_TEXT, value)” - поиск ссылки на странице, если текст селектора совпадает с любой частью текста ссылки.

На практике используются “find\_element(By.CSS\_SELECTOR, value)” и “find\_element(By.XPATH, value)” чаще всего, изменяя только значение.

Давайте найдем кнопку “Submit” на странице DEMOQA Text Box[[27]](#footnote-26).

Сперва заходим на сайт и находим уникальный селектор для кнопки. Нам повезло и у кнопки есть атрибут “id”: “id="submit"”.

Посмотрим на код:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. browser = webdriver.Chrome()
5. browser.get('https://demoqa.com/text-box')
6. button = browser.find\_element(By.CSS\_SELECTOR, '#submit')

Класс By содержит все возможные локаторы. Вы можете использовать какой-нибудь другой.

Поиск нескольких элементов.

Вы можете столкнуться с ситуацией, когда вам нужно получить сразу несколько элементов или собрать коллекцию похожих элементов для выполнения с ними каких-либо операций. Для этих целей есть метод “find\_elements()”, который мы рассмотрим чуть позже.

### 5.4.2 Работа с браузером в Selenium

Если вы запускаете примеры кода в своей среде разработке, то скорее всего вы задались вопросом о причине закрытия браузера, хотя явного закрытия браузера в коде нет. Дело в работе сборщика мусора Python, который делает это за нас. Но все же правильным будет указывать явно закрытие браузера.

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. link = 'https://demoqa.com/buttons'
5. browser = webdriver.Chrome()
6. browser.get(link)
7. click\_me\_button = browser.find\_element(By.CSS\_SELECTOR, '.col-md-6 div:nth-child(3) button')
8. click\_me\_button.click()
10. *# закрываем браузер после всех манипуляций*
11. browser.quit()

Но что произойдёт, если до момента закрытия браузера возникнет ошибка?

Один из подходов к решению этой проблемы - это использование конструкции “try/finally”:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. link = 'https://demoqa.com/buttons'
5. **try**:
6. browser = webdriver.Chrome()
7. browser.get(link)
8. click\_me\_button = browser.find\_element(By.CSS\_SELECTOR, '.col-md-6 div:nth-child(3) button')
9. click\_me\_button.click()
10. **finally**:
11. *# закрываем браузер после всех манипуляций*
12. browser.quit()

#### 5.4.2.1 Поиск всех необходимых элементов.

Допустим вам необходимо убедиться, что присутствуют все девять элементов содержащие гиперссылки на странице DEMOQA Links[[28]](#footnote-27). Для этого вы можете использовать метод “find\_elements()” и написать следующий код:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. page = 'https://demoqa.com/links'
5. browser = webdriver.Chrome()
6. browser.get(page)
8. link\_locator = (By.CSS\_SELECTOR, '#linkWrapper p a')
10. links = browser.find\_elements(\*link\_locator)
11. number\_of\_links = len(links)
13. browser.quit()
15. *# Проверяем, что количество ссылок на странице равно 9.*
16. **assert** number\_of\_links == 9

Отметим несколько важных моментов касательно поиска элементов.

* Идеальный селекто - это уникальный селектор, который идентифицирует один конкретный элемент на странице.
* Эффективные автотесты - тесты, сосредоточенные на проверки отдельных функциональных единиц. Не стоит писать большие тестовые сценарии, так как простые тесты легче читаются вашими коллегами и позволяют быстро найти проблему, если один из тестов упадёт.
* Использование метода “sleep()” из библиотеки “time” для ожидания загрузки или изменения свойств элементов считается плохой практикой, так как заранее сложно спрогнозировать время ожидания, которое зависит от большого числа внешних факторов. Мы ещё вернёмся к решению этого вопроса более красивыми методами, а пока вы можете использовать “time.sleep()” для наглядности.

# Глава 6. Полезные методы Selenium

## 6.1 Как работать с элементами типа checkbox и radiobutton?

Checkbox (чекбокс) и radiobutton (радиобаттон или переключатель) — часто используемые в формах элементы. Основная разница между ними состоит в том, что чекбоксы позволяют выбирать/отключать любой из представленных вариантов, а переключатели позволяют выбрать только один из вариантов. Далее мы будем называть эти элементы на англоязычный манер: checkbox и radiobutton.

Оба этих элемента создаются при помощи тега `input` со значением атрибута `type` равным “checkbox” или “radio” соответственно. В html-коде DEMOQA Practice Form[[29]](#footnote-28) страницы вы увидите:

<**input** name="gender" required="" type="radio" id="gender-radio-1" class="custom-control-input" value="Male">

<**input** type="checkbox" id="hobbies-checkbox-1" class="custom-control-input" value="1">

Radiobuttons объединяются в группу, где все элементы имеют одинаковые значения атрибута “name”, но разные значения атрибута “value”:

<**input** name="gender" required="" type="radio" id="gender-radio-1" class="custom-control-input" value="Male">

<**input** name="gender" required="" type="radio" id="gender-radio-2" class="custom-control-input" value="Female">

<**input** name="gender" required="" type="radio" id="gender-radio-3" class="custom-control-input" value="Other">

Checkboxes могут иметь как одинаковые, так и разные значения атрибута name. Поэтому и те, и другие лучше искать с помощью значения “id” или значения атрибута “value”.

Чтобы выбрать элемент, необходимо кликнуть по нему, для этого необходимо использовать метод “click()” для найденного элемента.

Пример выбора radiobutton пола Other в форме:

1. gender\_other\_radio = browser.find\_element(By.CSS\_SELECTOR, "[value='Other']")
2. gender\_other\_radio.click()

Рядом с радиобаттоном вы увидите тег “label”, который позволяет кликать по тексту. Текст заключен в тег “label”. Элемент “label” связан с элементом ввода с помощью атрибута “for”, который соответствует значению атрибута “id” элемента ввода:

1. <**div** class="custom-control custom-radio custom-control-inline">
2. <**input** name="gender" required="" type="radio" id="gender-radio-2" class="custom-control-input"
3. value="Female">
4. <**label** title="" for="gender-radio-2" class="custom-control-label">Female</**label**>
5. </**div**>

Раз текст радиобаттона кликабельный, то можно использовать метод “click()” на нём.

## 6.2 Метод get\_attribute()

Для более детальный проверок в тестах нам может потребоваться получать значение атрибутов HTML. Атрибуты могут быть как стандартными, например: alt, color, placeholder и другими, так и созданными разработчиками для задания собственных стилей или правил. Подробнее ознакомиться со стандартным списком HTML атрибутов вы сможете в статье HTML Attribute Reference[[30]](#footnote-29).

Давайте посмотрим на страницу DEMOQA Links[[31]](#footnote-30) ещё раз. На этой странице есть ссылки, которые ведут на другие страницы и ссылки, которые отправляют запросы к API. Допустим, что нам необходимо проверить, что ссылка “Home” действительно содержит ссылку на домашнюю страницу сайта. Для этого посмотрим на оформление HTML-кода элемента:

<**a** id="simpleLink" href="https://demoqa.com" target="\_blank">Home</**a**>

Атрибут “href” определяет URL страницы, на которую ведёт ссылка. Чтобы получить этот или любой другой атрибут нам необходимо воспользоваться методом get\_attribute() для найденного элемента.

Давайте попробуем написать тест, который будет проверять корректность указанного URL у ссылки “Home”:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. page = 'https://demoqa.com/links'
5. browser = webdriver.Chrome()
6. browser.get(page)
8. home\_link\_locator = (By.CSS\_SELECTOR, '#simpleLink')
10. home\_link = browser.find\_element(\*home\_link\_locator)
11. home\_link\_href = home\_link.get\_attribute('href')
13. browser.quit()
15. *# Проверяем, что ссылка у ссылки именно та, что должна быть.*
16. **assert** home\_link\_href == 'https://demoqa.com/'

Хорошей практикой является написание подробного сообщения об ошибке при проверке, чтобы сразу стало понятно, что пошло не так. Давайте напишем заведомо провальный тест с подробным описанием для проверки плейсхолдера в поле ввода “Full Name” на странице DEMOQA Text Box[[32]](#footnote-31).

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. page = 'https://demoqa.com/text-box'
5. browser = webdriver.Chrome()
6. browser.get(page)
8. full\_name\_field\_locator = (By.CSS\_SELECTOR, '#userName')
10. full\_name\_field = browser.find\_element(\*full\_name\_field\_locator)
11. full\_name\_field\_placeholder = full\_name\_field.get\_attribute('placeholder')
12. expected\_full\_name\_field\_placeholder = 'Current Address'
14. browser.quit()
16. **assert** full\_name\_field\_placeholder == expected\_full\_name\_field\_placeholder, \
17. f'Expected {expected\_full\_name\_field\_placeholder=}.' \
18. f'**\n**Got: {full\_name\_field\_placeholder=}'

При запуске такого теста вы получите “AssertionError”, так как плейсхолдер будет получен не тот, что вы написали в ожидаемом результате:

1. test\_full\_name\_placeholder.py:None **(**test\_full\_name\_placeholder.py**)**
2. test\_full\_name\_placeholder.py:16: **in** **<**module**>**
3. assert full\_name\_field\_placeholder == expected\_full\_name\_field\_placeholder, \
4. E AssertionError: Expected expected\_full\_name\_field\_placeholder='Current Address'.
5. E Got: full\_name\_field\_placeholder='Full Name'
6. E assert 'Full Name' == 'Current Address'

Мы использовали форматирование строк и подробное описание ожидаемого и фактического результата в нашем тесте. Теперь, будет легко понять, что за ошибка произошла при выполнении тестового скрипта. Следует отметить, что нарочитое проваливание написанных тестов - это одна из хороших практик при написании тестовых сценариев, которой вы должны всегда придерживаться. Следует обращать внимание на то, что тест действительно проваливается при неверных входных данных и понятность сообщения об ошибке.

## 6.3 Работа с файлами, списками и js-скриптами

### 6.3.1 Работа со списками

На веб-страницах мы также встречаем раскрывающиеся (выпадающие) списки. У таких списков есть несколько важных особенностей:

* у каждого элемента списка обычно есть уникальное значение атрибута “value”;
* в списках может быть разрешено выбирать как только один, так и несколько вариантов, в зависимости от типа списка.

Давайте посмотрим как выглядит выпадающий список “Old Style Select Menu” в HTML-коде на странице DEMOQA Select Menu[[33]](#footnote-32):

<**select** id="oldSelectMenu">

<**option** value="red">Red</**option**>

<**option** value="1">Blue</**option**>

<**option** value="2">Green</**option**>

<**option** value="3">Yellow</**option**>

<**option** value="4">Purple</**option**>

<**option** value="5">Black</**option**>

<**option** value="6">White</**option**>

<**option** value="7">Voilet</**option**>

<**option** value="8">Indigo</**option**>

<**option** value="9">Magenta</**option**>

<**option** value="10">Aqua</**option**>

</**select**>

Варианты ответа задаются тегом “option”, значение “value” может отсутствовать.

Давайте воспользуемся привычным уже нам методом “click()” для выбора ответа. Для этого сперва нам нужно раскрыть выпадающий список кликнув на “select” и затем уже кликнуть на один из вариантов.

Код может быть следующим:

1. **from** time **import** sleep
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
5. page\_link = 'https://demoqa.com/select-menu'
6. browser = webdriver.Chrome()
7. browser.get(page\_link)
9. old\_style\_select\_menu\_locator = (By.CSS\_SELECTOR, '#oldSelectMenu')
10. purple\_color\_locator = (By.CSS\_SELECTOR, 'option:nth-child(5)')
12. browser.find\_element(\*old\_style\_select\_menu\_locator).click()
13. sleep(1)
14. browser.find\_element(\*purple\_color\_locator).click()
15. sleep(1)

Есть более удобный способ, для которого используется специальный класс “Select” из библиотеки “WebDriver”. Вначале мы должны инициализировать новый объект. Далее можно найти любой вариант из списка с помощью метода “select\_by\_value()”:

1. **from** time **import** sleep
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
4. **from** selenium.webdriver.support.ui **import** Select
6. page\_link = 'https://demoqa.com/select-menu'
7. browser = webdriver.Chrome()
8. browser.get(page\_link)
10. old\_style\_select\_menu\_locator = (By.CSS\_SELECTOR, '#oldSelectMenu')
11. select = Select(browser.find\_element(\*old\_style\_select\_menu\_locator))
12. select.select\_by\_value('4')
13. sleep(1)

Есть ещё два метода “select\_by\_index()” и “select\_by\_visible\_text()”. Первый ищет по индексу элемента и индексация начинается с нуля, а второй по видимому тексту.

### 6.3.2 Метод execute\_script()

Вот еще один метод, одновременно мощный и очень полезный, хотя и требующий некоторых базовых знаний JavaScript. Метод “execute\_script()” позволяет запускать код JavaScript в браузере при выполнении скрипта автоматизированного тестирования. Хотя сначала это может показаться ненужным, поскольку автоматические тесты обычно взаимодействуют с интерфейсом веб-сайта так же, как обычный пользователь, нажимая кнопки, выбирая пункты меню и вводя текст в поля, бывают ситуации, когда выполнение JavaScript может быть невероятно полезным.

Стандартные методы Selenium могут оказаться недостаточными для всех сценариев веб-приложений. Веб-сайты сильно различаются по назначению: от простых блогов до сложных финансовых или графических приложений. Разработчики имеют доступ к огромному количеству библиотек для решения бизнес-сценариев, включая нестандартные текстовые редакторы, уникальные меню и оригинальные видеоплееры на веб-страницах. В некоторых случаях тестировщикам может потребоваться написать скрипт JavaScript для взаимодействия с, казалось бы, обычными кнопками. Когда это происходит, важно обратиться за помощью к фронтэнд-разработчикам и попросить пример желаемого сценария. Тестировщики могут протестировать скрипт, выполнив его в консоли браузера перед добавлением в автотест с помощью метода “execute\_script(javascript\_code)”.

Давайте попробуем вызвать “alert” в браузере с помощью WebDriver. Пример сценария:

1. **from** time **import** sleep
2. **from** selenium **import** webdriver
4. browser = webdriver.Chrome()
5. browser.execute\_script("alert('Hello world!');")
6. sleep(2)

Очень часто вы будете сталкиваться с ситуацией, когда вам нужно кликнуть на элемент, который есть на странице, но необходимо проскроллить страницу, чтобы элемент стал видимым и скрипт сработал. Для этого вам потребуется использовать код JavaScript: “return arguments[0].scrollIntoView(true);”.

Но давайте рассмотрим пример поинтереснее. Возьмём страницу DEMOQA Practice Form[[34]](#footnote-33) и попробуем нажать на кнопку “Submit”:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. browser = webdriver.Chrome()
5. practice\_form\_page\_link = "https://demoqa.com/automation-practice-form"
6. browser.get(practice\_form\_page\_link)
8. *# browser.execute\_script('document.getElementsByTagName(\'footer\')[0].remove();', practice\_form\_page\_link)*
9. *# browser.execute\_script('document.getElementById(\'close-fixedban\').remove();')*
10. submit\_button = browser.find\_element(By.CSS\_SELECTOR, "#submit")
11. submit\_button.click()

Получим исключение “ElementClickInterceptedException”, потому что у нас на странице дефект и футер перекрывает кнопку “Submit”. Для того чтобы проверить работу формы и обойти этот дефект мы можем удалить элемент с помощью JavaScript скрипта. Раскомментируйте восьмую и девятую строчки кода и перезапустите тест.

### 6.3.3 Загрузка файлов

Если нам понадобится загрузить файл на веб-странице, мы можем использовать уже знакомый нам метод “send\_keys”. Только теперь нам нужно в качестве аргумента передать путь к нужному файлу на диске вместо простого текста.

Чтобы указать путь к файлу, можно использовать стандартный модуль Python для работы с операционной системой — os. В этом случае ваш код не будет зависеть от операционной системы, которую вы используете. Добавление файла будет работать и на Windows, и на Linux, и даже на MaсOS.

Пример кода для загрузки файла на сайт:

1. **from** os **import** path, remove
2. **from** time **import** sleep
3. **from** random **import** randint
4. **from** selenium **import** webdriver
5. **from** selenium.webdriver.common.by **import** By

8. **def** generated\_file() -> tuple[str, str]:
9. """
10. File generation.
11. :returns: file name and absolute path.
12. """
13. file\_path = f'filetest{randint(0, 999)}.txt'
14. **with** open(file\_path, 'w+', encoding='UTF-8') **as** file:
15. file.write(f'Hello, world!{randint(0, 999)}')
16. **return** file.name, path.abspath(file\_path)

19. browser = webdriver.Chrome()
20. upload\_and\_download\_page\_link = 'https://demoqa.com/upload-download'
21. browser.get(upload\_and\_download\_page\_link)
23. file\_name, file\_path = generated\_file()
25. upload\_button\_locator = (By.CSS\_SELECTOR, '#uploadFile')
26. upload\_button = browser.find\_element(\*upload\_button\_locator)
27. upload\_button.send\_keys(file\_path)
28. remove(file\_path)
29. sleep(3)

### 6.3.4 Работа с окнами

#### 6.3.4.1 Alerts

Мы уже сталкивались с разными видами уведомлений, но не писали тестовые сценарии для их проверки. Предположим нам надо проверить содержание alert-а и нажать на кнопку “ОК”, чтобы его закрыть. Для того, чтобы закрыть предупреждение, скрипт должен сначала переключиться в окно alert-а, а затем использовать команду “accept()”.

Давайте напишем небольшой тестовый сценарий для страницы DEMOQA Alerts[[35]](#footnote-34):

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. browser = webdriver.Chrome()
5. alerts\_page\_link = 'https://demoqa.com/alerts'
6. browser.get(alerts\_page\_link)
8. alert\_button\_locator = (By.CSS\_SELECTOR, '#alertButton')
9. alert\_button = browser.find\_element(\*alert\_button\_locator)
10. alert\_button.click()
11. alert = browser.switch\_to.alert
13. alert\_text = alert.text
14. expected\_alert\_text = 'You clicked a button'
15. alert.accept()
17. **assert** alert\_text == expected\_alert\_text, \
18. f'Expected alert text to be {expected\_alert\_text}.' \
19. f'**\n** Got {alert\_text} instead.'

Следующий тип модального окна - confirm отличается от alert тем, что его можно либо принять, либо отменить. Чтобы принять нужно использовать метод “accept()”, чтобы отменить - метод “dismiss()”.

Третий тип модального окна - “prompt” - имеет дополнительное поле для ввода текста. Чтобы ввести текст, нужно использовать уже знакомый нам метод “send\_keys()”.

Давайте напишем тестовый сценарий для проверки текста, который мы введём в модальном окне prompt. Сначала проделайте такой тест вручную: вызовите prompt, введите текст, нажмите кнопку “ОК”, проверьте текст расположенный слева от prompt-а, там будет введённый текст сообщения. Теперь попробуем автоматизировать этот тестовый сценарий:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. prompt\_button\_locator = (By.XPATH, '//button[@id="promtButton"]')
5. prompt\_result\_locator = (By.XPATH, '//span[@id="promptResult"]')
7. browser = webdriver.Chrome()
8. alerts\_page\_link = 'https://demoqa.com/alerts'
9. browser.get(alerts\_page\_link)
11. prompt\_button = browser.find\_element(\*prompt\_button\_locator)
12. prompt\_button.click()
14. prompt = browser.switch\_to.alert
15. input\_text\_to\_prompt = 'Hello\_world!'
16. prompt.send\_keys(input\_text\_to\_prompt)
17. prompt.accept()
19. prompt\_result = browser.find\_element(\*prompt\_result\_locator)
20. prompt\_result\_text = prompt\_result.text.split()[-1]
22. **assert** prompt\_result\_text == input\_text\_to\_prompt, \
23. f'Entered text to prompt was {input\_text\_to\_prompt}' \
24. f'**\n** Expected equal to {prompt\_result\_text=}.'

#### 6.3.4.2 Переход на новую вкладку в браузере

При работе с веб-приложениями часто встречаются ссылки, которые открываются в новой вкладке браузера. Однако WebDriver может одновременно взаимодействовать только с одной вкладкой браузера. Когда ссылка открывается в новой вкладке, WebDriver продолжит работу на старой вкладке. Чтобы переключиться на только что открытую вкладку, нам нужно явно указать, с какой вкладкой мы хотим работать. Этого можно добиться с помощью команды “switch\_to.window”: “browser.switch\_to.window(window\_name)”.

Чтобы определить имя вновь открытой вкладки, можно использовать метод “window\_handles”, который возвращает массив имен всех вкладок. Как только мы подтвердим, что две вкладки открыты, мы можем переключиться на вторую вкладку, используя соответствующую индексацию: “new\_window = browser.window\_handles[1]“.

Давайте напишем тестовый сценарий страницы DEMOQA Browser Windows для открытия новой вкладки по нажатию по кнопке “New Tab”:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. new\_tab\_button\_locator = (By.CSS\_SELECTOR, '#tabButton')
5. sample\_text\_locator = (By.CSS\_SELECTOR, '#sampleHeading')
7. browser = webdriver.Chrome()
8. browser\_windows\_page\_link = 'https://demoqa.com/browser-windows'
9. browser.get(browser\_windows\_page\_link)
11. new\_tab\_button = browser.find\_element(\*new\_tab\_button\_locator)
12. new\_tab\_button.click()
14. browser.switch\_to.window(browser.window\_handles[1])
16. sample\_text\_element = browser.find\_element(\*sample\_text\_locator)
17. actual\_sample\_text = sample\_text\_element.text
18. expected\_sample\_text = 'This is a sample page'
20. **assert** actual\_sample\_text == expected\_sample\_text, \
21. f'Expected {expected\_sample\_text=} in new tab ' \
22. f'to be equal to {actual\_sample\_text=}.'

## 6.4 Настройка ожиданий

Разработчики усердно работали над созданием красивых и быстро загружаемых веб-страниц с плавными переходами между ними.

Это было достигнуто за счет использования подхода одностраничных приложений (SPA), когда сайт имеет только одну страницу, а ее содержимое динамически обновляется с помощью JavaScript, который обменивается информацией с сервером через REST API.

Хотя этот подход обеспечивает удобство работы с пользователем, он создает проблемы для тех, кто создает UI автотесты. Непредсказуемое появление или исчезновение элементов на странице, непредсказуемое время рендеринга и изменение текста в кнопках или сообщениях — вот некоторые особенности приложений SPA, которые необходимо учитывать при автотестировании. Это один из самых сложных и запутанных аспектов разработки автотестов в Selenium (и других фреймворках для E-2-E (end-to-end) тестов).

Мы рассмотрим эти распространенные проблемы и узнаем, как их преодолеть.

### 6.4.1 Работа методов get() и find\_element()

Допустим, нам надо проверить, что на странице DEMOQA Dynamic Properties[[36]](#footnote-35) кнопку “Visible After 5 Seconds” можно кликнуть.

Тестовый сценарий:

1. зайти на страницу;
2. кликнуть по кнопке.

Открываем страницу с помощью метода “get()”, ищем элемент методом “find\_element()” и кликаем по кнопке с помощью метода “click()”.

Вот как выглядит код автотеста:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. visiable\_after\_five\_sec\_button\_locator = (By.CSS\_SELECTOR, '#visibleAfter')
6. browser = webdriver.Chrome()
7. dynamic\_properties\_page\_link = 'https://demoqa.com/dynamic-properties'
8. browser.get(dynamic\_properties\_page\_link)
10. visiable\_after\_five\_sec\_button = browser.find\_element(\*visiable\_after\_five\_sec\_button\_locator)
11. visiable\_after\_five\_sec\_button.click()

Сперва попробуйте выполнить тест вручную, а затем переходите к автотесту. Когда вы будете проходить тест самостоятельно вы заметите, что кнопка появляется с задержкой. И в случае запуска автотеста вы получите ошибку “NoSuchElementException”, потому что автотест пытается найти кнопку, которая ещё не видна.

Такая ситуация часто происходит на страницах использующих JavaScript, что очень вероятно для современных сайтов.

### 6.4.2. Использование time.sleep()

Поскольку мы знаем о задержке появления кнопки, мы можем добавить паузу в наш тестовый сценарий перед поиском элемента. Воспользуемся библиотекой “time”.

1. **from** time **import** sleep
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
5. visiable\_after\_five\_sec\_button\_locator = (By.CSS\_SELECTOR, '#visibleAfter')
7. browser = webdriver.Chrome()
8. dynamic\_properties\_page\_link = 'https://demoqa.com/dynamic-properties'
9. browser.get(dynamic\_properties\_page\_link)
11. sleep(5)
12. visiable\_after\_five\_sec\_button = browser.find\_element(\*visiable\_after\_five\_sec\_button\_locator)
13. visiable\_after\_five\_sec\_button.click()

Текущее решение добавления фиксированной задержки перед поиском элемента может быть неэффективным при обработке различных сценариев. Например, если элемент с сообщением тоже появляется с задержкой, или время задержки меняется при появлении кнопки. Точно так же скорость Интернета на разных машинах может вызывать различия во временном интервале, когда появляется кнопка. Хотя добавление задержки перед каждым действием может показаться решением, это непрактично, так как может излишне увеличить время выполнения теста. С увеличением количества тестов эта проблема будет только усугубляться.

### 6.4.3 Использование Selenium Waits (Implicit Waits)

Решение с time.sleep() плохое: оно не масштабируемое и трудно поддерживаемое.

Лучший способ справиться с ложными ошибками теста, вызванными асинхронным выполнением сценариев или задержками сервера, — реализовать решение, которое ожидает появления определенного элемента на странице в течение заданного периода времени (например, 5 секунд). Мы можем проверять наличие элемента каждые 500 миллисекунд, и как только элемент будет найден, мы можем сразу перейти к следующему шагу теста. Таким образом, мы можем получить желаемый элемент как можно скорее в лучшем случае или в течение 5 секунд в худшем случае.

Selenium WebDriver предоставляет особый подход к обработке этого ожидания, который позволяет нам установить ожидание во время инициализации драйвера и применить его ко всем тестам. Это ожидание называется неявным ожиданием, поскольку нам не нужно явно указывать его каждый раз, когда мы ищем элементы. Он будет автоматически применяться к каждой последующей команде.

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. visiable\_after\_five\_sec\_button\_locator = (By.CSS\_SELECTOR, '#visibleAfter')
6. browser = webdriver.Chrome()
7. browser.implicitly\_wait(5)
9. dynamic\_properties\_page\_link = 'https://demoqa.com/dynamic-properties'
10. browser.get(dynamic\_properties\_page\_link)
12. visiable\_after\_five\_sec\_button = browser.find\_element(\*visiable\_after\_five\_sec\_button\_locator)
13. visiable\_after\_five\_sec\_button.click()

С неявным ожиданием, установленным во время инициализации драйвера, мы можем гарантировать, что наши тесты останутся стабильными даже при незначительных задержках на веб-сайте. Для каждого экземпляра команды “find\_element” WebDriver будет ждать до 5 секунд, пока элемент появится на странице, прежде чем вызвать исключение “NoSuchElementException”.

### 6.4.4 Использование Explicit Waits (WebDriverWait и expected\_conditions)

Для обеспечения надежности нашего теста мы должны убедиться, что наша кнопка не только появилась, но и является кликабельной. На страницах часто бывает, что кнопки загружаются перед тем, как стать доступными для клика. Наш предыдущий вариант теста с неявным ожиданием не будет дожидаться до активации элемента и тест упадет с ошибкой.

Давайте улучшим наш тест с использованием явных ожиданий и ожидаемых условий:

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
3. **from** selenium.webdriver.support.ui **import** WebDriverWait **as** wait
4. **from** selenium.webdriver.support **import** expected\_conditions **as** EC
6. visiable\_after\_five\_sec\_button\_locator = (By.CSS\_SELECTOR, '#visibleAfter')
8. browser = webdriver.Chrome()
10. dynamic\_properties\_page\_link = 'https://demoqa.com/dynamic-properties'
11. browser.get(dynamic\_properties\_page\_link)
13. visiable\_after\_five\_sec\_button = wait(browser, 5).until(
14. EC.element\_to\_be\_clickable(visiable\_after\_five\_sec\_button\_locator))
15. visiable\_after\_five\_sec\_button.click()

Следует отметить, что “element\_to\_be\_clickable()” вернёт элемент, когда он будет кликабельный или False в противном случае.

Модуль “expected\_conditions” имеет несколько других условий, которые можно использовать для определения ожиданий, с ними можно ознакомиться в спецификации[[37]](#footnote-36).

# Глава 7. Тестовые фреймворки

## 7.1 Тестирование web-приложений и тестовые фреймворки

Для начала нам необходимо познакомиться с распространенными тестовыми фреймворками unittest и Pytest. Это необходимо, чтобы называть наши скрипты тестами. А главный атрибут теста - это сопоставление ожидаемого результата фактическому.

### 7.1.1 Автоматизированное и ручное тестирование

Чтобы писать хорошие автотесты необходимо понимать различия, сильные и слабые стороны ручного и автоматизированного тестирования.

Таблица 1. Сравнение автоматизированного и ручного тестирования.

| **Критерий** | **Ручное тестирование** | **Автоматизированное тестирование** |
| --- | --- | --- |
| Тестовое покрытие | Ограниченное тестовое покрытие из-за нехватки времени и ресурсов | Полное тестовое покрытие может быть достигнуто с помощью больших наборов тестов |
| Скорость выполнения | Медленное выполнение и большая трудоемкость | Более быстрое выполнение, экономия времени и усилий |
| Человеческие ошибки | Подвержены человеческим ошибкам | Менее подвержены человеческим ошибкам |
| Содержание | Сценарии ручного тестирования требуют большего обслуживания и поддержки | Сценарии автоматизированного тестирования требуют меньше обслуживания и поддержки |
| Регрессионное тестирование | Ручное регрессионное тестирование отнимает много времени и подвержено ошибкам | Автоматизированное регрессионное тестирование эффективно и надежно |
| Начальная стоимость и настройка окружения | Низкие первоначальные затраты и время настройки | Высокая начальная стоимость и время настройки |
| Гибкость | Более гибкие с точки зрения модификации тест-кейсов | Менее гибкие с точки зрения модификации тест-кейсов |
| Требования к навыкам | Не требует специальных технических навыков | Требуются специальные технические навыки |
| Когда использовать | При тестировании небольших изолированных функций или новых приложений | При тестировании больших сложных систем с повторяющимися тестовыми сценариями |

Отметим, когда лучше использовать ручное тестирование:

* для исследовательского тестирования, когда тестирование основано на интуиции, творчестве и опыте, а не на заранее определенном наборе тест-кейсов;
* для тестирования удобства использования, когда основное внимание уделяется пользовательскому опыту и требуется обратная связь;
* для тестирования небольших изолированных функций или новых приложений, где тестовые наборы ограничены, а стоимость автоматизации высока;
* когда тестирование требует участия человека, например, тестирование пользовательского интерфейса или интуитивное (ad-hoc) тестирование.

Лучше использовать автоматизированное тестирование в следующих случаях:

* при тестировании больших и сложных систем с повторяющимися тест-кейсами;
* когда требуется регрессионное тестирование, чтобы убедиться, что новые изменения не нарушают существующую функциональность;
* когда требуется тестирование производительности или нагрузочное тестирование для имитации большого количества пользователей;
* когда требуется интеграционное тестирование, чтобы убедиться, что все компоненты системы работают без сбоев;
* при тестировании требуется высокая степень точности и согласованности, например, проверка данных или тестирование безопасности.

### 7.1.2 Структура автотеста

Основная структура тестирования как для модульных, так и UI автоматизированных тестов

может быть разбита на три части:

1. Test Setup: здесь подготавливаются среда и тестовые данные. В модульных тестах это может включать настройку фиктивных объектов(mocks) или заглушек для зависимостей, а в тестах пользовательского интерфейса — настройку веб-драйвера или эмулятора.
2. Test Execution: здесь выполняется фактическая тестовая логика. В модульных тестах это может включать вызов методов и проверку вывода, тогда как в тестах пользовательского интерфейса это может включать взаимодействие с элементами пользовательского интерфейса и проверку ожидаемого поведения.
3. Test Teardown: здесь выполняются любые действия по очистке. В модульных тестах это может включать сброс состояния любых фиктивных объектов, а в тестах пользовательского интерфейса это может включать закрытие веб-драйвера или эмулятора.

Несмотря на некоторые различия в деталях реализации модульных тестов и тестов пользовательского интерфейса, общая структура тестов схожа. Оба типа тестов направлены на проверку функциональности приложения и обеспечение его правильного поведения в различных сценариях.

Давайте обсудим, как именно можно производить проверки.

### 7.1.3 Проверка ожидаемого результата

Один из способов проверить ожидаемый результат — использовать встроенный оператор “assert” в Python. Этот оператор оценивает достоверность утверждений, позволяя вам убедиться, что ваш код работает должным образом. Если ваше утверждение истинно, никаких дополнительных сообщений генерироваться не будет. Однако, если утверждение ложно, будет выдано исключение “AssertionError” с текстом, которое вы можете указать после проверки.

Посмотрим как работает “assert” на примере:

1. *# Create a list of numbers*
2. numbers = [1, 2, 3, 4, 5]
4. *# Use the built-in sum() method to add up the numbers*
5. total = sum(numbers)
7. *# Assert that the total is 15*
8. **assert** total == 15, "The sum of the numbers is incorrect"

В этом примере мы используем встроенный метод “sum()” для сложения чисел в списке чисел. Затем мы используем оператор “assert”, чтобы проверить, что сумма чисел в списке равно 15, как мы ожидаем.

Если сумма не равна 15, оператор утверждения вызовет “AssertionError” и напечатает

сообщение "The sum of the numbers is incorrect".

Вы можете изменить числа в списке, чтобы проверка оказалась не выполнена и посмотреть на работу оператора “assert” в этом случае.

Обратите внимание на то, что когда ваших тестов станет огромное количество вы уже не будете помнить, что конкретно проверяет каждый тест. И фраза "Сумма чисел некорректна" не проясняет суть проблемы и вам придется тратить время, чтобы понять что конкретно пошло не так.

Давайте попытаемся уменьшить степень неопределенности нашего теста:

1. *# Create a list of numbers*
2. numbers = [1, 2, 3, 4, 5]
4. *# Use the built-in sum() method to add up the numbers*
5. total = sum(numbers)
7. *# Assert that the total is 100500*
8. expected\_sum = 100500
10. **assert** total == expected\_sum, \
11. f"The sum of the list {numbers} is incorrect. " \
12. f"Expected: {expected\_sum}. Got: {total}."

Такой автотест легче поддерживать, если ожидаемая сумма чисел изменится в ходе разработки. Также нам сразу станет очевидна суть проблемы, когда мы прочитаем сообщение об ошибке.

### 7.1.4 Тестовые сценарии

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

1. """Making first test scripts."""

4. **def** test\_sum\_of\_list\_elements\_equals\_fifteen(input\_list: list[int]) -> None:
5. """Testing that sum of list elements is equal to 15."""
6. total = sum(input\_list)
7. **assert** total == 15, \
8. f"The sum of the list {input\_list} is incorrect. " \
9. f"Expected: 15. Got: {total}."

12. **if** \_\_name\_\_ == "\_\_main\_\_":
13. """This method works if started from that module."""
14. numbers = [1, 2, 3, 4, 5]
15. test\_sum\_of\_list\_elements\_equals\_fifteen(numbers)
16. **print**("All tests passed!")

Тест-кейс помещен в отдельную функцию, чтобы можно было независимо к нему обращаться.

В этом файле мы вызываем функцию “test\_sum\_of\_list\_elements\_equals\_fifteen()” и передаём ей в качестве аргумента наш список. Функция выполняет тестовый сценарий. Если тесты прошли успешно, то мы увидим в терминале выполнение команды “print("All tests passed!")”.

Запустите тест и посмотрите как он работает. Попробуйте изменить входные данные так, чтобы тест провалился. Доработайте тест, чтобы он выдавал достоверные данных в сообщении об ошибке при изменении входных параметров.

Добавим в файл “test\_list\_sum.py” ещё один тест и обратим внимание на поведение оператора “assert”: при провале любого теста прекращается выполнение команд интерпретатором.

Добавим следующий код:

1. """Making first test scripts."""

4. **def** test\_sum\_of\_list\_elements\_equals\_fifteen(input\_list: list[int]) -> None:
5. """Test that sum of list elements is equal to 15."""
6. total = sum(input\_list)
7. **assert** total == 15, \
8. f"The sum of the list {input\_list} is incorrect. " \
9. f"Expected: 15. Got: {total}."

12. **def** test\_number\_is\_greater\_than\_five(number: int) -> None:
13. """Test that the number is greater than 5."""
14. **assert** number > 5, \
15. f"Number {number} is not greater than 5."

18. **if** \_\_name\_\_ == "\_\_main\_\_":
19. """This method works if started from that module."""
20. big\_number = -191
21. test\_number\_is\_greater\_than\_five(big\_number)
22. numbers = [1, 2, 3, 4, 5]
23. test\_sum\_of\_list\_elements\_equals\_fifteen(numbers)
24. **print**("All tests passed!")

### 7.1.5 Выбор test runner

В предыдущих шагах мы научились писать простые тесты и запускать их с помощью Python. Запустите код из файла “test\_list\_sum.py”, который мы написали в предыдущем шаге.

Обратите внимание на вывод в консоль:

1. Traceback (most recent call last):
2. File "C:\Users\Konuhov\Desktop\diplom\graduation-project-geek-brains\7\_Test\_frameworks\test\_list\_sum.py", line 21, in <module>
3. test\_number\_is\_greater\_than\_five(big\_number)
4. File "C:\Users\Konuhov\Desktop\diplom\graduation-project-geek-brains\7\_Test\_frameworks\test\_list\_sum.py", line 14, in test\_number\_is\_greater\_than\_five
5. assert number > 5, \
6. ^^^^^^^^^^
7. AssertionError: Number -191 is not greater than 5.

Рассмотрим минусы такого подхода к запуску автотестов:

* по мере роста количества тестов становится сложно изолировать и запускать только необходимые наборы тестов;
* чтобы выполнить каждый тест, вы должны подготовить тестовые данные и среду независимо друг от друга. Например, если вы хотите открывать и закрывать браузер для каждого теста, логика работы браузера должна быть воспроизведена в коде каждого отдельного теста;
* если тест завершается ошибкой из-за ошибки AssertionError, последующие тесты выполняться не будут. Следовательно, любые потенциальные проблемы в этих тестах останутся неизвестными до тех пор, пока неудачный тест не будет устранен или пока каждый тест не будет выполнен вручную по отдельности.

Чтобы упростить процесс написания и выполнения тестов, разработчики используют специализированные фреймворки, известные как средства выполнения тестов (test runners).

Существует три основных тестовых фреймворка для Python: unittest, Pytest и Nose. Заметным преимуществом обладает модуль unittest потому что он встроенный.

Между тем, Pytest и Nose предлагают расширенные функции, которые превосходят возможности unittest. Мы сначала кратко рассмотрим использование unittest, а затем углубимся в возможности Pytest, который позволяет использовать более простой тестовый код, чем unittest, и обеспечивает гибкое выполнение тестов. Кроме того, Pytest имеет обширную библиотеку плагинов, которые могут решить практически любую проблему, связанную с автоматическим тестированием.

#### 7.1.5.1 unittest

Тест-раннеры могут автоматически обнаруживать методы тестирования в указанных файлах при условии соблюдения стандартных соглашений. Фундаментальное соглашение во всех средах тестирования заключается в том, что имя метода тестирования должно начинаться со слова “test\_”. Впоследствии может следовать любой текст для создания уникального имени теста:

1. **def** test\_specific\_meaningful\_test\_text():
2. **pass**

Для unittest существуют собственные дополнительные правила:

* тесты обязательно должны находиться в специальном тестовом классе;
* вместо assert должны использоваться специальные “assertion” методы.

Давайте теперь изменим наши предыдущие тесты, чтобы их можно было запустить с помощью unittest. Для этого нам понадобится выполнить следующие шаги:

1. импортировать unittest в файл: “import unittest”;
2. создать класс, который должен наследоваться от класса TestCase: “class TestWithUnittest(unittest.TestCase)”;
3. превратить тестовые функции в методы, добавив ссылку на экземпляр класса “self” в качестве первого аргумента функции: “def test\_sum\_of\_list\_elements\_equals\_fifteen(self, input\_list: list[int]) -> None”;
4. изменить “assert” на “self.assertEqual()”;
5. заменить строку запуска программы на “unittest.main()”.
6. """Making first test scripts."""
7. **import** unittest

10. **class** TestWithUnittest(unittest.TestCase):
11. """Test class must be inherited from TestCase."""
12. big\_number = -191
13. numbers = [1, 2, 3, 4, 5]
15. **def** test\_sum\_of\_list\_elements\_equals\_fifteen(self) -> None:
16. """Test that sum of list elements is equal to 15."""
17. input\_list = self.numbers
18. total = sum(input\_list)
19. self.assertEqual(total, 15, \
20. f"The sum of the list {input\_list} is incorrect. " \
21. f"Expected: 15. Got: {total}.")
23. **def** test\_number\_is\_greater\_than\_five(self) -> None:
24. """Test that the number is greater than 5."""
25. number = self.big\_number
26. self.assertTrue(number > 5, \
27. f"Number {number} is not greater than 5.")

30. **if** \_\_name\_\_ == "\_\_main\_\_":
31. """This method works if started from that module."""
32. unittest.main()
33. **print**("All tests passed!")

После изменений запустим нам тест с помощью Python:

Ran 2 tests in 0.005s

FAILED (failures=1)

Failure

Traceback (most recent call last):

File "C:\Users\Konuhov\Desktop\diplom\graduation-project-geek-brains\7\_Test\_frameworks\test\_list\_sum\_unittest.py", line 21, in test\_number\_is\_greater\_than\_five

self.assertTrue(number > 5, \

AssertionError: False is not true : Number -191 is not greater than 5.

Теперь мы видим более подробную информацию о результатах запуска: было запущено два теста, один тест выполняется с ошибкой. Место ошибки и пояснение к ней отображаются в логе.

## 7.2 Тестирование с помощью Pytest

### 7.2.1 Pytest - преимущества и недостатки

Установить Pytest можно использовав команду: “pip install pytest”.

Рассмотрим преимущества использования Pytest .

1. Простой и удобный в использовании: Pytest разработан, чтобы быть интуитивно понятным и простым в использовании, с простым синтаксисом для определения тестов и фикстур.
2. Мощные фикстуры: фикстуры в Pytest — это мощный инструмент для настройки тестовых данных и ресурсов, и их можно использовать в тестах и наборах тестов.
3. Полная отчетность: Pytest предоставляет подробные и исчерпывающие отчеты о результатах тестирования, включая сводки, информацию о трассировке и продолжительность тестирования.
4. Параметризованное тестирование. С помощью Pytest вы можете легко выполнять параметризованное тестирование, когда одна тестовая функция запускается с несколькими наборами входных данных.
5. Система плагинов: Pytest имеет богатую систему плагинов, которая предоставляет широкий спектр функций, таких как параллельный запуск тестов, создание отчетов в формате HTML и многое другое.
6. Поддерживает как модульное, так и функциональное тестирование: Pytest поддерживает как модульное, так и функциональное тестирование, что делает его универсальным инструментом для тестирования всех типов кода Python.
7. Лучшее обнаружение тестов: Pytest обеспечивает лучшее обнаружение тестов, чем unittest, который может автоматически находить и запускать тесты в каталоге или модуле, не требуя явных тест-кейсов.

В целом, Pytest — это мощный и гибкий инструмент тестирования, который обеспечивает множество преимуществ по сравнению с unittest, включая более простой синтаксис, более мощные инструменты, улучшенную отчетность и большую расширяемость.

Хотя у Pytest есть много преимуществ, есть и некоторые недостатки, которые следует учитывать при его использовании.

1. Кривая обучения: Pytest имеет более обширный набор функций и требует некоторых дополнительных знаний по сравнению со встроенным модулем unittest. Это может затруднить обучение новичков.
2. Настройка: гибкость Pytest может быть палкой о двух концах. Его широкие возможности настройки могут затруднить для команд поддержание согласованности в наборах тестов.
3. Управление плагинами: Pytest имеет большое количество доступных плагинов, что может быть как силой, так и слабостью. Хотя они предлагают множество полезных функций, управление плагинами может занимать много времени, и может быть сложно найти и выбрать правильный плагин для ваших нужд.
4. Отладка: отладка может быть сложной задачей в Pytest из-за его модульной природы. Иногда может быть сложно определить, какой плагин или хук вызывает проблему, что приводит к увеличению времени отладки.
5. Накладные расходы: дополнительные функции и гибкость Pytest сопряжены с некоторыми накладными расходами. В некоторых случаях тесты Pytest могут выполняться дольше, чем тесты, написанные с использованием более простых сред тестирования.

При рассмотрении вопроса о том, является ли Pytest правильным выбором для вашего проекта, важно взвесить эти недостатки и преимущества.

### 7.2.2 Фиксация пакетов

Файл “requirements.txt” — это текстовый файл, в котором вы можете перечислить необходимые пакеты Python с их версиями.

Распространенной практикой является использование файла “requirements.txt”, чтобы сделать ваш проект более переносимым и чтобы любой, кто хочет запустить ваш код, могли легко установить все зависимости.

Когда вы создаете файл “requirements.txt”, вы можете перечислить все пакеты Python, необходимые для запуска вашего проекта. Вы можете указать версию каждого пакета или использовать диапазоны версий, чтобы указать, что любая версия в пределах определенного диапазона будет приемлемой.

Чтобы создать файл “requirements.txt” для вашего проекта Python, выполните следующие действия:

1. откройте командную строку или окно терминала;
2. перейдите в каталог вашего проекта;
3. убедитесь, что ваша виртуальная среда активна;
4. выполните следующую команду, чтобы создать файл “requirements.txt”: “pip freeze > requirements.txt”.

Это создаст новый файл с именем “requirements.txt” в каталоге вашего проекта. Файл будет содержать список всех установленных пакетов и их версий.

Если вы используете виртуальные среды, рекомендуется создать новый файл требований после установки нового пакета или обновления существующего. Это гарантирует наличие актуального списка зависимостей, которые можно легко установить в других средах.

Чтобы установить зависимости из файла “requirements.txt” нужно использовать команду: “pip install -r requirements.txt”.

Все необходимые зависимости установлены одной командой!

### 7.2.3 Pytest: правила запуска тестов

Мы кратко остановимся на важнейших функциях запуска тестов с использованием Pytest.

Когда мы инициируем команду “pytest”, средство запуска тестов соберет все тесты для выполнения на основе определенных правил.

* Если мы выполним команду “pytest” без передачи каких-либо аргументов, средство запуска тестов автоматически выполнит поиск тестов в текущем каталоге.
* В качестве аргумента вы можете передать файл, путь к каталогу или любую комбинацию каталогов 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.

* Далее Pytest выполнит рекурсивный поиск по всем вложенным каталогам.
* Во время поиска Pytest проверяет все каталоги на наличие файлов, которые соответствуют соглашению об именах “test\_\*.py” или “\*test.py” (т. е. файлы, которые начинаются с “test\_” или заканчиваются на “\_test” и имеют расширение “.py”).
* В каждом из этих файлов Pytest идентифицирует тестовые функции на основе следующих правил:

1. Pytest идентифицирует все тесты, которые находятся за пределами классов и имена которых начинаются с “test”;
2. кроме того, Pytest идентифицирует все тесты в классах, имена которых начинаются с “Test” (и которые не содержат метод инициализации) и имена методов которых начинаются с “test”.

Рекомендую изучить соглашение по обнаружению тестов[[38]](#footnote-37).

### 7.2.4 Pytest - отчёты

Давайте сравним формат отчётов unittest и Pytest.

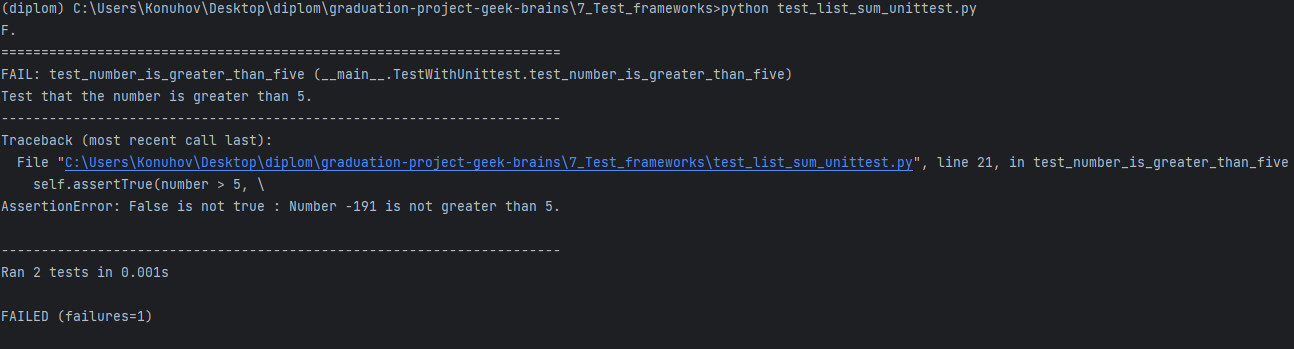


Рисунок 15. Отчёт сформированный unittest.

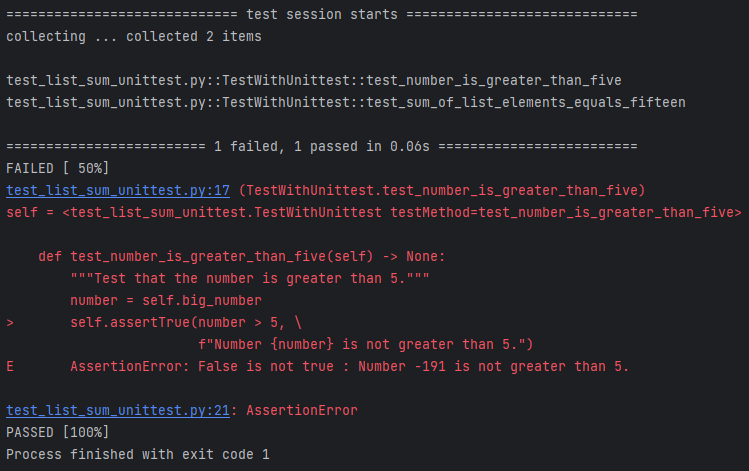


Рисунок 16. Отчёт сформированный Pytest.

При выполнении Pytest с флагом -v (verbose, подробный) отчет о тестировании включает дополнительную информацию, в которой перечислены тесты и их статус прохождения/непрохождения. Кроме того вы можете ознакомиться с другими полезными командами и функциями pytest использовав команду “pytest --help”.

### 7.2.5 Pytest - как пишут тесты.

В отличие от unittest, Pytest не требует использования дополнительных специфических конструкций в ваших тестах.

Мы уже запускали тесты, написанные в unittest стиле с помощью Pytest. Давайте перепишем их в более простом варианте, который понимает Pytest.

Назовем новый файл “test\_sum\_and\_value.py”.

1. """Making first test scripts."""

4. **class** TestWithPytest:
5. """This class provides data for tests."""
6. big\_number: int = -191
7. numbers: list[int] = [1, 2, 3, 4, 5]
9. **def** test\_sum\_of\_list\_elements\_equals\_fifteen(self) -> None:
10. """Test that sum of list elements is equal to 15."""
11. input\_list = self.numbers
12. total = sum(input\_list)
13. **assert** total == 15, \
14. f"The sum of the list {input\_list} is incorrect. " \
15. f"Expected: 15. Got: {total}."
17. **def** test\_number\_is\_greater\_than\_five(self) -> None:
18. """Test that the number is greater than 5."""
19. number = self.big\_number
20. **assert** number > 5, \
21. f"Number {number} is not greater than 5."

Запустите тест с помощью команды “pytest test\_sum\_and\_value.py” и посмотрите на формирование отчёта.

Результат должен быть такой: 1 тест пройден, 1 тест провален.

## 7.3 Использование фикстур в Pytest

### 7.3.1 Классические фикстуры

В контексте Pytest фикстуры — это вспомогательные функции для тестов, которые существуют вне самого тестового сценария.

Фикстуры служат различным целям, включая подготовку тестовой среды, очистку тестовой среды и данных после завершения теста, подключение к базам данных, создание тестовых файлов и подготовку данных в текущей среде с использованием методов API. Вы можете ознакомиться со статьёй о фикстурах в разработке программного обеспечения[[39]](#footnote-38).

Традиционный подход к использованию фикстур предполагает создание методов “setup()” и “teardown()” в тестовом файле. Вы можете найти более подробную информацию о том, как использовать эти методы в документации Pytest[[40]](#footnote-39).

Фикстуры часто создаются для определённых областей видимости, таких как модули, классы и функции, чтобы избежать повторения кода и переиспользовать его.

Давайте создадим фикстуры для инииализации браузера и его закрытия и сгруппируем тесколько тестов в тестовые наборы.

Затем рассмотрим два примера, на которых рассмотрим два разных подхода к использованию фикстур. В первом примере мы будем создавать один экземпляр браузера для тестового набора, а во втором примере - создавать по экземпляру браузера для каждого теста.

Давайте назовём наш файл “test\_fixture1.py”, запускать который мы будем командой “pytest -s test\_fixture1.py”, чтобы видеть вывод “print()” в консоль.

1. **from** selenium **import** webdriver
2. **from** selenium.webdriver.common.by **import** By
4. link = 'https://demoqa.com/'

7. **class** TestMainPage1:
9. @classmethod
10. **def** setup\_class(cls):
11. **print**('**\n**start browser for test suite..')
12. cls.browser = webdriver.Chrome()
14. @classmethod
15. **def** teardown\_class(cls):
16. **print**('quit browser for test suite..')
17. cls.browser.quit()
19. **def** test\_guest\_should\_see\_banner\_image(self):
20. self.browser.get(link)
21. self.browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
23. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self):
24. self.browser.get(link)
25. self.browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

28. **class** TestMainPage2:
30. **def** setup\_method(self):
31. **print**('start browser for test..')
32. self.browser = webdriver.Chrome()
34. **def** teardown\_method(self):
35. **print**('quit browser for test..')
36. self.browser.quit()
38. **def** test\_guest\_should\_see\_banner\_image(self):
39. self.browser.get(link)
40. self.browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
42. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self):
43. self.browser.get(link)
44. self.browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

Обратите внимание на количество запусков браузера в первом тестовом наборе и во втором. В первом наборе браузер запустился один раз, а во втором - дважды.

Запуск каждого теста в отдельном экземпляре браузера считается хорошей практикой, хоть и увеличивающей продолжительность прохождения тестов, но повышающей стабильность прохождения тестов, так как убирается взаимное влияние тестовых сценариев друг на друга.

В настоящее время используются более гибкие к применению фикстуры “@pytest.fixture”.

### 7.3.2 Фикстуры, возвращающие значение

Pytest предоставляет расширенный подход к фикстурам, где их можно определять глобально, передавать в качестве аргументов тестовым методам и даже включать в себя набор предварительно созданных фикстур. Этот подход гораздо более гибкий и удобный для работы со вспомогательными функциями, и мы сейчас рассмотрим его более подробно.

Давайте рассмотрим предыдущий пример, используя фикстуры Pytest. Мы можем создать фикстуру с именем “browser”, которая инициализирует объект WebDriver и возвращает его для использования в тестах.

Чтобы определить фикстуру, нам нужно создать метод с именем “browser” и декорировать его “@pytest.fixture”. После определения мы можем вызвать фикстуру в наших тестах, передав ее в качестве параметра. По умолчанию для каждого тестового метода создается фикстура, а это

значит, что для каждого теста будет запускаться отдельный экземпляр браузера.

1. **import** pytest
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
5. link = 'https://demoqa.com/'

8. @pytest.fixture
9. **def** browser():
10. **print**('**\n**start browser for test')
11. browser = webdriver.Chrome()
12. **return** browser

15. **class** TestMainPage1:
16. *# вызываем фикстуру в тесте, передав ее как параметр*
17. **def** test\_guest\_should\_see\_banner\_image(self, browser):
18. browser.get(link)
19. browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
21. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self, browser):
22. browser.get(link)
23. browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

### 7.3.3 Финализаторы, закрываем браузер

При выполнении тестов крайне важно закрывать экземпляры браузера. Одно правило из дзена Python[[41]](#footnote-40) гласит: “Явное лучше неявного” и не стоит всегда надеяться на сборщика мусора Python, а закрывать браузер самостоятельно.

Один из способов это сделать - использовать стандартный оператор “yield” языка Python. После завершения теста, вызвавшего фикстуру, выполнение фикстуры продолжается со строки, следующей за строкой с ключевым словом “yield”.

1. **import** pytest
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
5. link = 'https://demoqa.com/'

8. @pytest.fixture
9. **def** browser():
10. **print**('**\n**start browser')
11. browser = webdriver.Chrome()
12. **yield** browser
13. **print**('**\n**quit browser')
14. browser.quit()

17. **class** TestMainPage1:
18. **def** test\_guest\_should\_see\_banner\_image(self, browser):
19. browser.get(link)
20. browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
22. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self, browser):
23. browser.get(link)
24. browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

Вы также можете добавить финализатор согласно документации Pytest, использовав метод “addfinalizer()”[[42]](#footnote-41).

### 7.3.4 Область видимости

Фикстуры в PyTest могут быть привязаны к определенной области покрытия. Доступные значения для параметра области действия: “функция”, “класс”, “модуль” и “сеанс”. В зависимости от выбранной области фикстура будет вызываться один раз для каждой тестовой функции, один раз для каждого тестового класса, один раз для каждого тестового модуля или один раз для всего тестового сеанса.

Чтобы сэкономить время, мы можем установить область действия фикстуры браузера на “класс”, чтобы все тесты из класса “TestMainPage1” могли выполняться в одном и том же браузере.

1. **import** pytest
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
5. link = 'https://demoqa.com/'

8. @pytest.fixture(scope='class')
9. **def** browser():
10. **print**('**\n**start browser')
11. browser = webdriver.Chrome()
12. **yield** browser
13. **print**('**\n**quit browser')
14. browser.quit()

17. **class** TestMainPage1:
18. **def** test\_guest\_should\_see\_banner\_image(self, browser):
19. browser.get(link)
20. browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
22. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self, browser):
23. browser.get(link)
24. browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

В этом примере запустился один экземпляр браузера и тесты были проведены в нём последовательно друг за другом. Использовать такой подход к исполнению тестов не рекомендуется, кроме исполнения очень длительных тестов, как правило связанных с базами данных.

### 7.3.5 Автоиспользование фикстур

Вы можете указать дополнительный параметр при определении фикстуры, а именно “autouse=True”, что означает, что фикстура будет автоматически выполняться для каждого теста, не требуя явного вызова. Однако вы должны использовать этот параметр с осторожностью, так как фикстура будет выполняться для всех тестов. Поэтому рекомендуется использовать его только тогда, когда это необходимо.

## 7.4 Pytest - маркировка

### 7.4.1 Маркировка тестов часть первая

При работе с большим количеством тестов может быть полезно распределить их по категориям, а не только по именам. Вот где вступает в действие маркировка тестов. Отмечая тесты определенными категориями, такими как “smoke” для критических тестов или “regression” для регрессионных тестов, выполняемых перед выпуском, мы можем выборочно запускать тесты на основе их категории. Кроме того, у нас могут быть тесты, специфичные для определенного браузера, например Opera, и мы можем захотеть запустить эти тесты только в этом браузере.

PyTest предоставляет тестовую маркировку или метки для этой цели. Чтобы пометить тест, вы можете использовать декоратор, например “@pytest.mark.mark\_name”, где “mark\_name” — любая выбранная вами строка.

Разделим тесты в одном из предыдущих примеров на “smoke” и “regression”.

1. **import** pytest
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
5. link = 'https://demoqa.com/'

8. @pytest.fixture(scope='function')
9. **def** browser():
10. **print**('**\n**start browser')
11. browser = webdriver.Chrome()
12. **yield** browser
13. **print**('**\n**quit browser')
14. browser.quit()

17. **class** TestMainPage1:
18. @pytest.mark.smoke
19. **def** test\_guest\_should\_see\_banner\_image(self, browser):
20. **print**('smoke test')
21. browser.get(link)
22. browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
24. @pytest.mark.regression
25. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self, browser):
26. **print**('regression test')
27. browser.get(link)
28. browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

Вы можете выполнить тест с нужной меткой, передав параметр “-m”, а затем нужную метку в командной строке: “pytest -s -v -m smoke test\_fixture8.py”

Должен запуститься тест с маркировкой “smoke”. При этом вы увидите уведомление с предупреждением, что используемая метка не зарегистрирована. Рекомендуется всегда регистрировать метки, чтобы избежать опечаток при вызове определённых тестов.

### 7.4.2 Как регистрировать метки?

Создайте файл “pyproject.toml” в корневой директории вашего тестового проекта и добавьте в файл следующие строки:

1. [tool.pytest.ini\_options]
3. markers = [
4. "smoke: marker for smoke tests",
5. "regression: marker for regression tests"
6. ]

Запустите тесты повторно, предупреждений быть не должно.

Маркировать можно не только методы, но и сразу целые классы. В этом случае маркировка будет применена ко всем тестовым методам этого класса.

### 7.4.3 Маркировка тестов часть вторая

#### 7.4.3.1 Инверсия

Можно использовать логическую инверсию для запуска тестов помеченных не как “smoke”: “pytest -m "not smoke" test\_fixture8.py”.

#### 7.4.3.2 Объединение тестов с разной маркировкой

Чтобы запустить тесты с разными маркировками необходимо использовать логическое ИЛИ, например для запуска тестов с маркировкой “smoke” и “regression”: “pytest -s -v -m "smoke or regression" test\_fixture8.py”.

#### 7.4.3.3 Запуск тестов, имеющих несколько маркировок

Допустим, у вас есть несколько дымовых тестов для определённой операционной системы и вам необходимо запустить только их. Для этого вы должны маркировать тесты, которые вы хотите запустить на конкретной ОС соответствующей маркировкой. И не забываем регистрировать новые метки.

1. [tool.pytest.ini\_options]
3. markers = [
4. "smoke: marker for smoke tests",
5. "regression: marker for regression tests",
6. "win10"
7. ]
8. **import** pytest
9. **from** selenium **import** webdriver
10. **from** selenium.webdriver.common.by **import** By
12. link = 'https://demoqa.com/'

15. @pytest.fixture(scope='function')
16. **def** browser():
17. **print**('**\n**start browser')
18. browser = webdriver.Chrome()
19. **yield** browser
20. **print**('**\n**quit browser')
21. browser.quit()

24. **class** TestMainPage1:
25. @pytest.mark.smoke
26. **def** test\_guest\_should\_see\_banner\_image(self, browser):
27. **print**('smoke test 1')
28. browser.get(link)
29. browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
31. @pytest.mark.smoke
32. @pytest.mark.win10
33. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self, browser):
34. **print**('smoke test 2')
35. browser.get(link)
36. browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

Чтобы запустить только smoke-тесты для Windows 10, нужно использовать логическое И: “pytest -s -v -m "smoke and win10" test\_fixture81.py”.

### 7.4.4 Пропуск тестов

PyTest предоставляет встроенные маркеры, которые позволяют пропустить тест во время сбора тестов. Вам не нужно объявлять эти маркеры в “pyproject.toml”.

Чтобы пропустить тест необходимо отметить его как “@pytest.mark.skip”:

1. **import** pytest
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
5. link = 'https://demoqa.com/'

8. @pytest.fixture(scope='function')
9. **def** browser():
10. **print**('**\n**start browser')
11. browser = webdriver.Chrome()
12. **yield** browser
13. **print**('**\n**quit browser')
14. browser.quit()

17. **class** TestMainPage1:
18. @pytest.mark.skip(reason='Reason to skip test')
19. **def** test\_guest\_should\_see\_banner\_image(self, browser):
20. browser.get(link)
21. browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
23. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self, browser):
24. browser.get(link)
25. browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

Хорошей практикой является явное указание причины пропуска теста “@pytest.mark.skip(reason="Reason to skip test")”.

Команда для отображения всех зарегистрированных меток: “pytest --markers”.

### 7.4.5 XFail: отметить тест как ожидаемо падающий

Давайте включим тест в наш тестовый класс, который проверяет, присутствует ли кнопка “JOIN NOW”.

Допустим, должна быть кнопка “JOIN NOW”, но из-за недавних изменений кода ее больше нет. Ожидая, пока разработчики исправят проблему, мы по-прежнему хотим, чтобы все наши тесты прошли успешно, но мы хотим, чтобы неудачный тест был помечен соответствующим образом, чтобы мы могли не забыть проверить его позже. Поэтому давайте добавим маркер

“@pytest.mark.xfail” к проваленному тесту.

1. **import** pytest
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By
5. link = 'https://demoqa.com/'

8. @pytest.fixture(scope='function')
9. **def** browser():
10. browser = webdriver.Chrome()
11. **yield** browser
12. browser.quit()

15. **class** TestMainPage1:
16. **def** test\_guest\_should\_see\_banner\_image(self, browser):
17. browser.get(link)
18. browser.find\_element(By.CSS\_SELECTOR, '.banner-image')
20. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(self, browser):
21. browser.get(link)
22. browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')
24. @pytest.mark.xfail(reason="fixing this bug right now")
25. **def** test\_guest\_should\_see\_join\_now\_button\_on\_the\_main\_page(self, browser):
26. browser.get(link)
27. browser.find\_element(By.ID, 'join\_now')

Запустите тесты командой: “pytest -v test\_fixture10.py”.

Если ошибка устранена, тест, отмеченный “@pytest.mark.xfail”, теперь будет помечен как “XPASS” (неожиданно пройденный). На этом этапе можно удалить отметку “xfail”. Кроме того, к отметке “xfail” можно добавить параметр “reason”. Чтобы просмотреть это сообщение в консоли, во время запуска необходимо добавить параметр “pytest -rx”.

Дополнительно об использовании меток об неожиданном прохождении или падении вы можете прочитать в документации[[43]](#footnote-42).

## 7.5 Pytest - параметризация, конфигурирование, плагины

### 7.5.1 Conftest.py - конфигурирование тестов

В предыдущем разделе мы создали фикстуру браузера “browser”, которая создает экземпляр браузера для тестирования в текущем файле.

Однако, когда у нас есть несколько тестовых файлов, может быть неудобно переопределять эту фикстуру в каждом файле.

Чтобы избежать этого повторения и сохранить часто используемые фикстуры и глобальные настройки, мы можем использовать файл “conftest.py”, который должен находиться в каталоге верхнего уровня нашего тестового проекта.

Хотя можно создавать дополнительные файлы “conftest.py” в других каталогах, любые настройки, определенные в этих файлах, будут применяться только к тестам в подкаталогах.

Мы можем создать файл “conftest.py” в каталоге верхнего уровня нашего тестового проекта и переместить туда фикстуру браузера для хранения часто используемых фикстур и глобальных настроек. Благодаря этому файл с тестами становится более лаконичным.

conftest.py:

1. **import** pytest
2. **from** selenium **import** webdriver

5. @pytest.fixture(scope='function')
6. **def** browser():
7. browser = webdriver.Chrome()
8. **yield** browser
9. browser.quit()

С добавлением файла “conftest.py” в корневой каталог тестового проекта доступ к фикстуре браузера теперь возможен для всех тестовых файлов.

Фикстура передается тестовому методу в качестве аргумента, что позволяет легко повторно использовать вспомогательные функции в разных частях проекта, независимо от того, сколько тестовых файлов создано.

1. **from** selenium.webdriver.common.by **import** By
3. link = 'https://demoqa.com/'

6. **def** test\_guest\_should\_see\_banner\_image(browser):
7. browser.get(link)
8. browser.find\_element(By.CSS\_SELECTOR, '.banner-image')

Крайне важно понять важный аспект поведения файлов конфигурации. Pytest автоматически обнаруживает и загружает файлы “conftest.py” в каталог с тестами. Если у вас все тестовые сценарии в одной папке, будьте осторожны и не запускайте тесты из папки с тестами, так как это может помешать автоматическому обнаружению файлов “conftest.py”:

tests/

├── conftest.py

├── subfolder

│ └── conftest.py

│ └── test\_abs.py

следует избегать!

Если тесты запускаются из каталога, содержащего два файла “conftest.py”, будут применены оба файла, что может привести к непредвиденным ошибкам и конфликтам.

Этот метод можно использовать для переопределения различных фикстур, но для знакомства рекомендуется придерживаться одного файла для каждого проекта/задачи и сохранять их в горизонтальном порядке, например:

selenium\_course\_solutions/

├── section3

│ └── conftest.py

│ └── test\_languages.py

├── section4

│ └── conftest.py

│ └── test\_main\_page.py

правильно!

Подробнее можете ознакомиться в документации[[44]](#footnote-43).

### 7.5.2 Параметризация тестов

Используя декоратор “@pytest.mark.parametrize()” в Pytest, вы можете запустить один и тот же тест с различными входными параметрами. Например, предположим, что наш веб-сайт доступен на разных языках. Мы можем написать тест, который проверяет, отображается ли ссылка на форму входа для русской и английской версий главной страницы сайта. Мы можем

передать в наш тест ссылки на русскую и английскую версии главной страницы сайта.

Чтобы использовать декоратор “@pytest.mark.parametrize()”, вам необходимо указать параметр, который необходимо изменить, и список значений параметров. Тест также должен передавать параметр в качестве аргумента. Важно отметить, что при использовании декоратора имя параметра заключается в кавычки, а в списке тестовых аргументов кавычки не нужны.

Далее мы немного поработаем с сайтом Oscar - Sandbox[[45]](#footnote-44).

1. **import** pytest
2. **from** selenium **import** webdriver
3. **from** selenium.webdriver.common.by **import** By

6. @pytest.fixture(scope='function')
7. **def** browser():
8. browser = webdriver.Chrome()
9. **yield** browser
10. browser.quit()

13. @pytest.mark.parametrize('language', ['ru', 'en-gb'])
14. **def** test\_guest\_should\_see\_login\_link(browser, language):
15. link = f'http://selenium1py.pythonanywhere.com/{language}/'
16. browser.get(link)
17. browser.find\_element(By.CSS\_SELECTOR, '#login\_link')

Запустите тест командой “pytest -v test\_fixture7.py”.

Вы увидите, что запустилось два теста с разным параметром, заключённым в квадратные скобки. Параметризация помогает запускать тесты с большим количеством входных данных в одном сценарии без дублирования кода.

Можно параметризировать не только функции, но и целые классы. В таком случае параметры будут использованы для всех функций, которые расположены внутри класса.

Дополнительно ознакомьтесь с документацией о том как правильно использовать параметризацию в Pytest[[46]](#footnote-45).

### 7.5.3 Установка Firefox и Selenium-драйвера geckodriver

До сих пор мы выполняли наши тесты только в браузере Chrome. Однако что, если мы хотим протестировать наше веб-приложение и в других браузерах? В таком сценарии мы можем запускать одни и те же тесты в разных браузерах, указав имя браузера при запуске тестов. Например, мы можем выбрать Firefox в качестве второго браузера, поскольку он широко используется и может работать на любой платформе. Чтобы запустить тесты в конкретном браузере, нам нужно указать параметр “browser\_name” при запуске с помощью следующей команды: “pytest -s -v --browser\_name=firefox test\_cmd.py”.

Установите последнюю версию браузера Firefox[[47]](#footnote-46) для вашей платформы.

Selenium-драйвер для браузера Firefox называется geckodriver, его вам стоит скачать из официального репозитория[[48]](#footnote-47) компании Mozilla. Ознакомьтесь с инструкцией по установке драйвера[[49]](#footnote-48), если у вас возникнут сложности.

### 7.5.4 Conftest.py и передача параметров в командной строке

На этом этапе мы разберемся, как настраивать тестовые среды, передавая параметры через командную строку с помощью встроенной фикстуры “request”, которая может получать данные о текущем выполняемом тесте. Это может позволить нам хранить дополнительные данные в отчете и выполнять ряд других полезных задач.

Это делается с помощью встроенной функции “pytest\_addoption()” и фикстуры “request”. Сначала добавляем в файле “conftest.py” обработчик опции в функции “pytest\_addoption()”, затем напишем фикстуру, которая будет обрабатывать переданные в опции данные.

Подробности использования можно узнать в документации[[50]](#footnote-49).

Добавим логику обработки командной строки в файл “conftest.py”.

1. **import** pytest
2. **from** selenium **import** webdriver

5. **def** pytest\_addoption(parser):
6. parser.addoption('--browser\_name', action='store', default=None,
7. help='Choose browser: chrome or firefox')

10. @pytest.fixture(scope='function')
11. **def** browser(request):
12. browser\_name = request.config.getoption('browser\_name')
13. browser = None
14. **if** browser\_name == 'chrome':
15. **print**('**\n**start chrome browser')
16. browser = webdriver.Chrome()
17. **elif** browser\_name == 'firefox':
18. **print**('**\n**start firefox browser')
19. browser = webdriver.Firefox()
20. **else**:
21. **raise** pytest.UsageError('--browser\_name should be chrome or firefox')
22. **yield** browser
23. **print**('**\n**quit browser')
24. browser.quit()

Запустите следующий тест без параметра командой “pytest -s -v test\_parser.py”:

1. **from** selenium.webdriver.common.by **import** By
3. link = 'http://selenium1py.pythonanywhere.com/'

6. **def** test\_guest\_should\_see\_login\_link(browser):
7. browser.get(link)
8. browser.find\_element(By.CSS\_SELECTOR, '#login\_link')

Вы получите ошибку “ERROR \_pytest.config.exceptions.UsageError: --browser\_name should be chrome or firefox”.

Чтобы не вводить параметр браузера можно использовать значение по умолчанию. Для этого необходимо добавить “default='chrome'” в “conftest.py”.

Запустите тест в браузере Chrome: “pytest -s -v --browser\_name=chrome test\_parser.py” и в Firefox: “pytest -s -v --browser\_name=firefox test\_parser.py”.

### 7.5.5 Плагины и перезапуск тестов

Pytest имеет богатый выбор плагинов для расширений возможностей фреймворка. Полный список доступных фреймворков можно найти в документации[[51]](#footnote-50).

Обсудим еще одну распространенную проблему, часто возникающую при написании сквозных тестов в Selenium, — flaky-тесты или “мигающие” автотесты. Это тесты, которые иногда проваливаются из-за внешних факторов, не зависящих от нас, или из-за трудно воспроизводимых ошибок, хотя в большинстве случаев они успешно проходят. Это может произойти во время тестов из-за одновременных обновлений сайта, проблем с сетью или странных совпадений. Несмотря на то, что устранение таких проблем и выявление причин ошибок необходимо, в реальном мире это часто требует значительных усилий. Таким образом, чтобы убедиться, что тест действительно обнаружил ошибку и не произошел случайный сбой, мы можем повторить неудачный тест.

Реализация этого — простой процесс. Мы будем использовать плагин pytest-rerunfailures.

Нам понадобится установить плагин в виртуальную среду и, соответственно, добавить плагин в наши зависимости в файл “requirements.txt”. Установить плагин можно командой “pip install pytest-rerunfailures”.

Чтобы установить количество повторных запусков неудачных тестов, вы можете добавить параметр командной строки: “--reruns n”, где “n” представляет количество повторных запусков. Если тесты пройдены на любом из повторных запусков, тестовый запуск будет считаться успешным, а количество повторных запусков будет отображаться в отчёте для дальнейшего анализа проблемных тестов.

Дополнительно используем параметр “--tb=line”, чтобы сократить вывод лога: “pytest -v --tb=line --reruns 1 --browser\_name=chrome test\_rerun.py”.

Давайте запустим два теста, один из которых будет выполняться успешно, а второй будет падать с ошибкой, и посмотрим как выполняется перезапуск тестов:

1. **from** selenium.webdriver.common.by **import** By
3. link = 'https://demoqa.com/'

6. **def** test\_guest\_should\_see\_elements\_card\_on\_the\_main\_page(browser):
7. browser.get(link)
8. browser.find\_element(By.XPATH, '//\*[@class="card mt-4 top-card"][.//h5[text()="Elements"]]')

11. **def** test\_guest\_should\_see\_join\_now\_button\_on\_the\_main\_page(browser):
12. browser.get(link)
13. browser.find\_element(By.CSS\_SELECTOR, '#join\_now')

Вы увидите отчет о запуске тестов: "1 failed, 1 passed, 1 rerun in 50.16s", следовательно первоначально провалившийся тест был перезапущен.

# Глава 8. Использование Page Object Model

## 8.1 Что такое Page Object?

В этой мы будем использовать принципы объектно-ориентированного программирования и вам не помешает освежить в памяти основные принципы ООП[[52]](#footnote-51).

### 8.1.1 Code Style

В сообществе программистов существует широко распространенное соглашение, известное как “стиль кода” (Code Style). Стиль кода охватывает все аспекты, не связанные с функциональностью самого кода, такие как форматирование, имена переменных и функций, и другие подобные соображения. Python, в частности, хвалят за его удобочитаемость, но даже этот язык может превратиться в нечитаемый беспорядок, если его не поддерживать должным образом. Нечитаемый код представляет опасность, поскольку становится трудным для понимания и может вызвать путаницу в будущем как у вас, так и у ваших коллег. С другой стороны, хорошо написанный код предлагает множество преимуществ, включая экономию времени при исправлении ошибок, более плавную адаптацию новых членов команды и более эффективную разработку кода в целом. Поэтому поддержание читаемости кода всегда должно быть одним из приоритетов.

Вам обязательно стоит прочитать рекомендации по написанию кода[[53]](#footnote-52) и сопроводительной документации[[54]](#footnote-53) для языка Python. В предыдущих частях мы кратко обсуждали эту тему, а теперь более уместно углубиться в неё более подробно и рассмотрим некоторые важные аспекты.

#### 8.1.1.1 Отступы

Отступы являются важным аспектом синтаксиса Python, поскольку они означают вложенность блоков, таких как тело условного оператора или цикла. Важно отметить, что на следующих шагах должны поддерживаться правильные отступы для всех функций внутри класса.

1. *# Тест вне класса, отступ не нужен.*
2. **def** test\_equal(a, b):
3. **assert** a == b, f'Expected {a=} to be equal {b=}'

6. **class** TestInteractionsPage:
7. """Class represents Interactions tab.
8. Sortable
9. Selectable,
10. Resizable,
11. Droppable
12. """
14. *# Класс внутри класса. Необходим отступ, чтобы показать вложенность.*
15. **class** TestSortablePage:
16. """Class represents Sortable tab tests."""
17. sortable\_page\_link = 'https://demoqa.com/sortable'
19. *# Тест внутри вложенного класса. Необходим отступ.*
20. **def** test\_order\_of\_list\_can\_be\_changed(self, driver):
21. """Test order of list can be changed."""
22. sortable\_page = SortablePage(driver, self.sortable\_page\_link)
23. sortable\_page.open()
24. order\_of\_list\_before = sortable\_page.get\_items\_of\_list()
25. sortable\_page.change\_list\_order()
26. order\_of\_list\_after = sortable\_page.get\_items\_of\_list()
27. **assert** order\_of\_list\_before != order\_of\_list\_after, \
28. 'Expected order of list to be different.' \
29. f'Order before: {order\_of\_list\_before}' \
30. f'Order after: {order\_of\_list\_after}'

#### 8.1.1.2 Имена переменных и финкций

Именование — важнейший элемент написания читаемого кода. Будь то объявление переменных, определение функций или присвоение имен классам, важно убедиться, что назначенные имена имеют смысл и точно отражают суть соответствующих объектов. Избегайте использования однобуквенных или расплывчатых имен, таких как “var1”, “x”, “y”, “my\_function”, “class2” и т. д.

Цель состоит в том, чтобы создать код, который говорит сам за себя и не требует дополнительных объяснений. Если вы обнаружите, что хотите написать поясняющий комментарий, это хорошая возможность реорганизовать код и устранить необходимость в комментариях за счет улучшения соглашений об именах.

Как правило, компании имеют свои внутренние соглашения относительно именования переменных. Однако общие отраслевые правила именования переменных относительно одинаковы в разных организациях.

Функции и методы пишутся с помощью “snake\_case”:

1. **def** test\_revertable\_draggable\_will\_return\_to\_current\_position\_after\_dragging(self, driver):
2. **pass**
4. **def** switch\_to\_frame(self, frame\_locator, timeout=5):
5. **pass**

Классы пишут с помощью “CamelCase”:

1. **class** AccordianPage(BasePage):
2. **pass**
4. **class** TestAccordianPage:
5. **pass**

Константы пишут в стиле “UPPERCASE”:

1. NO\_ERRORS\_TEXT = 'No errors'

#### 8.1.1.3 Максимальная простота кода

Распространенные принципы DRY[[55]](#footnote-54) (Don’t repeat yourself) и KISS[[56]](#footnote-55) (Keep it simple, stupid).

Рассмотрим основные моменты хороших практик и принципов программирования:

* Старайтесь писать как можно более простой код, избегая ненужной сложности.
* Сведите к минимуму использование сложных конструкций, таких как лямбда-выражения, карты и другие сложные методы, если только они не являются действительно необходимыми.
* Всякий раз, когда у вас есть возможность заменить часть кода более простой альтернативой, сделайте это.
* Подчеркните линейную структуру кода, так как ее легче понять и следовать.
* Избегайте чрезмерной вложенности блоков кода, так как это может затруднить чтение кода.
* По возможности избегайте дублирования логики. Если вы заметили повторяющиеся шаблоны, выделите эту логику в отдельный компонент или функцию, чтобы избежать повторения.
* По возможности предпочтите явный код неявному коду. Лучше иметь четкий и понятный код, чем полагаться на скрытое или магическое поведение.

### 8.1.2 Code Style в автотестах

Важные принципы написания автотестов:

* Стремитесь поддерживать высокий уровень линейности в своем тестовом коде, сводя к минимуму использование ветвлений и циклов в тестовых примерах. Если вы обнаружите, что вам нужно ввести условные операторы (например, оператор “if”) в свои тесты, рассмотрите возможность разделения теста на два отдельных теста или внесите изменение тестовой среды, чтобы исключить необходимость ветвления.
  + Уменьшая число ветвлений и сохраняя линейность тестового кода, вы можете улучшить читаемость и удобство сопровождения ваших тестов, упрощая их понимание и устранение неполадок.
* Сведите к минимуму использование зависимых тестов, основанных на определенном порядке выполнения. По мере расширения набора тестов вы можете захотеть запускать тесты одновременно в нескольких потоках, что усложняется при использовании зависимых тестов. Кроме того, зависимые тесты могут привести к проблемам с надежностью.
  + Избегая зависимостей между тестами, вы обеспечиваете большее распараллеливание и повысите общую надежность набора тестов. Каждый тест должен быть автономным и независимым, что обеспечивает гибкость выполнения и более точные результаты тестирования. Подробнее о зависимостях в тестах вы можете прочитать в статье “...почему зависимости между тестами это плохо?”[[57]](#footnote-56).
* Стремитесь сделать ваши тесты самодостаточными и независимыми от конкретного контента. Вместо этого сосредоточьтесь на подготовке необходимых данных в самом тесте и обеспечьте надлежащую очистку после завершения теста. Рекомендуется использовать чистые браузеры и создавать новые учетные записи пользователей для повышения воспроизводимости.
  + Генерируя необходимые данные и управляя тестовой средой в тестовом коде, вы уменьшаете зависимость от внешних факторов, делая ваши тесты более надежными и воспроизводимыми при различных исполнениях. Такой подход обеспечивает лучшую изоляцию и повышает общую стабильность тестового набора.
* Названия тестов должны соответствовать единому стилю и эффективно отражать различия между похожими сценариями. Подобно добавлению имен к тест-кейсам в тестовой документации, вы можете использовать аналогичные подходы для именования своих тестов.
  + Важно выбрать описательные и осмысленные имена для ваших тестов, подчеркивая конкретные различия или тестируемые аспекты. Следуя единому стилю именования, вы повышаете удобочитаемость и понятность своего набора тестов. Кроме того, понятные и информативные названия тестов облегчают эффективную коммуникацию и сотрудничество между членами команды.
* Вместо того, чтобы дублировать идентичные тесты с небольшими различиями в содержании, например, в языке интерфейса, рекомендуется их параметризовать. Параметризация позволяет повторно использовать одну и ту же тестовую логику, предоставляя разные входные значения.
  + Параметризируя тесты, вы можете определить один тестовый пример и указать различное содержимое в качестве параметров. Такой подход устраняет дублирование кода, повышает удобство сопровождения и упрощает процесс добавления новых тест-кейсов с аналогичными характеристиками.
* Рекомендуется писать тесты, которые являются атомарными и неделимыми, то есть каждый тест фокусируется на определенной функциональности или сценарии. Вместо того, чтобы создавать один большой тест, который проверяет сразу несколько аспектов, лучше разбить его на более мелкие, более целенаправленные тесты. Такой подход позволяет лучше локализовать проблему и упрощает устранение неполадок при их возникновении.
  + Сохраняя тесты атомарными, вы можете точно определить местонахождение сбоев и определить конкретную функциональность, вызывающую проблему. Это также способствует модульности и возможности повторного использования, поскольку тесты меньшего размера можно комбинировать или повторно использовать в разных сценариях.

Рекомендую ознакомится с практиками написания “хорошего” кода в статьях “Что такое красивый код, и как его писать?”[[58]](#footnote-57) и “Практика хорошего кода”[[59]](#footnote-58).

### 8.1.3 Подготовка окружения

Целью этого дипломного проекта является обучение основам автоматизированного тестирования UI и, чтобы приблизиться к достижению этой цели, вы должны практиковаться.

Мы создадим отдельный публичный репозиторий и будем постепенно добавлять и изменять код автотестов, чтобы сохранить историю коммитов и приблизиться к промышленной разработке.

Начнём:

1. Создайте отдельный публичный репозиторий с осмысленным названием на GitHub.
2. Добавьте в свой репозиторий файлы “README.md” и Python “.gitignore”.
3. Клонируйте его к себе на локальную машину.
4. Добавьте туда файл “conftest.py” из предыдущего модуля. Убедитесь дополнительно, что там есть параметр для задания языка интерфейса, по умолчанию равный “en”. Вам придется реализовать выбор локализации самостоятельно. Знаний, полученных в предыдущих модулях вполне достаточно.
5. Убедитесь что ни во вложенных папках, ни во внешних папках нет других файлов “conftest.py”.
6. Добавьте в репозиторий файл “requirements.txt”, добавьте туда зависимости и установите их в новое виртуальное окружение для проекта.
7. Создайте пустой файл “\_\_init\_\_.py”, чтобы работали относительные импорты.
8. Создайте файл “test\_main\_page.py” и добавьте в него тест:
9. **def** test\_guest\_can\_go\_to\_login\_page(browser):
10. link = 'http://selenium1py.pythonanywhere.com/'
11. browser.get(link)
12. login\_link = browser.find\_element(By.CSS\_SELECTOR, '#login\_link')
13. login\_link.click()
15. Убедитесь, что тест работает, с помощью следующей команды: “pytest -v --tb=line --language=en test\_main\_page.py”.
16. Добавьте в “README.md” небольшое описание тестового проекта.
17. Зафиксируйте изменения коммитом с осмысленным сообщением.

### 8.1.4 Что такое Page Object Model?

Page Object Model (Page Object) — это широко используемый шаблон программирования в автоматизации тестирования, особенно для тестирования веб-продуктов. Он служит стандартным подходом и предлагает удобный способ структурирования кода, облегчая его обслуживание, модификацию и удобство использования.

Фундаментальная концепция модели Page Object Model заключается в представлении каждой веб-страницы приложения в виде объекта класса. Взаимодействие со страницей можно описать с помощью методов внутри класса. В идеале тесты, использующие Page Objects, должны быть сосредоточены на бизнес-логике тестового сценария, абстрагируясь от методов Selenium для взаимодействия браузера и страницы. Такой подход упрощает обслуживание, поскольку изменения макета страницы не потребуют модификации связанных тестов. Вместо этого необходимо обновить только класс, представляющий страницу.

К объектной модели страницы применяются те же принципы разработки кода: улучшение читаемости кода и инкапсуляция деталей в абстрактные методы. Тесты должны быть написаны просто и понятно, с выделением повторяющихся сегментов кода в отдельные функции. Внутри Page Object мы отличаем логику действий, таких как аутентификация пользователя, от конкретных деталей реализации (например, поиск поля электронной почты, ввод данных, поиск поля пароля, ввод данных, расположение кнопки и т. д.).

Рассмотрим такой простой тест-кейс:

1. открыть главную страницу;

2. Перейти на страницу логина.

Ожидаемый результат: открыта страница логина.

Давайте посмотрим на кода теста, который реализует первую часть этого теста “test\_main\_page.py”:

1. link = 'http://selenium1py.pythonanywhere.com/'

4. **def** test\_guest\_can\_go\_to\_login\_page(browser):
5. browser.get(link)
6. login\_link = browser.find\_element(By.CSS\_SELECTOR, '#login\_link')
7. login\_link.click()

Что здесь происходит?

Открывается ссылка, находится элемент с определенным селектором и производится клик на этот элемент.

Что мы на самом деле имеем в виду?

Мы хотим открыть страницу логина. Давайте выделим это действие в отдельную функцию с понятным названием, пока все в том же файле “test\_main\_page.py”:

1. **def** go\_to\_login\_page(browser):
2. login\_link = browser.find\_element(By.CSS\_SELECTOR, '#login\_link')
3. login\_link.click()

И наш тест упрощается:

1. **def** test\_guest\_can\_go\_to\_login\_page(browser):
2. browser.get(link)
3. go\_to\_login\_page(browser)

При написании последующих тестов, требующих перехода на страницу входа с главной страницы, нет необходимости дублировать или переписывать код. Вместо этого мы можем повторно использовать ранее написанный метод, гарантируя возможность повторного использования кода и уменьшая избыточность.

### 8.1.5 Почему именно Page Object?

Хотя можно хранить всю тестовую логику в одном файле, таком как «steps.py», этот подход может стать проблематичным по мере увеличения размера веб-продукта и сложности его состояний и переходов. Файл может стать слишком большим, что затруднит поиск определенных методов. Кроме того, бывают случаи, когда один и тот же метод может иметь разные реализации на разных страницах. Например, в нашем интернет-магазине функция “добавить в корзину” доступна как со страницы каталога, так и со страницы отдельного продукта.

Чтобы улучшить организацию и удобство сопровождения, полезно сгруппировать методы, которые логически связаны с конкретной веб-страницей, в выделенный класс в нашем коде. Эта концепция приводит к термину Page Object, который относится к абстрактному объекту, инкапсулирующему методы для взаимодействия с конкретной веб-страницей.

Важно! Обычно методы у Page Object бывают двух типов: сделать что-то и проверить что-то.

Рассмотрим страницу товара[[60]](#footnote-59) в интернет магазине.

Какие могут быть методы у Page Object, ассоциированного с такой страницей?

Запишем основные сценарии:

* добавить в корзину;
* проверить, что есть сообщение об успешном добавлении в корзину;
* перейти к написанию отзыва;
* проверить, что есть название, цена, описание товара;
* вернуться на главную.

Стоит отметить, что все утверждения также реализованы в виде отдельных методов. В самом тестовом примере нет вспомогательных ключевых слов, таких как “assert”, а скорее описательное представление шагов, аналогичное нашей тестовой документации.

Тесты будут выглядеть примерно так:

1. **def** test\_add\_to\_cart(browser):
2. product\_page = ProductPage(browser, url='') *# инициализируем объект Page Object*
3. product\_page.open() *# открываем страницу в браузере*
4. product\_page.should\_be\_add\_to\_cart\_button() *# проверяем что есть кнопка добавления в корзину*
5. product\_page.add\_product\_to\_cart() *# жмем кнопку добавить в корзину*
6. product\_page.should\_be\_success\_message() *# проверяем что есть сообщение с нужным текстом*

## 8.2 Первые тесты с применением Page Model

### 8.2.1 Базовая страница для проекта: BasePage

Теперь мы проведём рефакторинг теста в файле “test\_main\_page.py” который мы представили

на этапе настройки среды, используя шаблон Page Object. Наше внимание будет сосредоточено на главной странице нашего приложения, и для этого мы дадим классу говорящее имя, например "MainPage".

Примечание. Мы создадим базовую реализацию шаблона Page Object с нуля. Сейчас крайне важно сосредоточиться на понимании основных принципов этого шаблона.

1. Создайте в своем проекте директорию pages, там мы будем хранить все наши Page Object.
2. В папке создайте два файла: base\_page.py и main\_page.py. Для начала мы создадим базовый класс страницы, который будет служить основой для всех остальных классов. Эта базовая страница будет содержать служебные методы для взаимодействия с драйвером.
3. В файле base\_page.py создайте класс с названием BasePage.
4. Далее мы добавим методы в наш класс. Прежде всего, давайте включим конструктор, который является методом, вызываемым при создании объекта. Конструктор определяется с помощью ключевого слова \_\_init\_\_. Мы передадим экземпляр драйвера и URL-адрес в качестве параметров конструктору. Внутри конструктора мы будем хранить эти данные как атрибуты нашего класса.
5. **def** \_\_init\_\_(self, browser, url):
6. self.browser = browser
7. self.url = url
9. Реализуем метод открытия страницы, путём вызовы метода “get()”:
10. **def open(self):**
11. **pass**

Вам необходимо самостоятельно реализовать этот метод. В результате в файле “base\_page.py” у вас должно быть следующее:

1. **class** BasePage:
2. **def** \_\_init\_\_(self, browser, url):
3. self.browser = browser
4. self.url = url
6. **def** open(self):
7. *# ваша реализация*

### 

### 8.2.2 Page Object для главной страницы сайта

Теперь приступим к реализации Page Object, который будет связан с главной страницей интернет-магазина.

1. Откройте файл “main\_page.py”.
2. В нем нужно сделать импорт базового класса BasePage: “from .base\_page import BasePage”.
3. В нём создайте класс MainPage. Наследуйте его от класса BasePage, чтобы он получил все его методы и атрибуты: “class MainPage(BasePage):”.
4. Перенесите метод открытия страницы логина в класс MainPage:
5. **def** go\_to\_login\_page(browser):
6. login\_link = browser.find\_element(By.CSS\_SELECTOR, '#login\_link')
7. login\_link.click()

Поскольку экземпляр браузера мы создаём и передаём на этапе создания Page Object, то нам больше не нужно передавать его в качестве аргумента. Вместо этого необходимо использовать аргумент “self”, чтобы иметь доступ к методам и атрибутам класса.

Так как браузер - это аргумент класса BasePage, то обращаться к нему нужно через “self”:

1. self.browser.find\_element(By.CSS\_SELECTOR, '#login\_link')

В итоге, ваш файл “main\_page.py” должен выглядеть следующим образом:

1. **from** .base\_page **import** BasePage
2. **from** selenium.webdriver.common.by **import** By

5. **class** MainPage(BasePage):
6. **def** go\_to\_login\_page(self):
7. login\_link = self.browser.find\_element(By.CSS\_SELECTOR, '#login\_link')
8. login\_link.click()

### 8.2.3 Первый тест на основе Page Object

Давайте теперь перепишем тест с помощью Page Object:

1. Создайте файл для тестов главной страницы, поместите его на уровне с “conftest.py” и “requirements.txt”, то есть в корневом каталоге вашего проекта. Назовём его “test\_main\_page.py”.
2. Откройте файл “test\_main\_page.py”.
3. Импортируйте класс главной страницы из директории “pages”:
4. **from** .pages.main\_page **import** MainPage
6. Теперь отрефакторим сам тест:
7. **from** .pages.main\_page **import** MainPage

10. **def** test\_guest\_can\_go\_to\_login\_page(browser):
11. link = 'http://selenium1py.pythonanywhere.com/'
12. page = MainPage(browser, link) *# инициализация Page Object*
13. page.open() *# открыть страницу*
14. page.go\_to\_login\_page() *# перейти на страницу логина*
16. Запустите тест, чтобы убедиться, что он работает. Можете использовать любую команду для запуска, например “pytest -v --tb=line --language=en test\_main\_page.py”.
17. Не забывайте коммитить вносимые изменения (обязательно с осмысленными сообщениями о том, что изменилось. Рекомендую использовать только английский язык как для сообщений в коммитах, так и в пул реквестах, описаниях, комментариях и докстрингах).

Теперь наш тест почти что полность написан согласно стилю Page Object. Почему почти - узнаете чуть позже.

### 8.2.4 Методы-проверки в Page Object

Давайте автоматизируем следующий сценарий:

1. Открыть главную страницу
2. Проверить, что есть ссылка, которая ведет на логин

Как нам реализовать проверку наличия ссылки? Во первых, это будет один из методов страницы MainPage, потому что по условию мы должны проводить эту проверку на этой странице. Во-вторых, методы-проверки обычно называются похожим образом и могут выглядеть примерно так: “should\_be\_…”.

Давайте создадим в нашем классе MainPage метод “should\_be\_login\_link()”:

1. **def** should\_be\_login\_link(self):
2. self.browser.find\_element(By.CSS\_SELECTOR, '#login\_link\_invalid')

Обратите внимание, мы намеренно сделали селектор неправильным, чтобы наблюдать за результатом теста, когда он сталкивается с ошибкой. Хорошей практикой считается сначала писать тесты, которые проваливаются, а затем делать их проходящими.

Добавим новый тест в наш тестовый набор для главной страницы:

1. **def** test\_guest\_should\_see\_login\_link(browser):
2. link = 'http://selenium1py.pythonanywhere.com/'
3. page = MainPage(browser, link)
4. page.open()
5. page.should\_be\_login\_link()

Запустите получившийся тест командой: “pytest -v --tb=line --language=en test\_main\_page.py”.

Вывод сообщения об ошибке не очень понятный и требуется время, чтобы разобраться почему тест упал с ошибкой. Далее мы рассмотрим как ошибки можно обработать, чтобы сделать вывод более понятным.

Рекомендую ознакомиться с работой с исключениями в языке “Python в статье Исключения в python. Конструкция try - except для обработки исключений”[[61]](#footnote-60).

### 8.2.5 Проверка элемента на странице

Для вывода адекватного сообщения об ошибке мы будем перехватывать исключение с использованием оператора “assert”.

Для этого реализуем служебный метод поиска элемента и поместим его в BasePage. Метод будет возвращать нам “True”, если браузер смог обнаружить элемент и “False” в противном случае.

Рассмотрим один из вариантов реализации с использованием неявных ожиданий.

1. В конструктор BasePage добавим команду для неявного ожидания в 5 секунд:
2. **def** \_\_init\_\_(self, browser, url, timeout=5):
3. self.browser = browser
4. self.url = url
5. self.browser.implicitly\_wait(timeout)
7. Теперь напишем метод “is\_element\_present()”, который и будет заниматься проверкой наличия элемента на странице. Его реализация уже будет немного поинтереснее: мы должны передавать в него 2 аргумента: способ поиска (CSS/XPATH и т.д.) и значение поиска (сам селектор). Используем перехват исключения с помощью конструкции “try/except”.
8. **def** is\_element\_present(self, how, what):
9. """
10. @param how: способ поиска.
11. @param what: строка-селектор.
12. """
13. **try**:
14. self.browser.find\_element(how, what)
15. **except** имя\_исключения:
16. **return** False
17. **return** True

Если вам необходимо использовать не стандартное исключение языка Python, а исключение созданное в какой-либо библиотеке, то его нужно импортировать. Для реализации метода поиска вам потребуется импортировать исключение из библиотеки Selenium:

1. **from** selenium.common.exceptions **import** имя\_исключения

После успешного импорта исключения вы можете использовать новый метод для проверки элемента на наличие на любой странице, которая будет унаследована от базовой.

1. Теперь отрефакторим метод проверки ссылки на логин так, чтобы он выдавал адекватное сообщение об ошибке:
2. **def** should\_be\_login\_link(self):
3. **assert** self.is\_element\_present(By.CSS\_SELECTOR, '#login\_link\_invalid'), \
4. 'Login link is not presented on the page'

Запустите тесты и проверьте вывод сообщения об ошибке, использовав команду “pytest -v --tb=line --language=en test\_main\_page.py”.

Не забудьте исправить селектор, чтобы тест успешно проходил и сделайте коммит с осмысленным сообщением (рекомендуется использовать английский язык) о внесённых изменениях.

8.2.6 Элементы страниц в паттерне Page Object

Помните мы говорили, что наши тесты почти соответствуют подходу Page Object? Давайте разберёмся почему на небольшом примере.

Допустим вы работаете в команде, где постоянно вносятся изменения в проект. У вас уже написаны тесты взаимодействующие со страницей логина. К вам обращается руководство с просьбой протестировать страницу сайта промо-акции.

На первый взгляд ничего не изменилось кроме ссылки. Замените ссылку, с которой запускаются тесты главной на <http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook_209?promo=midsummer>.

Запустите тесты командой: “pytest -v --tb=line --language=en test\_main\_page.py”.

Тесты не прошли и их нужно фиксить. Подберите новый локатор к ссылке на логин.

Посчитайте сколько строк кода вам придётся менять, чтобы исправить ситуацию. А теперь представьте, что у вас написано несколько десятков или даже сотен тестов.

Для избежания такой ситуации, при проектировании тестов, хорошей практикой является вынесение локаторов для страниц в отдельный файл. Давайте и мы это сделаем.

1. В директории “pages” создайте новый файл “locators.py”.
2. Внутри него создайте класс. Класс будет создаваться для конкретной страницы и содержать все необходимые локаторы элементов для её тестирования, поэтому он должен иметь соответствующее название.
3. **from** selenium.webdriver.common.by **import** By

6. **class** MainPageLocators:
7. LOGIN\_LINK = (By.CSS\_SELECTOR, '#login\_link')

Обратите, что локатор теперь состоит из кортежа: как искать и что искать.

1. В файле “main\_page.py” импортируйте новый класс с локаторами:
2. **from** .locators **import** MainPageLocators
4. Теперь в классе MainPage заменим все старые записи использования локаторов на новый вариант использования через атрибуты класса MainPageLocators. Посмотрите на пример и выполните замену в остальных тестов:
5. **def** should\_be\_login\_link(self):
6. **assert** self.is\_element\_present(\*MainPageLocators.LOGIN\_LINK), 'Login link is not presented'

Обратите внимание на символ звёздочки, в языке Python он используется для распаковки, в данном случае кортежа, чтобы метод “is\_element\_present()” получил пару аргументов для вызова.

1. Запустите тесты повторно: “pytest -v --tb=line --language=en test\_main\_page.py”.

Тесты конечно же снова упадут с ошибкой, но если вам придётся снова менять локатор то менять его придется всего лишь в одном месте, а не искать все тесты в которых он использован.

Итак, PageObject — это не только методы, но и элементы.

Элементы на страницах часто меняются в процессе разработки. Поэтому важно поддерживать свой код в состоянии готовым быстро изменяться. В долгосрочной перспективе это уменьшит время на поддержку и обслуживание наших тестовых наборов.

Удалите ссылку на промо-акцию и замените её на обычную ссылку: “link = 'http://selenium1py.pythonanywhere.com/'”.

Верните старый локатор для LOGIN\_LINK, чтобы тесты проходили. Не забудьте добавить изменения вместе с осмысленным сообщением в ваш удалённый репозиторий.

### 8.2.6 Реализация LoginPage

На данный момент у нас реализована проверка ссылки к странице логина, но нет ни одной проверки для этой страницы.

Давайте проверим, что мы действительно перешли на нужную страницу. Для этого нам понадобится создать новый Page Object, назовем его LoginPage.

Возьмите за основу следующий код для файла “login\_page.py”, который необходимо будет поместить в директорию к остальным объектам страниц:

1. **from** .base\_page **import** BasePage

4. **class** LoginPage(BasePage):
5. **def** should\_be\_login\_page(self):
6. self.should\_be\_login\_url()
7. self.should\_be\_login\_form()
8. self.should\_be\_register\_form()
10. **def** should\_be\_login\_url(self):
11. *# реализуйте проверку на корректный url адрес*
12. **assert** True
14. **def** should\_be\_login\_form(self):
15. *# реализуйте проверку, что есть форма логина*
16. **assert** True
18. **def** should\_be\_register\_form(self):
19. *# реализуйте проверку, что есть форма регистрации на странице*
20. **assert** True

Вам предстоит самостоятельно реализовать эти проверки.

1. В файлу “locators.py” создайте новый класс для обнаружения элементов на странице логина LoginPageLocators.

2. Подберите селекторы, необходимые для проверок, и добавьте их в класс LoginPageLocators.

3. Реализуйте проверки с помощью локаторов. Не забудьте указывать осмысленные сообщения для ошибок и протестировать работу новых проверок.

4. Обратите внимание на метод “should\_be\_login\_url()” он должен проверить наличие подстроки в текущем url браузера. Для этого вам потребуется использовать свойство веб-драйвера: current\_url[[62]](#footnote-61).

5. Добавьте изменения в репозиторий.

### 8.2.7 Переходы между страницами

Переход можно осуществлять разными способами.

Мы рассмотрим неявный переход с явной инициализацией страницы. Такой способ позволяет делать тесты менее связными перекрёстными импортами.

1. Для этого в файле “main\_page.py” необходимо импортировать LoginPage:

1. **from** .login\_page **import** LoginPage

2. Инициализируем страницу в теле теста:

1. **def** test\_guest\_can\_go\_to\_login\_page(browser):
2. link = "http://selenium1py.pythonanywhere.com"
3. page = MainPage(browser, link)
4. page.open()
5. page.go\_to\_login\_page()
6. login\_page = LoginPage(browser, browser.current\_url)
7. login\_page.should\_be\_login\_page()

Можно также написать общий метод открытия странице по ссылке в классе BasePage и пользоваться им с любой страницы, которая будет наследована от BasePage. Так же есть и другие способы реализовать нужный функционал, но мы остановимся на текущей реализации.

### 8.2.7 Удобство поддержки тестов — инкапсуляция бизнес-логики в методах

Что делать, когда меняется логика взаимодействия со страницей, которая используется во множестве тестов? Например, давайте рассмотрим сценарий, в котором нам нужно проверить возможность перехода на страницу логина по ссылке в панели навигации на различных страницах веб-сайта. Предположим, что таких страниц 20, у нас будет 20 тестов, использующих метод “go\_to\_login\_page()” из класса “MainPage”. А теперь представьте, что разработчики ввели alert, который срабатывает при переходе по нужной нам ссылке и его нужно принять. Следовательно, все 20 тестов провалятся, потому что метод “go\_to\_login\_page()” не обрабатывает предупреждение, что также приводит к сбою метода “should\_be\_login\_page()” так как он не срабатывает.

Чтобы исправить эту ситуацию, мы можем отрефакторить метод “go\_to\_login\_page()”, включив в него обработку предупреждения. Таким образом, мы можем восстановить функциональность всех тестов, не внося никаких изменений в сами тесты.

1. **def** go\_to\_login\_page(self):
2. link = self.browser.find\_element(By.CSS\_SELECTOR, '#login\_link')
3. link.click()
4. alert = self.browser.switch\_to.alert
5. alert.accept()

Использование шаблона Page Object дает еще одно преимущество: разделение самого теста и логики взаимодействия со страницей. Это разделение повышает читаемость и поддерживаемость тестов, особенно когда код приложения подвергается изменениям. Отделяя тест от тонкостей взаимодействия со страницей, мы устанавливаем четкие границы, которые способствуют ясности кода и облегчают будущие изменения.

## 8.3 Улучшаем дизайн тестов

Вы уже знаете практически всю теорию о применении Page Object Model и далее будет много заданий на самостоятельное выполнение. Вы будете самостоятельно их делать и проверять себя. Это сильно увеличит эффективность усвоения теоретических знаний и у вас появится проект, который можно будет продемонстрировать.

### 8.3.1 Задание: добавление в корзину со страницы товара

Представим вам поручили автоматизировать следующий сценарий:

1. Открыть страницу “[http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-hand](http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook_209/?promo=newYear)

[book\_209/?promo=newYear](http://selenium1py.pythonanywhere.com/catalogue/the-shellcoders-handbook_209/?promo=newYear)”.

2. Нажать на кнопку "Добавить товар".

3. Решить уравнение и получить скидку, ввести ответ. Для этого использовав метод “solve\_quiz\_and\_get\_code()”, который вам отдали уже реализованный.

1. **from** selenium.common.exceptions **import** NoAlertPresentException

4. **def** solve\_quiz\_and\_get\_code(self):
5. alert = self.browser.switch\_to.alert
6. x = alert.text.split()[2]
7. answer = str(math.log(abs((12 \* math.sin(float(x))))))
8. alert.send\_keys(answer)
9. alert.accept()
10. **try**:
11. alert = self.browser.switch\_to.alert
12. alert\_text = alert.text
13. **print**(f'Code: {alert\_text}')
14. alert.accept()
15. **except** NoAlertPresentException:
16. **print**('No second alert present')

Ожидаемый результат:

1. Получение сообщения, что товар добавлен в корзину. Название товара в сообщении соответствует добавленному товару в корзину.

2. Сообщение со стоимостью корзины, стоимость которой соответствует стоимости добавленного товара.

Тест нужно написать, используя паттерн Page Object. Для этого вам нужно:

1. Добавить новый файл для тест-кейсов, связанных со страницей товара. Например “test\_product\_page.py”.

2. Создать класс для страницы товара. Опишите его в файле “product\_page.py” в директории “pages”.

3. Реализовать метод для добавления в корзину.

4. Реализовать методы-проверки.

5. Добавить локаторы к элементам страницы в отдельном классе файла “locators.py”.

6. Реализовать сам тест-кейс. Назовите тест “test\_guest\_can\_add\_product\_to\_basket”.

Хорошей практикой является создание новой ветки, отходящей от основной, в которой уже будет реализован новый функционал. Так же рекомендуется сперва создавать структуру, а затем уже переходить к реализации. Например, сперва создаются пустые файлы и прописываются импорты, затем описываются необходимые локаторы, после чего уже можно переходить к созданию объекта страницы и методы-проверки, и только в самом конце писать тесты. Но вы можете выполнять шаги с любого пункта.

Напомню, чтобы видеть результаты работы функции “print()” необходимо запускать Pytest с флагом “-s”.

### 8.3.2 Задание: независимость от данных

Для эффективных автотестов крайне важно минимизировать зависимости от конкретных данных. Жесткие проверки объектов, уникальных для конкретного экземпляра, настоятельно не рекомендуются. Почему? Что ж, данные имеют свойство часто меняться, что потребует постоянных фиксов автотестов. Более того, такой подход затрудняет повторное использование кода. Например, если мы собираемся запустить тест для нескольких продуктов, нам потребуется создать множество проверок, по одной для каждого продукта. В конечном итоге такая практика снижает качество продукта, поскольку эти тесты охватывают

лишь ограниченное количество страниц.

Общая рекомендация состоит в том, чтобы убедиться, что ваши тесты не зависят от факторов, находящихся вне вашего контроля. Сюда входят данные, хранящиеся в базах данных или сторонних службах, используемых вашим приложением. Желательно избегать проверки конкретных данных, если у вас нет специально подготовленной тестовой базы данных, которая инициализируется перед каждым запуском теста, или если вы можете добавить необходимые данные непосредственно в базу данных или через API приложения. Осуществляя контроль над тестовой средой и данными, вы можете поддерживать более

надежные и воспроизводимые тесты.

Запустите тест, который мы написали на предыдущем этапе разработки, со

ссылкой: [http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207](http://selenium1py.pythonanywhere.com/catalogue/coders-at-work_207/?promo=newYear2019)

[/?promo=newYear2019](http://selenium1py.pythonanywhere.com/catalogue/coders-at-work_207/?promo=newYear2019).

Если вы на прошлом шаге проверяли конкретное название товара, то ваш тест упадёт с ошибкой, так как теперь был добавлен другой товар. Если тест прошел успешно, то вы написали тест независимый от контекста и правильно сделали.

Чтобы тест был независимым от контента:

* Измените методы проверки так, чтобы они принимали название товара и цену товара.
* Сделайте метод, который берёт из элемента название товара и возвращает его.
* Сделайте такой же метод для цены.
* Проверьте, что название товара в сообщении совпадает с заголовком товара.

Вот несколько основных советов, как добиться независимости тестов:

1. Используйте тестовые данные: вместо того, чтобы полагаться на конкретные данные, существующие в определенном контексте, инициализируйте тестовые данные в самом тесте. Это гарантирует, что тест может выполняться с собственным набором данных независимо от контекста.

2. Избегайте "хардкодинга" значений: вместо "хардкодинга" конкретных значений или идентификаторов используйте переменные или константы в тестовом коде. Это обеспечивает гибкость при изменении значений без изменения логики тестирования.

3. Моделирование внешних зависимостей: если ваши тесты зависят от внешних систем или служб, рассмотрите возможность использования фиктивных (mock) фреймворков или методов для имитации этих зависимостей. Таким образом, ваши тесты могут выполняться независимо, не полагаясь на доступность или поведение внешних компонентов.

4. Изолируйте тестовые наборы: убедитесь, что каждый тестовый набор является автономным и не зависит от выполнения или результатов других тестовых наборов. Избегайте зависимостей между тестами.

5. Используйте методы setup и teardown: используйте методы настройки и очистки срезы, предоставляемые средами тестирования, для подготовки необходимой тестовой среды и очистки после теста. Это гарантирует, что каждый тест начинается с известного состояния и заканчивается приводя систему в исходное состояние.

6. Инкапсулируйте тестовую логику. Организуйте тестовый код в повторно используемые функции или классы, которые инкапсулируют тестовую логику. Это повышает читаемость, поддерживаемость и возможность повторного использования ваших тестов.

7. Избегайте зависимостей от внешних файлов конфигурации: если ваши тесты полагаются на файлы конфигурации, рассмотрите возможность предоставления специфичных для теста конфигураций в самом тестовом коде или с использованием методов имитации конфигурации.

### 8.3.3 Задание: независимость контента, ищем баг

Итак, вводная: была запущено несколько промо-акций, одна из которых приводит к появлению дефекта. Промо-акция включается путем добавления параметра “?promo=offerN” к ссылке на товар.

На не придется писать отдельные тесты для каждой ссылки на промо-акцию, чтобы проверить её работоспособность, мы будем использовать параметризацию. Вам нужно определить при каком значении параметра акции тест будет падать. Для этого мы должны обнаружить проблемный URL, на котором возникает ошибка. Значение “N” у параметра “?promo=offerN” изменяется от 0 до 9.

Параметризация используется для запуска теста с разными входными данными. Вот пример, как вы можете использовать декоратор параметризации:

1. @pytest.mark.parametrize('link', ["http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer0",
2. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer1",
3. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer2",
4. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer3",
5. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer4",
6. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer5",
7. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer6",
8. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer7",
9. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer8",
10. "http://selenium1py.pythonanywhere.com/catalogue/coders-at-work\_207/?promo=offer9"])
11. **def** test\_guest\_can\_add\_product\_to\_basket(browser, link):
12. *# ваша реализация теста*

Скажем, что после обнаружения бага, вам сказал, что фиксить баг в этом спринте вы не будете. Вам необходимо отметить тест, как ожидаемо падающий, использовав xfail. Мы уже говорили, как использовать эту фичу, вы можете освежить знания в памяти Skip/xfail with parametrize[[63]](#footnote-62).

Ваша задача:

1. Параметризовать тест.

2. Обнаружить дефект.

3. Пометить тест с дефектом, как ожидаемо падающий.

4. Зафиксировать изменения коммитом с осмысленным сообщением.

### 8.3.4 Отрицательные проверки: как проверить отсутствие элемента

Иногда при создании автотестов возникают ситуации, когда нам нужно проверить не только наличие элемента на странице, но и его отсутствие. Здесь стоит разделить две важные особенности:

1. Элемент может быть на странице (но не должен). Например, такое часто касается различных кнопок, которые появляются только при выполнении определённых условий. Примером для нашего интернет-магазина может быть проверка отображения сообщения об успешном добавлении товара в корзину при открытии корзины. Если мы будем осуществлять поиск элемента при негативной проверке без ожидания, то рискуем получить проходящий успешно тест и пропустить баг, если сообщение таки отображается, но с небольшой задержкой.

2. Элемент отображается на странице и должен исчезнуть через определённое время или в результате действий пользователя. Это может быть исчезновение прогрессбара или изображения товара при отводе курсора от его изображения.

#### 8.3.4.1 Почему нужно писать такие проверки с осторожностью?

Во-первых, тестам всегда придётся ждать обнаружения элемента, чтобы убедиться, что элемент действительно отсутствует или появляется. Если мы используем нашу написанную функцию “is\_element\_present()”, то тест с такой проверкой будет ждать полные и честные 5 секунд:

1. **def** should\_not\_be\_success\_message(self):
2. **assert** **not** self.is\_element\_present(\*ProductPageLocators.SUCCESS\_MESSAGE), \
3. 'Success message is presented'

Это довольно продолжительное время для позитивного сценария и стоит подобрать для этого теста индивидуальное время путём проб и ошибок. А для этих целей оправдано использование явных ожиданий.

Во-вторых, негативные тесты часто дают ложноположительные результаты, если селектор элемента устарел. Элемента с таким селектором на странице нет, а сообщение отображается. Тест при этом выдаст зелёный ответ, что все в порядке. Это плохо!

Поэтому на каждый негативный тест вы должны написать позитивный тест. В одном тесте проверяете, что элемента нет, в соседнем, что элемент есть.

Про распространённые ошибки составления UI-автотестов вы можете прочитать в

статье “UI-автотесты: как делать не стоит[[64]](#footnote-63)”.

#### 8.3.4.2 Как же тогда реализовывать такие проверки?

Почти всегда вам придётся ориентироваться в конкретной ситуации по разному, но общая рекомендация - использовать явные ожидания. Хорошей практикой будет придерживаться исключительно явных ожиданий в рамках проекта и не использовать неявные.

Добавим в BasePage абстрактный метод, который проверяет, что элемент не появляется на странице в течение заданного времени:

1. **def** is\_not\_element\_present(self, how, what, timeout=4):
2. **try**:
3. WebDriverWait(self.browser, timeout).until(EC.presence\_of\_element\_located((how, what)))
4. **except** TimeoutException:
5. **return** True
6. **return** False

Использование данного метода для страницы товара будет следующим:

1. **def** should\_not\_be\_success\_message(self):
2. **assert** self.is\_not\_element\_present(\*ProductPageLocators.SUCCESS\_MESSAGE), \
3. 'Success message is presented, but should not be'

Если же мы хотим проверить, что какой-то элемент исчезает, то следует воспользоваться явным ожиданием вместе с функцией “until\_not()”, в зависимости от ожидаемого результата:

1. **def** is\_disappeared(self, how, what, timeout=1):
2. **try**:
3. WebDriverWait(self.browser, timeout).until\_not(EC.presence\_of\_element\_located((how, what)))
4. **except** TimeoutException:
5. **return** False
6. **return** True

Напишите самостоятельно метод-проверку на исчезновение для класса страницы товара

аналогично “should\_not\_be\_success\_message()”.

Обратите внимание на разницу между методами “is\_not\_element\_present()” и “is\_disappeared()”.

Поскольку к проектированию таких тестов необходимо относиться очень аккуратно и писать дополнительные проверки, то рекомендуется их избегать без явной необходимости.

### 8.3.5 Задание: отрицательные проверки

Добавьте к себе в проект функции из предыдущего шага и реализуйте несколько простых тестов:

1. test\_guest\_cant\_see\_success\_message\_after\_adding\_product\_to\_basket:

* Открыть страницу товара.
* Добавить товар в корзину.
* Проверить, что нет сообщения об успехе с помощью “is\_not\_element\_present”.

2. test\_guest\_cant\_see\_success\_message:

* Открыть страницу товара.
* Проверить, что нет сообщения об успехе с помощью “is\_not\_element\_present”.

3. test\_message\_disappeared\_after\_adding\_product\_to\_basket:

* Открыть страницу товара.
* Добавить товар в корзину.
* Проверить, что нет сообщения об успехе с помощью “is\_disappeared”.

Пометьте тесты, которые упали как XFail.

Не забудьте зафиксировать изменения с осмысленным сообщением коммита и запушить в свой репозиторий.

### 8.3.6 Плюсы наследования: магия ООП

Рассмотрим один аспект парадигмы ООП, который мы до сих пор немного использовали: механизм наследования.

Мы уже его использовали, когда выделили базовый класс для наших проверок BasePage, а от него уже наследовали остальные объекты страниц: LoginPage, MainPage. Сейчас в нашем классе BasePage базовые технические детали - поиск элементов, метод для открытия страницы и вспомогательные методы. Но мы можем в него добавить и методы общих для всех страниц.

Важным моментом будет отделение общего от частного. Таким образом, в элемент базовой страницы должно попасть исключительно общее для всех страниц.

Используя такой подход мы уменьшаем количество дублируемого кода и упрощаем поддержку кода.

### 8.3.7 Плюсы наследования: пример

Вы наверное уже давно заметили, что ссылка на страницу логина присутствует на каждой странице. Если мы хотим добавить тест “test\_guest\_can\_go\_to\_login\_page”, то для избежания дублирования следует перенести соответствующие методы в класс BasePage.

В файле “locators.py” создаем новый класс BasePageLocators:

1. **class** BasePageLocators():
2. LOGIN\_LINK = (By.CSS\_SELECTOR, '#login\_link')
3. LOGIN\_LINK\_INVALID = (By.CSS\_SELECTOR, '#login\_link\_invalid')

Переносим методы в “base\_page.py”:

1. **from** .locators **import** BasePageLocators

4. **class** BasePage():
5. **def** go\_to\_login\_page(self):
6. link = self.browser.find\_element(\*BasePageLocators.LOGIN\_LINK\_INVALID)
7. link.click()
9. **def** should\_be\_login\_link(self):
10. **assert** self.is\_element\_present(\*BasePageLocators.LOGIN\_LINK), 'Login link is not presented'

Обратите внимание, что новые методы должны быть добавлены к уже имеющимся, иначе ваши тесты разучатся находить элементы и т.п.

В классе MainPage у нас не осталось никаких методов, поэтому добавим туда заглушку:

1. **class** MainPage(BasePage):
2. **def** \_\_init\_\_(self, \*args, \*\*kwargs):
3. super(MainPage, self).\_\_init\_\_(\*args, \*\*kwargs)

Добавляем в файл c тестами “test\_product\_page.py” новые тесты:

1. **def** test\_guest\_should\_see\_login\_link\_on\_product\_page(browser):
2. link = 'http://selenium1py.pythonanywhere.com/en-gb/catalogue/the-city-and-the-stars\_95/'
3. page = ProductPage(browser, link)
4. page.open()
5. page.should\_be\_login\_link()

Добавьте самостоятельно второй тест “test\_guest\_can\_go\_to\_login\_page\_from\_product\_page”.

Запустите тесты и убедитесь, что они выполняются положительно. Зафиксируйте изменения коммитом с осмысленным сообщением.

### 8.3.8 Задание: наследование и отрицательные проверки

В файл “test\_main\_page.py” добавьте тест с названием “test\_guest\_cant\_see\_product\_in\_basket\_opened\_from\_main\_page”:

1. гость открывает главную страницу;

2. переходит в корзину по кнопке в шапке сайта;

3. ожидаем, что в корзине нет товаров;

4. ожидаем, что есть текст о том что корзина пуста.

В файл “test\_product\_page.py” добавьте тест с названием “test\_guest\_cant\_see\_product\_in\_basket\_opened\_from\_product\_page”:

1. гость открывает страницу товара;

2. переходит в корзину по кнопке в шапке;

3. ожидаем, что в корзине нет товаров;

4. ожидаем, что есть текст о том что корзина пуста.

В классе BasePage реализуйте соответствующий метод для перехода в корзину.

Создайте файл “basket\_page.py” и в нем класс BasketPage. Реализуйте там необходимые проверки, в том числе негативную. Убедитесь, что тесты проходят и зафиксируйте изменения в коммите.

### 8.3.9 Группировка тестов: setup

Воспользуемся ООП для организации тестов. Pytest позволяет объединять тесты в тестовые наборы. Зачем это делать и почему это удобно?

Во-первых, вы можете объединить тесты по смыслу, разделив функционал по фичам, компонентам или по какому-либо другому признаку. Давайте тесты для MainPage необходимые для проверки логина выделим в отдельный тестовый набор в файле “test\_main\_page.py” и пометим его меткой “login\_guest”:

1. @pytest.mark.login\_guest
2. **class** TestLoginFromMainPage:
3. *# не забываем передать первым аргументом self*
4. **def** test\_guest\_can\_go\_to\_login\_page(self, browser):
5. *# реализация теста*
7. **def** test\_guest\_should\_see\_login\_link(self, browser):
8. *# реализация теста*

Запустите тестовый набор используя маркировку, необходимо добавить флаг “-m” и название метки. Обратите внимание, что использована всего лишь одна метка, а запущено два теста.

Во-вторых, можно выделять общие функции для всего тестового набора, чтобы не заниматься дублированием кода. Эти функции уже нам знакомы: setup - функция, которая выполняется до запуска теста и teardown - функция, которая запускается после каждого теста, как правило в неё помещают методы очистки окружения или его восстановление после теста. Такие функции реализуют с помощью фикстур. Важно обратить внимание на область использование фикстуры “scope="function"”, что будет означать, что фикстура будет исполняться для каждой функции, и на параметр “autouse=True”, который говорит, что фикстура будет использоваться без явного вызова.

Для обеспечения независимости от контента там, где мы взаимодействуем с товаром, создавать новый товар по API и взаимодействовать с ним. Удаляя товар после выполнения тестового сценария. Наш интернет-магазин не поддерживает такой опции, но важно понимать важность использования п группировки тестовы и выделения подготовительных и завершающих шагов.

Давайте посмотрим на пример реализации тестов в подобном случае:

1. @pytest.mark.login
2. **class** TestLoginFromProductPage:
3. @pytest.fixture(scope='function', autouse=True)
4. **def** setup(self):
5. self.product = ProductFactory(title='Created book')
6. *# создание по API*
7. self.link = self.product.link
8. **yield**
9. *# после этого ключевого слова начинается teardown*
10. *# выполнится после каждого теста в классе*
11. *# удаляем те данные, которые мы создали*
12. self.product.delete()
14. **def** test\_guest\_can\_go\_to\_login\_page\_from\_product\_page(self, browser):
15. page = ProductPage(browser, self.link)
16. *# реализация теста*
18. **def** test\_guest\_should\_see\_login\_link(self, browser):
19. page = ProductPage(browser, self.link)
20. *# реализация теста*

### 8.3.10 Задание: группировка тестов и setup

Давайте попрактикуемся в написании сетапов для объединённых в группу тестов. Важно отметить, что этот пример исключительно для учебных целей и практики для закрепления теоретических аспектов. В реальной разработке используются API методы или база данных для этих целей.

Настала пора добавить тестовые сценарии не только для гостей нашего интернет-магазина, но и для зарегистрированных пользователей.

1. В файле “test\_product\_page.py” создайте новый класс “TestUserAddToBasketFromProductPage”.

2. Скопируйте в него уже реализованные тесты “test\_guest\_cant\_see\_success\_message” и “test\_guest\_can\_add\_product\_to\_basket” и переименуйте в них “guest” на “user”. Содержимое этих тестов можно не менять. Всё, что вам нужно будет сделать, - это добавить шаг регистрации.

3. Реализуйте в LoginPage метод “register\_new\_user(email: str, password: str)”, который принимает две строки и регистрирует пользователя.

4. Добавьте в BasePage проверку того, что пользователь залогинен:

1. **def** should\_be\_authorized\_user(self):
2. **assert** self.is\_element\_present(\*BasePageLocators.USER\_ICON), \
3. 'User icon is not presented, probably unauthorised user'

5. Добавьте локатор соответственно в BasePageLocators:

1. USER\_ICON = (By.CSS\_SELECTOR, '.icon-user')

6. Добавьте в класс фикстуру setup. В этой функции нужно:

* открыть страницу регистрации;
* зарегистрировать нового пользователя;
* проверить, что пользователь залогинен.

Убедитесь, что тесты проходят и действительно регистрируют новых пользователей.

Зафиксируйте изменения с осмысленным сообщением в репозитории.

## 8.4 Создание отчётов о тестировании с помощью фреймворка Allure

### 8.4.1 Что такое отчёт о тестировании и зачем он нужен?

Отчётность - одна из важных составляющих проекта, которая всегда проблематична для составления. Если у вас нет примеров для составления подобной документации на проекте, то вы обязательно зададитесь вопросами о том, что в неё написать, как описать выполненные действия, зачем это вообще описывать и кто будет читать этот отчёт.

На самом деле, отчёт важен не только заказчику, чтобы посмотреть в каком состоянии находится выполнение текущего этапа проекта, но и самой команде, чтобы разобраться в проблемных местах системы.

Отчет о результатах тестирования - периодический отчет, в котором документируется подробная информация о выполнении теста и его результате. Также он содержит условия, предположения, ограничения теста, какой элемент теста кем тестируется. Помимо этого вносится подробная информация об оставшейся работе, чтобы показать, сколько еще работы

необходимо выполнить в проекте.[[65]](#footnote-64)

Мы будем использовать фреймворк для создания отчётов Allure Framework[[66]](#footnote-65), но давайте

сначала зададимся вопросом "А действительно ли необходимо использовать какой-то инструмент для формирования отчёта?".

### 8.4.2 Стоит ли использовать инструменты формирования отчётов?

Использование инструмента создания отчетов, такого как Allure, не является обязательным, но оно может дать несколько преимуществ, когда речь идет о создании отчётов по тестированию и их анализе. Разберём несколько причин, по которым использование инструмента создания отчетов является более целесообразным.

1. Усовершенствованные отчёты о тестировании. Инструменты создания отчётов предоставляют более структурированные и визуально привлекательные отчёты по сравнению с выходными данными программы запуска тестов по умолчанию. Эти отчёты могут включать дополнительную информацию, такую как этапы тестирования, вложения (снимки экрана, журналы и т. д.) и общее состояние тестирования. Это облегчает понимание результатов тестирования и выявление проблем.

2. Исторические данные и тренды. Инструменты генерации отчётов часто имеют возможность сохранять исторические данные испытаний и генерировать отчёты о трендах. Это позволяет вам отслеживать ход ваших тестов с течением времени, выявлять закономерности и принимать решения для улучшения тестов на основе данных.

3. Совместная работа и документация. Отчёты могут служить артефактом документации для ваших тестов, облегчая членам команды и заинтересованным сторонам просмотр и понимание результатов тестирования. Он способствует совместной работе и предоставляет централизованное место для общения и обсуждения результатов тестирования.

4. Интеграция с CI/CD. Многие инструменты генерации отчётов хорошо интегрируются с конвейерами непрерывной интеграции/непрерывного развертывания (CI/CD). Они могут автоматически генерировать отчёты после каждого запуска тестов и предоставлять сводку результатов. Это обеспечивает плавную интеграцию в существующий рабочий процесс разработки.

Хотя использование инструмента создания отчётов не является обязательным, он может значительно улучшить аспекты визуализации, анализа и совместной работы при выполнении теста. В конечном итоге это зависит от требований вашего проекта и уровня отчётности и анализа, необходимых для ваших тестов.

### 8.4.3 Использование Allure

#### 8.4.3.1 Установка зависимостей

Вам понадобится сам фреймворк allure-python[[67]](#footnote-66). Мы будем использовать генерацию отчёта в HTML, для этого нам понадобится библиотека allure-pytest[[68]](#footnote-67). После установки зависимостей в вашу виртуальную среду не забудьте добавить их в “requirements.txt”.

Посмотрим на применение маркировки тестов для страницы Accordian[[69]](#footnote-68).

1. @allure.suite('Alerts Frame Windows tab')
2. **class** TestAlertsFrameWindows:
3. """Class represents Alerts, Frame & Windows tab.
4. Contains tabs:
5. Browser Windows,
6. Alerts,
7. Frames,
8. Nested Frames,
9. Modal Dialogs.
10. """
11. EXPECTED\_TEXT = 'This is a sample page'
13. @allure.feature('Browser Windows')
14. **class** TestBrowserWindows:
15. """Class represents Browser Windows tab tests."""
16. browser\_windows\_page\_link = 'https://demoqa.com/browser-windows'
18. @allure.title('Test opening a new tab and getting the text from it.')
19. **def** test\_new\_tab(self, driver: WebDriver):
20. """Test opening a new tab and getting the text from it."""
21. browser\_windows\_page = BrowserWindowsPage(driver=driver, url=self.browser\_windows\_page\_link)
22. browser\_windows\_page.open()
23. new\_tab\_text = browser\_windows\_page.check\_opened(what\_to\_open='tab')
24. **assert** new\_tab\_text == TestAlertsFrameWindows.EXPECTED\_TEXT, \
25. f'New tab text should be **\'**{TestAlertsFrameWindows.EXPECTED\_TEXT}**\'**' \
26. f' but got {new\_tab\_text}'

Обратите внимание, что мы объявили класс TestAlertsFrameWindows тестовым набором с помощью декоратора “@allure.suite('Alerts Frame Windows tab')”, в который входит большой перечень фичей: Browser Windows, Alerts, Frames и т.д. Каждую фичу мы должны пометить декоратором “@allure.feature()” с названием самой фичи. Фича представляет собой класс хранящий в себе тесты посвященные определённой тематике. Каждый тест отметим декоратором “@allure.title()” внутри которого стоит указать название нашего теста. Хорошей практикой является придерживаться форматов именования тестов, которые приняты на вашем проекте и используются в ручном тестировании.

Давайте посмотрим на пример декорирования шагов тестового сценария:

1. **class** BrowserWindowsPage(BasePage):
2. """Browser Windows page object."""
3. locators = BrowserWindowsPageLocators()
5. @allure.step('Check if new tab of window is opened.')
6. **def** check\_opened(self, what\_to\_open: str) -> str:
7. """
8. Check if new tab of window is opened.
9. :param what\_to\_open: Element to open.
10. tab -> open the tab.
11. window -> open the window.
12. :returns: title of new tab or window.
13. """
14. available\_cases = {
15. 'tab': self.locators.NEW\_TAB\_BUTTON,
16. 'window': self.locators.NEW\_WINDOW\_BUTTON
17. }
18. **with** allure.step(f'Click on {what\_to\_open}'):
19. self.element\_is\_clickable(
20. locator=available\_cases[what\_to\_open]
21. ).click()
22. **with** allure.step('Switch to new tab'):
23. self.switch\_to\_new\_tab()
24. **with** allure.step('Get title text'):
25. text\_title = self.element\_is\_present(
26. locator=self.locators.SAMPLE\_TEXT
27. ).text
28. **return** text\_title

Каждое действие метода-проверки помечается соответствующим декоратором. Не забывайте писать осмысленные сообщения о действии в шаге и никогда не пропускайте шаги вашего тест-кейса.

#### 8.4.3.2 Получение скриншотов экрана

Вы можете получать скриншоты экрана с помощью средств автоматизации и прикреплять их к вашему отчёту. Это будет удобно для визуального наблюдения дефекта, когда вы запускаете автотесты в “headless” режиме или ставите ночной прогон тестов.

1. @pytest.fixture(scope='function')
2. **def** driver():
3. """Google Chrome is used as driver."""
4. driver = webdriver.Chrome(
5. service=ChromeService(ChromeDriverManager().install())
6. )
7. driver.maximize\_window()
8. **yield** driver
9. screenshot = driver.get\_screenshot\_as\_png()
10. allure.attach(
11. body=screenshot,
12. name=f'Screenshot {datetime.today()}',
13. attachment\_type=allure.attachment\_type.PNG
14. )
15. driver.quit()

#### 8.4.3.3 Готовый отчёт о тестировании

Используйте приведённые примеры для оформления ваших собственных методов, тестовых наборов и тестов.

Чтобы получить отчёт о тестировании вам необходимо использовать команду с указанием директории для сохранения файлов отчёта: “pytest --alluredir=allure\_results .\tests”, где “tests” - директория, содержащая все тесты, “allure\_results” - директория для сохранения файлов отчёта.

Чтобы собрать свой отчёт в HTML используйте команду “allure serve allure\_results”.

Откроется окно браузера, в котором у вас будет готовый отчёт по тестированию.

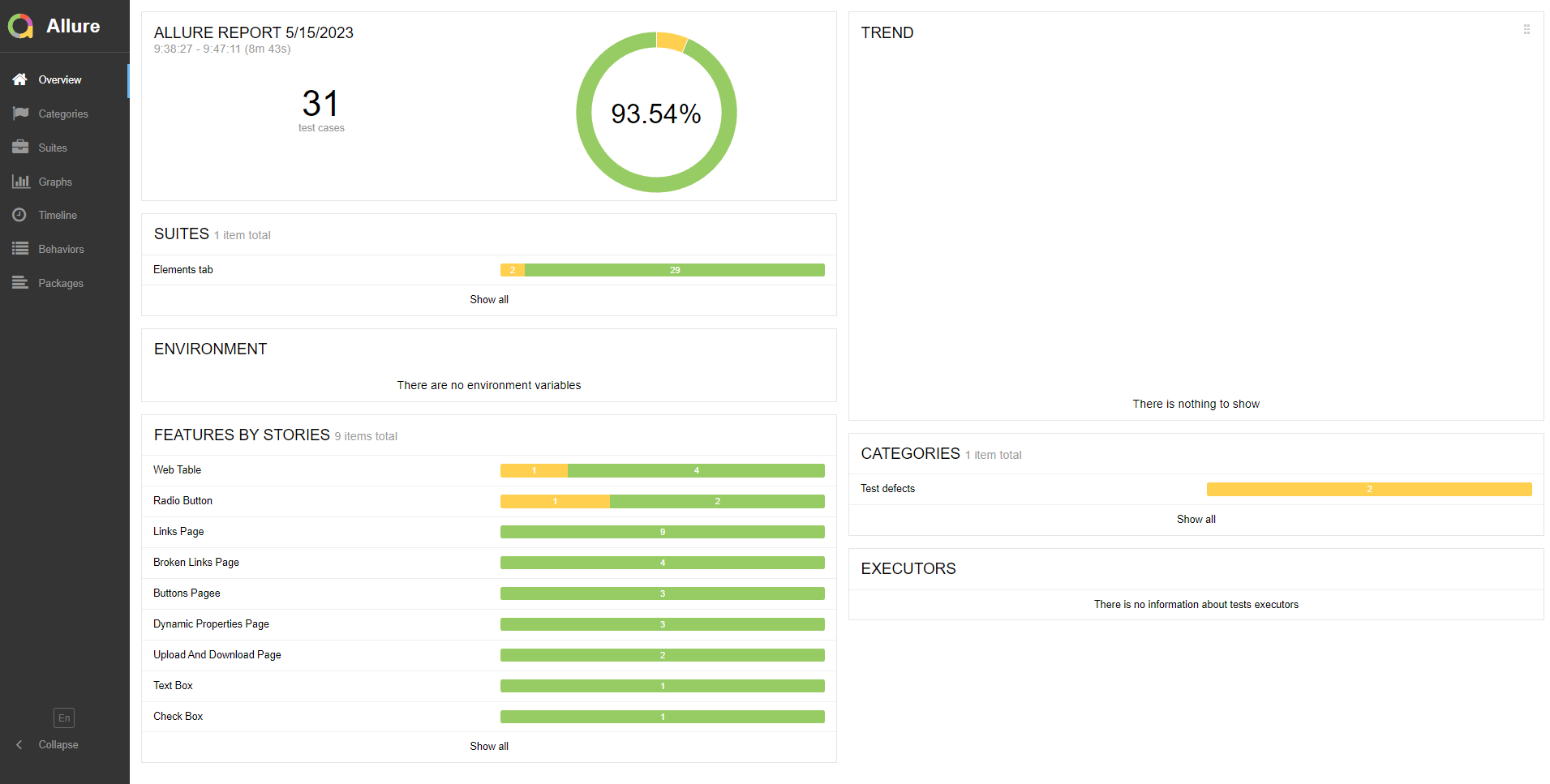


Рисунок 17. Allure отчёт.

# Библиографический список литературы

1. Виды отчетов (Reports) // QA bible : сайт. – URL: <https://vladislaveremeev.gitbook.io/qa_bible/testovaya-dokumentaciya-i-artefakty-test-deliverablestest-artifacts/vidy-otchetov-reports> (дата обращения: 30.05.2023)
2. Инструменты разработчика // MDN Web Docs : сайт. – URL: <https://developer.mozilla.org/ru/docs/Glossary/Developer_Tools> (дата обращения: 18.05.2023)
3. Исключения в python. Конструкция try - except для обработки исключений // Python 3 для начинающих : сайт. – URL: <https://pythonworld.ru/tipy-dannyx-v-python/isklyucheniya-v-python-konstrukciya-try-except-dlya-obrabotki-isklyuchenij.html> (дата обращения: 26.05.2023)
4. Куликов, С. С. Тестирование программного обеспечения. Базовый курс. (3-е издание) / С. С. Куликов. – Newtown, Pennsylvania : EPAM Systems, 2023. – 301 с. – URL: <https://svyatoslav.biz/software_testing_book/> (дата обращения: 20.03.2023). – Режим доступа: свободный
5. Основы CSS // MDN Web Docs : сайт. – URL: <https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/CSS_basics> (дата обращения: 21.03.2023
6. Основы HTML // MDN Web Docs : сайт. – URL: <https://developer.mozilla.org/ru/> (дата обращения: 20.03.2023)
7. Практика хорошего кода // Habr : сайт. – URL: <https://habr.com/ru/articles/206868/> (дата обращения: 11.05.2023)
8. Принципы тестирования: нас 7 // Habr : сайт. – URL: <https://habr.com/ru/articles/699990/> (дата обращения: 20.03.2023)
9. Скачать chromedriver для автоматизации Google Chrome в Python : сайт. – URL: <https://selenium-python.com/install-chromedriver-chrome> (дата обращения: 05.04.2023)
10. Тестирование программного обеспечения // Wikipedia : сайт. – URL: [https://ru.wikipedia.org/wiki/Тестирование\_программного\_обеспечения](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) (дата обращения: 20.03.2023)
11. Установка драйвера geckodriver для Firefox Selenium // selenium-python : сайт. – URL: <https://selenium-python.com/install-geckodriver> (дата обращения: 04.05.2023)
12. Фундаментальная теория тестирования // Habr : сайт. – URL: <https://habr.com/ru/articles/549054/> (дата обращения: 20.03.2023)
13. Что такое красивый код, и как его писать? // Habr : сайт. – URL: <https://habr.com/ru/articles/266969/> (дата обращения: 11.05.2023)
14. Шпаргалка по принципам ООП // Tproger : сайт. – URL: <https://tproger.ru/translations/oop-principles-cheatsheet/> (дата обращения: 11.05.2023)
15. ...почему зависимости между тестами это плохо? // barancev.github.io : сайт. – URL: <http://barancev.github.io/test-deps-are-evil/> (дата обращения: 11.05.2023)
16. UI-автотесты: как делать не стоит // habr.com : сайт. – URL: <https://habr.com/ru/companies/badoo/articles/419419/> (дата обращения: 29.05.2023)
17. Adding finalizers directly // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/how-to/fixtures.html#adding-finalizers-directly> (дата обращения: 28.04.2023)
18. ChromeDriver - WebDriver for Chrome : сайт. – URL: <https://chromedriver.chromium.org/> (дата обращения: 05.04.2023)
19. Conventions for Python test discovery // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/stable/explanation/goodpractices.html#conventions-for-python-test-discovery> (дата обращения: 26.04.2023)
20. DEMOQA : сайт. – URL: <https://demoqa.com/> (дата обращения: 05.04.2023)
21. Don't repeat yourself // Wikipedia : сайт. – URL: <https://en.wikipedia.org/wiki/Don't_repeat_yourself> (дата обращения: 11.05.2023)
22. Download PyCharm: Python IDE for Professional Developers by JetBrains : сайт. – URL: <https://www.jetbrains.com/pycharm/download/> (дата обращения: 05.04.2023)
23. Download Python | Python.org : сайт. – URL: <https://www.python.org/downloads/> (дата обращения: 05.04.2023)
24. GitHub : сайт. – URL: <https://github.com/mozilla/geckodriver/releases> (дата обращения: 04.05.2023)
25. GitHub : сайт. – URL: <https://github.com/allure-framework> (дата обращения: 30.05.2023)
26. How to change command line options defaults // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/example/simple.html?highlight=addoption> (дата обращения: 04.05.2023)
27. How to implement xunit-style set-up // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/how-to/xunit_setup.html> (дата обращения: 28.04.2023)
28. How to parametrize fixtures and test functions // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/how-to/parametrize.html> (дата обращения: 04.05.2023)
29. How to use skip and xfail to deal with tests that cannot succeed // pytest.org : сайт. – URL: <https://pytest.org/en/stable/how-to/skipping.html#skip-xfail-with-parametrize> (дата обращения: 29.05.2023)
30. HTML Attribute Reference // W3Schools Online Web Tutorials : сайт. – URL: <https://www.w3schools.com/tags/ref_attributes.asp> (дата обращения: 11.04.2023)
31. ISTQB Glossary : сайт. – URL: <https://glossary.istqb.org/en_US/term/testing-4-2> (дата обращения: 20.03.2023)
32. KISS principle // Wikipedia : сайт. – URL: <https://en.wikipedia.org/wiki/KISS_principle> (дата обращения: 11.05.2023)
33. LearnGitBranching : сайт. – URL: <https://learngitbranching.js.org/?locale=ru_RU> (дата обращения: 28.03.2023)
34. Markdown-Cheatsheet // GitHub : сайт. – URL: <https://github.com/Lexxx42/Markdown-Cheatsheet> (дата обращения: 29.03.2023)
35. Markdown Cheatsheet // GitHub : сайт. – URL: <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet> (дата обращения: 29.03.2023)
36. Mozilla : сайт. – URL: <https://www.mozilla.org/en-US/firefox/new/> (дата обращения: 04.05.2023)
37. Oscar - Sandbox : сайт. – URL: <http://selenium1py.pythonanywhere.com/> (дата обращения: 04.05.2023)
38. Override a fixture on a folder (conftest) level // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/7.1.x/how-to/fixtures.html?highlight=fixture%20folder#override-a-fixture-on-a-folder-conftest-level> (дата обращения: 04.05.2023)
39. PEP 8 – Style Guide for Python Code // peps.python.org : сайт. – URL: <https://peps.python.org/pep-0008/> (дата обращения: 11.05.2023)
40. PEP 20 – The Zen of Python // peps.python.org : сайт. – URL: <https://peps.python.org/pep-0020/> (дата обращения: 28.04.2023)
41. PEP 257 – Docstring Conventions // peps.python.org : сайт. – URL: <https://peps.python.org/pep-0257/> (дата обращения: 11.05.2023)
42. PyPI : сайт. – URL: <https://pypi.org/project/allure-pytest/> (дата обращения: 30.05.2023)

Plugin List // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/reference/plugin_list.html> (дата обращения: 04.05.2023)

1. Selenium Test Pages : сайт. – URL: <https://testpages.herokuapp.com/styled/index.html> (дата обращения: 18.05.2023)
2. selenium.webdriver.support.expected\_conditions // Selenium : сайт. – URL: <https://www.selenium.dev/selenium/docs/api/py/webdriver_support/selenium.webdriver.support.expected_conditions.html> (дата обращения: 21.04.2023)
3. Statcounter Global Stats - Browser, OS, Search Engine including Mobile Usage Share : сайт. – URL: <https://gs.statcounter.com/> (дата обращения: 05.04.2023)
4. Test fixture // Wikipedia : сайт. – URL: <https://en.wikipedia.org/wiki/Test_fixture#Software> (дата обращения: 28.04.2023)
5. WebDriver API // selenium-python.readthedocs.io : сайт. – URL: <https://selenium-python.readthedocs.io/api.html#selenium.webdriver.remote.webdriver.WebDriver.current_url> (дата обращения: 27.05.2023)
6. XPath // MDN Web Docs : сайт. – URL: <https://developer.mozilla.org/en-US/docs/Web/XPath> (дата обращения: 06.04.2023)

1. Тестирование программного обеспечения // Wikipedia : сайт. – URL: [https://ru.wikipedia.org/wiki/Тестирование\_программного\_обеспечения](https://ru.wikipedia.org/wiki/%D0%A2%D0%B5%D1%81%D1%82%D0%B8%D1%80%D0%BE%D0%B2%D0%B0%D0%BD%D0%B8%D0%B5_%D0%BF%D1%80%D0%BE%D0%B3%D1%80%D0%B0%D0%BC%D0%BC%D0%BD%D0%BE%D0%B3%D0%BE_%D0%BE%D0%B1%D0%B5%D1%81%D0%BF%D0%B5%D1%87%D0%B5%D0%BD%D0%B8%D1%8F) (дата обращения: 20.03.2023) [↑](#footnote-ref-0)
2. Куликов, С. С. Тестирование программного обеспечения. Базовый курс. (3-е издание) / С. С. Куликов. – Newtown, Pennsylvania : EPAM Systems, 2023. – 301 с. – URL: https://svyatoslav.biz/software\_testing\_book/ (дата обращения: 20.03.2023). – Режим доступа: свободный [↑](#footnote-ref-1)
3. ISTQB Glossary : сайт. – URL: <https://glossary.istqb.org/en_US/term/testing-4-2> (дата обращения: 20.03.2023) [↑](#footnote-ref-2)
4. Фундаментальная теория тестирования // Habr : сайт. – URL: <https://habr.com/ru/articles/549054/> (дата обращения: 20.03.2023) [↑](#footnote-ref-3)
5. Принципы тестирования: нас 7 // Habr : сайт. – URL: <https://habr.com/ru/articles/699990/> (дата обращения: 20.03.2023) [↑](#footnote-ref-4)
6. Основы HTML // MDN Web Docs : сайт. – URL: <https://developer.mozilla.org/ru/> (дата обращения: 20.03.2023) [↑](#footnote-ref-5)
7. Основы CSS // MDN Web Docs : сайт. – URL: https://developer.mozilla.org/ru/docs/Learn/Getting\_started\_with\_the\_web/CSS\_basics (дата обращения: 21.03.2023) [↑](#footnote-ref-6)
8. LearnGitBranching : сайт. – URL: <https://learngitbranching.js.org/?locale=ru_RU> (дата обращения: 28.03.2023) [↑](#footnote-ref-7)
9. Markdown-Cheatsheet // GitHub : сайт. – URL: <https://github.com/Lexxx42/Markdown-Cheatsheet> (дата обращения: 29.03.2023) [↑](#footnote-ref-8)
10. Markdown Cheatsheet // GitHub : сайт. – URL: <https://github.com/adam-p/markdown-here/wiki/Markdown-Cheatsheet> (дата обращения: 29.03.2023) [↑](#footnote-ref-9)
11. Download PyCharm: Python IDE for Professional Developers by JetBrains : сайт. – URL: <https://www.jetbrains.com/pycharm/download/> (дата обращения: 05.04.2023) [↑](#footnote-ref-10)
12. Download Python | Python.org : сайт. – URL: <https://www.python.org/downloads/> (дата обращения: 05.04.2023) [↑](#footnote-ref-11)
13. Statcounter Global Stats - Browser, OS, Search Engine including Mobile Usage Share : сайт. – URL: <https://gs.statcounter.com/> (дата обращения: 05.04.2023) [↑](#footnote-ref-12)
14. ChromeDriver - WebDriver for Chrome : сайт. – URL: <https://chromedriver.chromium.org/> (дата обращения: 05.04.2023) [↑](#footnote-ref-13)
15. Скачать chromedriver для автоматизации Google Chrome в Python : сайт. – URL: <https://selenium-python.com/install-chromedriver-chrome> (дата обращения: 05.04.2023) [↑](#footnote-ref-14)
16. DEMOQA : сайт. – URL: <https://demoqa.com/> (дата обращения: 05.04.2023) [↑](#footnote-ref-15)
17. Selenium Test Pages : сайт. – URL: <https://testpages.herokuapp.com/styled/index.html> (дата обращения: 18.05.2023) [↑](#footnote-ref-16)
18. Basic Web Page Title : сайт. – URL: <https://testpages.herokuapp.com/styled/basic-web-page-test.html> (дата обращения: 18.05.2023) [↑](#footnote-ref-17)
19. Инструменты разработчика // MDN Web Docs : сайт. – URL: <https://developer.mozilla.org/ru/docs/Glossary/Developer_Tools> (дата обращения: 18.05.2023) [↑](#footnote-ref-18)
20. Basic Web Page Title : сайт. – URL: <https://testpages.herokuapp.com/styled/basic-web-page-test.html> (дата обращения: 18.05.2023) [↑](#footnote-ref-19)
21. Test Page For Element Attributes : сайт. – URL: <https://testpages.herokuapp.com/styled/attributes-test.html> (дата обращения: 18.05.2023) [↑](#footnote-ref-20)
22. Windows Example Test : сайт. – URL: https://testpages.herokuapp.com/styled/windows-test.html (дата обращения: 18.05.2023) [↑](#footnote-ref-21)
23. Welcome to the Find By Playground : сайт. – URL: <https://testpages.herokuapp.com/styled/find-by-playground-test.html> (дата обращения: 18.05.2023) [↑](#footnote-ref-22)
24. HTML5 Form Elements Test Page : сайт. – URL: <https://testpages.herokuapp.com/styled/html5-form-test.html> (дата обращения: 18.05.2023) [↑](#footnote-ref-23)
25. XPath // MDN Web Docs : сайт. – URL: <https://developer.mozilla.org/en-US/docs/Web/XPath> (дата обращения: 06.04.2023) [↑](#footnote-ref-24)
26. DEMOQA Practice Form : сайт. – URL: <https://demoqa.com/automation-practice-form> (дата обращения: 18.05.2023) [↑](#footnote-ref-25)
27. DEMOQA Text Box : сайт. – URL: <https://demoqa.com/text-box> (дата обращения: 18.05.2023) [↑](#footnote-ref-26)
28. DEMOQA Links : сайт. – URL: <https://demoqa.com/links> (дата обращения: 18.05.2023) [↑](#footnote-ref-27)
29. DEMOQA Practice Form : сайт. – URL: <https://demoqa.com/automation-practice-form> (дата обращения: 18.05.2023) [↑](#footnote-ref-28)
30. HTML Attribute Reference // W3Schools Online Web Tutorials : сайт. – URL: <https://www.w3schools.com/tags/ref_attributes.asp> (дата обращения: 11.04.2023) [↑](#footnote-ref-29)
31. DEMOQA Links : сайт. – URL: <https://demoqa.com/links> (дата обращения: 18.05.2023) [↑](#footnote-ref-30)
32. DEMOQA Text Box : сайт. – URL: <https://demoqa.com/text-box> (дата обращения: 18.05.2023) [↑](#footnote-ref-31)
33. DEMOQA Select Menu : сайт. – URL: <https://demoqa.com/select-menu> (дата обращения: 19.05.2023) [↑](#footnote-ref-32)
34. DEMOQA Practice Form : сайт. – URL: <https://demoqa.com/automation-practice-form> (дата обращения: 18.05.2023) [↑](#footnote-ref-33)
35. DEMOQA Alerts : сайт. – URL: <https://demoqa.com/alerts> (дата обращения: 19.05.2023) [↑](#footnote-ref-34)
36. DEMOQA Dynamic Properties : сайт. – URL: <https://demoqa.com/dynamic-properties> (дата обращения: 20.05.2023) [↑](#footnote-ref-35)
37. selenium.webdriver.support.expected\_conditions // Selenium : сайт. – URL: <https://www.selenium.dev/selenium/docs/api/py/webdriver_support/selenium.webdriver.support.expected_conditions.html> (дата обращения: 21.04.2023) [↑](#footnote-ref-36)
38. Conventions for Python test discovery // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/stable/explanation/goodpractices.html#conventions-for-python-test-discovery> (дата обращения: 26.04.2023) [↑](#footnote-ref-37)
39. Test fixture // Wikipedia : сайт. – URL: <https://en.wikipedia.org/wiki/Test_fixture#Software> (дата обращения: 28.04.2023) [↑](#footnote-ref-38)
40. How to implement xunit-style set-up // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/how-to/xunit_setup.html> (дата обращения: 28.04.2023) [↑](#footnote-ref-39)
41. PEP 20 – The Zen of Python // peps.python.org : сайт. – URL: <https://peps.python.org/pep-0020/> (дата обращения: 28.04.2023) [↑](#footnote-ref-40)
42. Adding finalizers directly // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/how-to/fixtures.html#adding-finalizers-directly> (дата обращения: 28.04.2023) [↑](#footnote-ref-41)
43. How to use skip and xfail to deal with tests that cannot succeed // pytest.org : сайт. – URL: <https://pytest.org/en/stable/how-to/skipping.html> (дата обращения: 03.05.2023) [↑](#footnote-ref-42)
44. Override a fixture on a folder (conftest) level // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/7.1.x/how-to/fixtures.html?highlight=fixture%20folder#override-a-fixture-on-a-folder-conftest-level> (дата обращения: 04.05.2023) [↑](#footnote-ref-43)
45. Oscar - Sandbox : сайт. – URL: <http://selenium1py.pythonanywhere.com/> (дата обращения: 04.05.2023) [↑](#footnote-ref-44)
46. How to parametrize fixtures and test functions // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/how-to/parametrize.html> (дата обращения: 04.05.2023) [↑](#footnote-ref-45)
47. Mozilla : сайт. – URL: <https://www.mozilla.org/en-US/firefox/new/> (дата обращения: 04.05.2023) [↑](#footnote-ref-46)
48. GitHub : сайт. – URL: <https://github.com/mozilla/geckodriver/releases> (дата обращения: 04.05.2023) [↑](#footnote-ref-47)
49. Установка драйвера geckodriver для Firefox Selenium // selenium-python : сайт. – URL: <https://selenium-python.com/install-geckodriver> (дата обращения: 04.05.2023) [↑](#footnote-ref-48)
50. How to change command line options defaults // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/example/simple.html?highlight=addoption> (дата обращения: 04.05.2023) [↑](#footnote-ref-49)
51. Plugin List // docs.pytest.org : сайт. – URL: <https://docs.pytest.org/en/latest/reference/plugin_list.html> (дата обращения: 04.05.2023) [↑](#footnote-ref-50)
52. Шпаргалка по принципам ООП // Tproger : сайт. – URL: <https://tproger.ru/translations/oop-principles-cheatsheet/> (дата обращения: 11.05.2023) [↑](#footnote-ref-51)
53. PEP 8 – Style Guide for Python Code // peps.python.org : сайт. – URL: <https://peps.python.org/pep-0008/> (дата обращения: 11.05.2023) [↑](#footnote-ref-52)
54. PEP 257 – Docstring Conventions // peps.python.org : сайт. – URL: <https://peps.python.org/pep-0257/> (дата обращения: 11.05.2023) [↑](#footnote-ref-53)
55. Don't repeat yourself // Wikipedia : сайт. – URL: <https://en.wikipedia.org/wiki/Don't_repeat_yourself> (дата обращения: 11.05.2023) [↑](#footnote-ref-54)
56. KISS principle // Wikipedia : сайт. – URL: <https://en.wikipedia.org/wiki/KISS_principle> (дата обращения: 11.05.2023) [↑](#footnote-ref-55)
57. ...почему зависимости между тестами это плохо? // barancev.github.io : сайт. – URL: <http://barancev.github.io/test-deps-are-evil/> (дата обращения: 11.05.2023) [↑](#footnote-ref-56)
58. Что такое красивый код, и как его писать? // Habr : сайт. – URL: <https://habr.com/ru/articles/266969/> (дата обращения: 11.05.2023) [↑](#footnote-ref-57)
59. Практика хорошего кода // Habr : сайт. – URL: <https://habr.com/ru/articles/206868/> (дата обращения: 11.05.2023) [↑](#footnote-ref-58)
60. Oscar - Sandbox: The shellcoder's handbook : сайт. – URL: <http://selenium1py.pythonanywhere.com/en-gb/catalogue/the-shellcoders-handbook_209/> (дата обращения: 11.05.2023) [↑](#footnote-ref-59)
61. Исключения в python. Конструкция try - except для обработки исключений // Python 3 для начинающих : сайт. –URL:<https://pythonworld.ru/tipy-dannyx-v-python/isklyucheniya-v-python-konstrukciya-try-except-dlya-obrabotki-isklyuchenij.html> (дата обращения: 26.05.2023) [↑](#footnote-ref-60)
62. WebDriver API // selenium-python.readthedocs.io : сайт. – URL: <https://selenium-python.readthedocs.io/api.html#selenium.webdriver.remote.webdriver.WebDriver.current_url> (дата обращения: 20.08.2023) [↑](#footnote-ref-61)
63. How to use skip and xfail to deal with tests that cannot succeed // pytest.org : сайт. – URL: <https://pytest.org/en/stable/how-to/skipping.html#skip-xfail-with-parametrize> (дата обращения: 29.05.2023) [↑](#footnote-ref-62)
64. UI-автотесты: как делать не стоит // habr.com : сайт. – URL: <https://habr.com/ru/companies/badoo/articles/419419/> (дата обращения: 29.05.2023) [↑](#footnote-ref-63)
65. Виды отчетов (Reports) // QA bible : сайт. – URL: <https://vladislaveremeev.gitbook.io/qa_bible/testovaya-dokumentaciya-i-artefakty-test-deliverablestest-artifacts/vidy-otchetov-reports> (дата обращения: 30.05.2023) [↑](#footnote-ref-64)
66. GitHub : сайт. – URL: <https://github.com/allure-framework> (дата обращения: 30.05.2023) [↑](#footnote-ref-65)
67. GitHub : сайт. – URL: <https://github.com/allure-framework> (дата обращения: 30.05.2023) [↑](#footnote-ref-66)
68. PyPI : сайт. – URL: <https://pypi.org/project/allure-pytest/> (дата обращения: 30.05.2023) [↑](#footnote-ref-67)
69. DEMOQA : сайт. – URL: <https://demoqa.com/> (дата обращения: 05.04.2023) [↑](#footnote-ref-68)