Автоматизация UI тестирования.

Selenium

Инструменты для автоматизации тестирования веб-приложений

Инструмент	Языки программирования	Поддерживаемые браузеры	Особенности
Selenium	Java, Python, C# JavaScript (Node.js), Ruby	Chrome, Firefox, Safari	Широкие возможности, поддержка различных языков
Cypress	JavaScript	Chrome, Firefox	Высокая скорость, удобный интерфейс
TestCafe	JavaScript, TypeScript	Все основные браузеры	Не требует плагинов, удобные средства отладки
Puppeteer	JavaScript, TypeScript	Chrome, Chromium	Высокоуровневый API, создание скриншотов
Playwright	JavaScript, TypeScript, Python	Chrome, Firefox, Safari	Поддержка нескольких браузеров, мощные средства отладки

Тренды 2024:

- Playwright набирает популярность из-за скорости и удобства.
- Cypress доминирует во frontend-тестировании.
- Appium + Selenium остаются стандартом для мобильных и кросс-браузерных тестов.

В 2018 году **WebDriver стал рекомендацией World Wide Web Consortium (W3C)**. Что это значит? Основные поставщики браузеров (Mozilla, Google, Apple, Microsoft) поддерживают WebDriver и постоянно работают над улучшением браузеров и кода управления браузерами, что приводит к более единообразному поведению в разных браузерах, делая ваши скрипты автоматизации более стабильными.

Selenium был создан Джейсоном Хаггинсом в 2004 году. Инженер ThoughtWorks, он работал над веб-приложением, которое требовало частого тестирования. Он создал программу на JavaScript «JavaScriptTestRunner», которая автоматически контролировала действия браузера. Это было началом….

Далее Selenium активно развивался стараниями разных разработчиков.

Название «Selenium» (в переводе с англ. — селен) стало использоваться после того, как в одном из своих электронных писем ***Хаггинс*** *пошутил о конкурирующем проекте, имеющем название «Mercury Interactive QuickTest Professional» (в переводе с англ. — ртуть), написав о том, что можно вылечиться от отравления ртутью, принимая с пищей селен.*

Основными понятиями в Selenium Webdriver являются:

- ✓ **Webdriver** - самая важная сущность, ответственная за управление браузером. Основной ход скрипта строится именно вокруг экземпляра этой сущности.
- ✓ **Webelement** - вторая важная сущность, представляющая собой абстракцию над веб-элементом (кнопки, ссылки, поля ввода и др.). Webelement инкапсулирует методы для взаимодействия пользователя с элементами и получения их текущего статуса.
- ✓ **By** - абстракция над локатором веб-элемента. Этот класс инкапсулирует информацию о селекторе(например, CSS), а также сам локатор элемента, то есть всю информацию, необходимую для нахождения нужного элемента на странице.

WebDriver: Basic concepts

Java

1 WEBDRIVER

2 WEBELEMENT

3 BY

```
public void simpleScenario() {
    WebDriver driver = new ChromeDriver();
    driver.get("http://www.google.com/");
    By searchBtnSelector = By.xpath("//input[@name='btnK']");
    WebElement searchButton = driver.findElement(searchBtnSelector);
    searchButton.click();
    driver.quit();
}
```

Working with browser

Looking for elements

Working with elements

Java

Python

Для поиска элементов доступно два метода:

1. Первый - найдёт только первый элемент, удовлетворяющий локатору:

```
WebElement element = driver.findElement(By.id("#element_id")));
```

```
elem = driver.find_element(By.ID, "element_name")
```

2. Второй - вернёт весь список элементов, удовлетворяющих запросу:

```
List elements = driver.findElements(By.name("elements_name"))
```

```
elems = driver.find_elements(By.ID, "element_name")
```

Типы локаторов

Java

Поскольку Webdriver - это инструмент для автоматизации веб приложений, то большая часть работы с ним это работа с веб элементами. **WebElements - ни что иное, как DOM объекты, находящиеся на веб странице.** А для того, чтобы осуществлять какие-то действия над DOM объектами / веб элементами необходимо их точным образом определить(найти).

```
WebElement element = driver.findElement(By.<Selector>);
```

Таким образом в Webdriver определяется нужный элемент.

By - класс, содержащий статические методы для идентификации элементов.

Java

Python

1. By.id

```
WebElement element = driver.findElement(By.id("element_id"));
```

```
elem = driver.find_element(By.ID, "element_name")
```

2. By.name

```
WebElement element = driver.findElement(By.name("element_name"));
```

```
elem = driver.find_element(By.NAME, "element_name")
```

3. By.className

```
WebElement element = driver.findElement(By.className("element_class"));
```

```
elem = driver.find_element(By.CLASS_NAME, "element_name")
```

4. By.TagName

```
List<WebElement> elements = driver.findElements(By.tagName("a"));
```

5. By.LinkText

```
WebElement element = driver.findElement(By.linkText("text"));
```

6. By.PartialLinkText

```
WebElement element = driver.findElements(By.partialLinkText("text"));
```

7. By.cssSelector

Java

Python

```
<div class='main'>
    <p>text</p>
    <p>Another text</p>
</div>
```

```
WebElement element=driver.FindElement(By.cssSelector("div.main"));
```

```
elem = driver.find_element(By.CSS_SELECTOR, "div.main")
```

8. By.XPath

```
<div class='main'>
    <p>text</p>
    <p>Another text</p>
</div>
```

```
WebElement element = driver.findElement(By.xpath("//div[@class='main']"));
```

```
elem = driver.find_element(By.XPATH, «//div[@class='main']")
```

Пример

```
<body>
  <header> … </header>
  <main role="main">
    <section class="jumbotron text-center"> … </section>
    <div class="album py-5 bg-light">
      <div class="container">
        <div class="row"> flex
          <div class="col-sm-4">
            <div class="card mb-4 box-shadow"> … </div> flex
          </div>
          <div class="col-sm-4">
            <div class="card mb-4 box-shadow"> flex
              <img class="card-img-top" data-src="holder.js/100px225?theme=thumb&bg=55595c&fg
              [100%x225]" src="images/serious_cat.jpg" data-holder-rendered="true"> == $0
              <div class="card-body"> … </div>
            </div>
          </div>
```

Селекторы, которые генерирует браузер по кнопке "скопировать css селектор" или расширения, зачастую строят полный путь, начиная от body — а это очень нестабильный селектор, писать такие в своем коде это плохая практика.

При малейшем изменении структуры страницы все ваши селекторы потеряют актуальность.

Сгенерированный	body > main > div > div > div > div:nth-child(2) > div > img
Написанный вручную	div.col-sm-4:nth-child(2) img

Из доклада QA Lead https://www.youtube.com/watch?v=RFUMO7XTAQY&list=PL8761XQAJnradvEvL8QLs4VloEqtOWDZj

Селекторы бывают 2-х видов

- **Ненормальные** (правой кнопкой мыши - copy selector)

```
body > div.wrapper > div.profile > div > div >
main > div.profile-game-headline >
div.mod-static.profile-game-select >
div.profile-game-select-value.js-select-current-
item > span > span.profile-game-select-title
```

- **Все остальные**

20

Взаимодействие с веб-элементами

Над элементом можно выполнить только 5 основных команд:

- click (applies to any element)
- send keys (only applies to text fields and content editable elements)
- clear (only applies to text fields and content editable elements)
- submit (only applies to form elements)
- select (see Select List Elements)

```
# Отметить checkbox "I'm the robot".
option1 = driver.find_element(By.CSS_SELECTOR, value: "[for='robotCheckbox']")
option1.click()
```

Действия с клавиатурой

https://www.selenium.dev/documentation/webdriver/actions_api/keyboard/

Функция send_keys() принимает различные ключи в качестве параметра. Поэтому необходимо импортировать ключи перед использованием этой функции.

```
1   from selenium import webdriver
2   from selenium.webdriver.common.keys import Keys
3   from selenium.webdriver.common.by import By
4
5   driver = webdriver.Chrome()
6   driver.get("http://www.python.org")
7   elem = driver.find_element(By.NAME, value: "q")
8   elem.clear()
9   elem.send_keys("pycon")
10  elem.send_keys(Keys.RETURN)
11  assert "No results found." not in driver.page_source
12  driver.close()
```

Ввод в текстовое поле

Нажатие клавиши Ввод

Действия с клавиатурой

С помощью клавиатуры можно выполнить только 2 действия: **нажатие клавиши и отпускание нажатой клавиши**. Помимо поддержки символов ASCII, каждая клавиша клавиатуры имеет представление, которое можно нажимать или отпускать в определенных последовательностях.

В Selenium можно выполнять все операции с клавиатурой. Класс **selenium.webdriver.common.keys** содержит различные методы, которые можно использовать для этой цели.

```
ActionChains(driver)\
    .key_down(Keys.SHIFT)\
    .send_keys("a")\
    .key_up(Keys.SHIFT)\
    .send_keys("b")\
    .perform()
```

ActionChains — это способ автоматизации низкоуровневых взаимодействий, таких как движения мыши, действия кнопок мыши, нажатия клавиш и взаимодействия с контекстным меню. Это полезно для выполнения более сложных действий, таких как наведение и перетаскивание.

Генерация действий пользователя.

Когда вы вызываете методы для действий на объекте ActionChains, действия сохраняются в очереди в объекте ActionChains. Когда вы вызываете perform(), события запускаются в том порядке, в котором они поставлены в очередь.

ActionChains можно использовать в цепочке:

```
ActionChains(driver).move_to_element(menu).click(hidden_submenu).perform()
```

Или действия можно выстроить в очередь одно за другим, а затем выполнить:

```
actions = ActionChains(driver)
actions.move_to_element(menu)
actions.click(hidden_submenu)
actions.perform()
```

https://www.selenium.dev/selenium/docs/api/py/webdriver/selenium.webdriver.common.action_chains.html

```
from selenium import webdriver
from selenium.webdriver.common.keys import Keys
from selenium.webdriver.common.action_chains import ActionChains

driver = webdriver.Firefox(executable_path="C:\geckodriver.exe")
driver.get("url")

actions = ActionChains(driver)
actions.send_keys(value=username)
actions.send_keys(keys.TAB)
actions.send_keys(value=password)
actions.send_keys(keys.ENTER)
actions.perform()
driver.quit()
```

Действия с мышью

https://www.selenium.dev/documentation/webdriver/actions_api/mouse/

С помощью мыши можно выполнить только 3 действия: нажать кнопку, отпустить нажатую кнопку и переместить мышь.

```
clickable = driver.find_element(By.ID, "click")
ActionChains(driver)\
    .click(clickable)\
    .perform()
```

```
clickable = driver.find_element(By.ID, "clickable")
ActionChains(driver)\
    .click_and_hold(clickable)\
    .perform()
```

Действия с мышью

Щелчок правой кнопкой.

```
clickable = driver.find_element(By.ID, "clickable")
ActionChains(driver)\
    .context_click(clickable)\
     .perform()
```

Двойной щелчок

```
clickable = driver.find_element(By.ID, "clickable")
ActionChains(driver)\
     .double_click(clickable)\
     .perform()
```

Действия с мышью

Перейти к элементу.

```
hoverable = driver.find_element(By.ID, "hover")
ActionChains(driver)\
    .move_to_element(hoverable)\
    .perform()
```

Перетащить элемент

```
draggable = driver.find_element(By.ID, "draggable")
droppable = driver.find_element(By.ID, "droppable")
ActionChains(driver)\
    .drag_and_drop(draggable, droppable)\
    .perform()
```

Настройка ожиданий

Ожидания - непременный атрибут любых UI тестов для динамических приложений. Нужны они для синхронизации работы UI и тестового скрипта.

Скрипт выполняется намного быстрее реакции приложения на команды, поэтому часто в скриптах необходимо дожидаться определенного состояния приложения для дальнейшего с ним взаимодействия.

Самый простой пример - переход по ссылке и нажатие кнопки:

```
driver.get("http://google.com")
driver. find_element(By.ID("element_id")).click();
```

В данном случае необходимо дождаться пока не появится кнопка с id = element_id и только потом совершать действия над ней. Для этого и существуют ожидания.

Для ожиданий можно использовать библиотеку **time** в Python.

```
from selenium import webdriver
from selenium.webdriver.common.by import By
import time

browser = webdriver.Chrome()
browser.get("http://suninjuly.github.io/wait1.html")

time.sleep(1)
button = browser.find_element(By.ID, "verify")
button.click()
message = browser.find_element(By.ID, "verify_message")

assert "successful" in message.text
```

Вывод: решение с time.sleep() плохое:

оно не масштабируемое и трудно поддерживаемое.

- А если другие элементы тоже будут появляться с задержкой?
- А если изменится время задержки?
- А еще на разных машинах с разной скоростью интернета элементы могут появляться через разные промежутки времени.

Selenium Waits (Implicit Waits)

Ожидания бывают:

Неявные ожидания - Implicit Waits - конфигурируют экземпляр WebDriver делать многократные попытки найти элемент (элементы) на странице в течение заданного периода времени, если элемент не найден сразу.

```
from selenium import webdriver
from selenium.webdriver.common.by import By

browser = webdriver.Chrome()
# говорим WebDriver искать каждый элемент в течение 5 секунд
browser.implicitly_wait(5)

browser.get("http://suninjuly.github.io/wait1.html")

button = browser.find_element(By.ID, "verify")
button.click()
message = browser.find_element(By.ID, "verify_message")

assert "successful" in message.text
```

В данном примере наличие элемента будет проверяться каждые 500 мс. Как только элемент будет найден, мы сразу перейдем к следующему шагу в тесте.

Selenium Waits (Explicit Waits)

Методы find_element проверяют только то, что элемент появился на странице. В то же время элемент может иметь дополнительные свойства, которые могут быть важны для наших тестов. Рассмотрим пример с кнопкой, которая отправляет данные:

- ✓ Кнопка может быть неактивной, то есть её нельзя кликнуть;
- ✓ Кнопка может содержать текст, который меняется в зависимости от действий пользователя. Например, текст "Отправить" после нажатия поменяется на "Отправлено";
- ✓ Кнопка может быть перекрыта каким-то другим элементом или быть невидимой.

> ***Пример**: Если мы хотим в тесте кликнуть на кнопку, а она в этот момент неактивна, то WebDriver все равно проэмулирует действие нажатия на кнопку, но данные не будут отправлены. Чтобы тест был надежным, нужно не только найти кнопку на странице, но и дождаться, когда кнопка станет кликабельной.*

Для реализации подобных ожиданий в Selenium WebDriver существует понятие **явных ожиданий (Explicit Waits)**, которые позволяют задать специальное ожидание для конкретного элемента.

Задание явных ожиданий реализуется с помощью инструментов WebDriverWait и expected_conditions.

```
from selenium.webdriver.common.by import By
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC
from selenium import webdriver

browser = webdriver.Chrome()

browser.get("http://suninjuly.github.io/wait2.html")

# говорим Selenium проверять в течение 5 секунд, пока кнопка не станет кликабельной
button = WebDriverWait(browser, 5).until(
        EC.element_to_be_clickable((By.ID, "verify"))
    )
button.click()
message = browser.find_element(By.ID, "verify_message")

assert "successful" in message.text
```

```python
# говорим Selenium проверять в течение 5 секунд, пока кнопка не станет кликабельной
button = WebDriverWait(browser, 5).until(
        EC.element_to_be_clickable((By.ID, "verify"))
    )
button.click()
```

Правило ожидания

В модуле expected_conditions есть много других правил, которые позволяют реализовать необходимые ожидания:

- title_is
- title_contains
- visibility_of_element_located
- visibility_of
- presence_of_all_elements_located
- text_to_be_present_in_element
- element_to_be_clickable
- element_to_be_selected
- element_located_to_be_selected
- element_selection_state_to_be
- element_located_selection_state_to_be
- alert_is_present . . .

Описание каждого правила можно найти на сайте.

Проверка наличия элементов в Selenium

https://qarocks.ru/elements-presence-verifying-in-selenium/

Проверка наличия элементов страницы в Selenium чаще всего осуществляется при помощи трех команд:

- **verifyElementPresent / verifyElementNotPresent** – для проверки наличия или отсутствия элемента
- **verifyTextPresent / verifyTextNotPresent** – для проверки наличия или отсутствия текста
- **verifyElementPositionLeft / verifyElementPositionTop** – для проверки местоположения элемента на странице

Exceptions

Если при поиске элемента произойдет ошибка,
то WebDriver выбросит одно из следующих исключений:

- ✓ **NoSuchElementException** - если элемент не был найден за отведенное время.
- ✓ **StaleElementReferenceException** - если элемент был найден в момент поиска, но при последующем обращении к элементу DOM изменился. Например, мы нашли элемент Кнопка и через какое-то время решили выполнить с ним уже известный нам метод click. Если кнопка за это время была скрыта скриптом, то метод применять уже бесполезно — **элемент "устарел" (stale)** и мы увидим исключение.
- ✓ **ElementNotVisibleException** - если элемент был найден в момент поиска, но сам элемент невидим (например, имеет нулевые размеры), и реальный пользователь не смог бы с ним взаимодействовать.

Знание причин появления исключений помогает отлаживать тесты и понимать, где находится баг в случае его возникновения.

Настройка параметров Selenium WebDriver

1. Импорт опций Chrome

```
from selenium import webdriver
from selenium.webdriver.chrome.options import Options
```

2. Инициализация опций Chrome

```
chrome_options = Options()
```

3. Добавление желаемых возможностей

```
chrome_options.add_argument("--disable-extensions")
```

4. Добавление желаемых возможностей сессии

```
driver = webdriver.Chrome(chrome_options=chrome_options)
```

Список доступных и наиболее часто используемых аргументов для класса ChromeOptions:

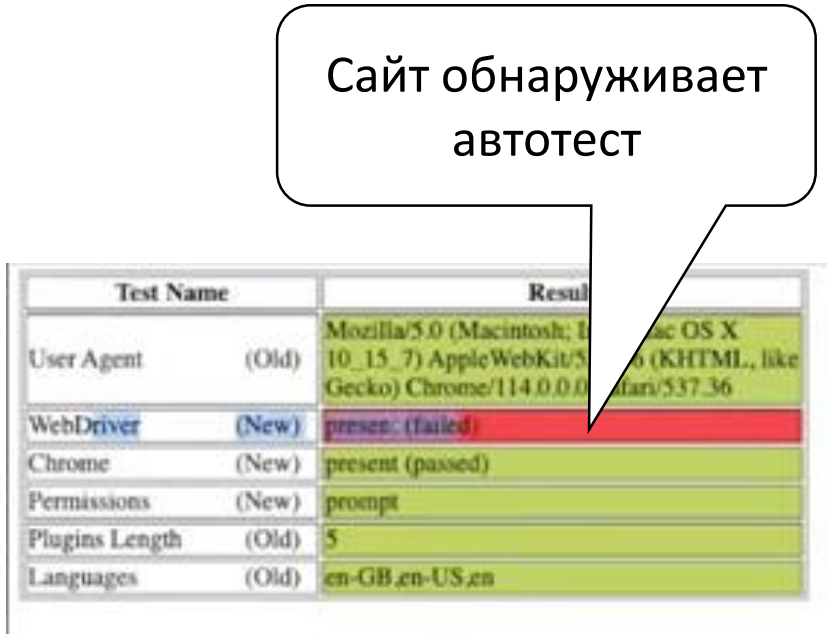
- **start-maximized**: открывает Chrome в режиме максимизации.
- **incognito**: открывает Chrome в режиме инкогнито.
- **headless**: открывает Chrome в безголовом режиме.
- **disable-extensions**: отключает существующие расширения в браузере Chrome
- **disable-popup-blocking**: отключает всплывающие окна, отображаемые в браузере Chrome
- **make-default-browser**: делает Chrome браузером по умолчанию
- **version**: выводит версию браузера Mhrome
- **disable-infobars**: запрещает Chrome отображать уведомление «Chrome управляется автоматическим программным обеспечением».

УПРАВЛЕНИЕ WEBDRIVER-МОДОМ И USER-AGENT

Как притворится человеком

https://www.youtube.com/watch?v=bg_0nB4-DmM

Test Name		Result
User Agent	(Old)	Mozilla/5.0 (Windows NT 10.0; Win64; x64) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/130.0.0.0 Safari/537.36
WebDriver	(New)	missing (passed)
Chrome	(New)	present (passed)
Permissions	(New)	prompt
Plugins Length	(Old)	5
Languages	(Old)	ru-RU,ru,en-US,en



Этот сайт демонстрирует определение человека или робота

https://intoli.com/blog/not-possible-to-block-chrome-headless/chrome-headless-test.html

статья об этом https://intoli.com/blog/not-possible-to-block-chrome-headless/

```python
import time
from selenium import webdriver
from webdriver_manager.chrome import ChromeDriverManager
from selenium.webdriver.chrome.options import Options
from selenium.webdriver.chrome.service import Service
from selenium.webdriver.support.ui import WebDriverWait
from selenium.webdriver.support import expected_conditions as EC

options = Options()
options.add_argument("--window-size=1920,1080")
options.add_argument("--disable-blink-features=AutomationControlled")

service = Service(ChromeDriverManager().install())
driver = webdriver.Chrome(service=service, options=options)
wait = WebDriverWait(driver, 5, poll_frequency=1)

driver.get("https://intoli.com/blog/not-possible-to-block-chrome-headless/chrome-head

time.sleep(3)
```

Можно отключить видимость сайтом использование webdriver

Работа с User-Agent

Selenium использует строку User-Agent для идентификации себя при выполнении HTTP-запросов.

Можно статически изменить User-Agent

```
custom_user_agent = "Mozilla/5.0 (Windows NT 10.0; Win64; x64)
AppleWebKit/537.36 (KHTML, как Gecko) Chrome/122.0.0.0 Safari/537.36"

chrome_options = webdriver.ChromeOptions()
chrome_options.add_argument(f'--user-agent={custom_user_agent}')
```

```python
import time
from selenium import webdriver
from fake_useragent import UserAgent

ua = UserAgent()
chrome_options = webdriver.ChromeOptions()
chrome_options.add_argument(f'--user-agent={ua.random}')

driver = webdriver.Chrome(options=chrome_options)
driver.get("https://httpbin.org/user-agent")

time.sleep(10)
driver.quit()
```

Фикстуры (Fixtures)

Фикстуры в контексте PyTest — это вспомогательные функции для тестов, которые не являются частью тестового сценария.

Фикстуры можно использовать для самых разных целей:

- **подготовки тестового окружения** и очистка тестового окружения и данных после завершения теста;
- **для подключения к базе данных**, с которой работают тесты;
- **для повторного использования кода**, фикстуры позволяют разработчикам переиспользовать код настройки и очистки для множества тестов, уменьшая дублирование и упрощая изменения.;
- **подготовки данных в текущем окружении** с помощью API-методов и т.д.
- они **обеспечивают изоляцию тестов**, гарантируя, что каждый тест начинается с одного и того же известного состояния, что делает тесты более предсказуемыми и надежными.

Пример фикстуры в Pytest

```
import pytest

# Простая фикстура для подготовки данных
@pytest.fixture
def some_data():
    return [1, 2, 3, 4, 5]

# Пример использования фикстуры
def test_fixture(some_data):
    assert len(some_data) == 5
    assert sum(some_data) == 15
```

Фикстуры для подготовки и очистки тестового окружения

```python
from selenium import webdriver
from selenium.webdriver.common.by
import By link = "http://selenium1py.pythonanywhere.com/"

class TestMainPage1():

    @classmethod
    def setup_class(self):
        print("\nstart browser for test suite..")
        self.browser = webdriver.Chrome()

    @classmethod
    def teardown_class(self):
        print("quit browser for test suite..")
        self.browser.quit()

    def test_guest_should_see_login_link(self):
        self.browser.get(link)
        self.browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self):
        self.browser.get(link)
        self.browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

- **префиксы setup_*, teardown_*** отвечают за порядок исполнения фикстур: до чего-то, после чего-то.
- **постфиксы *_class, *_method** и другие отвечают за уровень применения фикстур: ко всему классу, к каждому методу в классе и тд.

Фикстуры, возвращающие значение

```python
import pytest
from selenium import webdriver
from selenium.webdriver.common.by import By

link = "http://selenium1py.pythonanywhere.com/"


@pytest.fixture
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    return browser


class TestMainPage1():
    # вызываем фикстуру в тесте, передав ее как параметр
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

Для того, чтобы функцию зарегистрировать как фикстуру, в pytest есть специальный маркер (декоратор) @pytest.fixture

```python
@pytest.fixture
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    yield browser
    # этот код выполнится после завершения теста
    print("\nquit browser..")
    browser.quit()


class TestMainPage1():
    # вызываем фикстуру в тесте, передав ее как параметр
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

setup

teardown

Из фикстуры можно передать значение в тест-сьют с помощью оператора **yield**. При этом после yield можно добавить ещё код, который будет выполнен после кейса.

Таким образом можно сказать, что **всё, что идёт до оператора yield является “setup”, а всё, что после — “teardown”** (yield может ничего и не возвращать, а просто будет разделителем, отделяющим “setup” от “teardown”).

Область видимости scope

```python
@pytest.fixture(scope="class")
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    yield browser
    print("\nquit browser..")
    browser.quit()


class TestMainPage1():

    # вызываем фикстуру в тесте, передав ее как параметр
    def test_guest_should_see_login_link(self, browser):
        print("start test1")
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")
        print("finish test1")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        print("start test2")
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
        print("finish test2")
```

Для фикстур можно задавать область покрытия фикстур.

Допустимые значения: **function**, **class**, **module**, **session**.

function: - это значение по умолчанию.

Автоиспользование фикстур

```python
@pytest.fixture
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    yield browser
    print("\nquit browser..")
    browser.quit()

@pytest.fixture(autouse=True)
def prepare_data():
    print()
    print("preparing some critical data for every test")


class TestMainPage1():
    def test_guest_should_see_login_link(self, browser):
        # не передаём как параметр фикстуру prepare_data, но она все равно выполняется
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

autouse=True указывает, что фикстуру нужно запустить для каждого теста даже без явного вызова

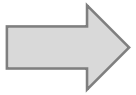
```
preparing some critical data for every test

start browser for test..
.
quit browser..

preparing some critical data for every test

start browser for test..
.
quit browser..
```

Категоризация тестов

Pytest позволяет определять категории для тестов и предоставляет опции для включения или исключения категорий при запуске набора. Тест можно отметить любым количеством категорий.

```
class TestMainPage1():

    @pytest.mark.smoke
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    @pytest.mark.regression
    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

В PyTest настоятельно рекомендуется регистрировать метки явно перед использованием. Для этого создается файл *pytest.ini* в корневой директории тестового проекта



```
[pytest]
markers =
    smoke: marker for smoke tests
    regression: marker for regression tests
```

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

  pytest -s -v -m smoke *имя_файла.py*

- ✓ Чтобы **запустить все тесты, не имеющие заданную маркировку**, можно использовать инверсию. Для запуска всех тестов, не отмеченных как smoke, нужно выполнить команду:

  pytest -s -v -m "not smoke" *имя_файла.py*

- ✓ Для **запуска тестов с разными метками** можно использовать логическое ИЛИ. Запустим smoke и regression-тесты:

  pytest -s -v -m "smoke or regression" *имя_файла.py*

- ✓ Для **запуска тестов с несколькими маркировками**, нужно использовать логическое И:

  pytest -s -v -m "smoke and win10" *имя_файла.py*

```
@pytest.mark.smoke
@pytest.mark.win10
def test_guest_should_see_basket_
    browser.get(link)
    browser.find_element(By.CSS_S
```

Категоризация

- Функциональные категории
- Категории по типу теста
- Категории по приоритету
- Категории по функциональности
- Кастомные категории

```
@pytest.mark.my_custom_category
def test_custom_feature():
    assert True
```

Функциональные

- @**pytest.mark.smoke:** Обозначает тесты для быстрой проверки базовой функциональности.
- @**pytest.mark.reggression**: Обозначает тесты для проверки регрессии.
- @**pytest.mark.performance**: Тесты на производительность.
- @**pytest.mark.integration**: Тесты интеграции.
- @**pytest.mark.end_to_end**: Тесты, которые проходят через всю систему или ее большую часть.

Категории по типу теста

- @pytest.mark.unit: Маркировка для модульных тестов.
- @pytest.mark.functional: Функциональные тесты.
- @pytest.mark.api: Тесты для API.
- @pytest.mark.ui: Тесты для пользовательского интерфейса.

Категории по приоритету

- @pytest.mark.high_priority: Тесты высокого приоритета.
- @pytest.mark.medium_priority: Тесты среднего приоритета.
- @pytest.mark.low_priority: Тесты низкого приоритета.

Категории по функциональности

- @pytest.mark.login: Тесты для проверки входа в систему.
- @pytest.mark.search: Тесты для проверки поиска.
- @pytest.mark.checkout: Тесты для проверки процесса оформления заказа.

Пропуск тестов

В PyTest есть стандартные метки, которые позволяют пропустить тест при сборе тестов для запуска или запустить, но отметить особенным статусом тот тест, который ожидаемо упадёт из-за наличия бага, чтобы он не влиял на результаты прогона всех тестов.
Эти метки не требуют дополнительного объявления в pytest.ini.

```
class TestMainPage1():

    @pytest.mark.skip
    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")
```

XFail: помечать тест как ожидаемо падающий

```
class TestMainPage1():

    def test_guest_should_see_login_link(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")

    def test_guest_should_see_basket_link_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, ".basket-mini .btn-group > a")

    @pytest.mark.xfail
    def test_guest_should_see_search_button_on_the_main_page(self, browser):
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "button.favorite")
```

Когда баг починят, после запуска тест будет отмечен как **XPASS** (“unexpectedly passing” — неожиданно проходит).

xfail (reason…)

К маркировке **xfail** можно добавлять параметр **reason**. Чтобы увидеть это сообщение в консоли, при запуске нужно добавлять параметр pytest **-rx**.

```
@pytest.mark.xfail(reason="fixing this bug right now")
def test_guest_should_see_search_button_on_the_main_page(self, browser):
    browser.get(link)
    browser.find_element(By.CSS_SELECTOR, "button.favorite")
```

@pytest.mark.**xfail(condition=False, *, reason=None, raises=None, run=True, strict=xfail_strict)**

https://docs.pytest.org/en/latest/reference/reference.html#pytest.mark.xfail

Conftest.py — конфигурация тестов

Для хранения часто употребимых фикстур и хранения глобальных настроек нужно использовать файл **conftest.py,** который должен лежать в директории верхнего уровня в проекте с тестами.

conftest.py:

```python
import pytest
from selenium import webdriver
from selenium.webdriver.common.by import By

@pytest.fixture(scope="function")
def browser():
    print("\nstart browser for test..")
    browser = webdriver.Chrome()
    yield browser
    print("\nquit browser..")
    browser.quit()
```

test_conftest.py:

```python
from selenium.webdriver.common.by import By

link = "http://selenium1py.pythonanywhere.com/"

def test_guest_should_see_login_link(browser):
    browser.get(link)
    browser.find_element(By.CSS_SELECTOR, "#login_link")
```

Параметризация тестов

PyTest позволяет запустить один и тот же тест с разными входными параметрами. Для этого используется декоратор **@pytest.mark.parametrize()**.

Пример: сайт доступен для разных языков. Напишем тест, который проверит, что для сайта с русским и английским языком будет отображаться ссылка на форму логина. Передадим в тест ссылки на русскую и английскую версию главной страницы сайта.

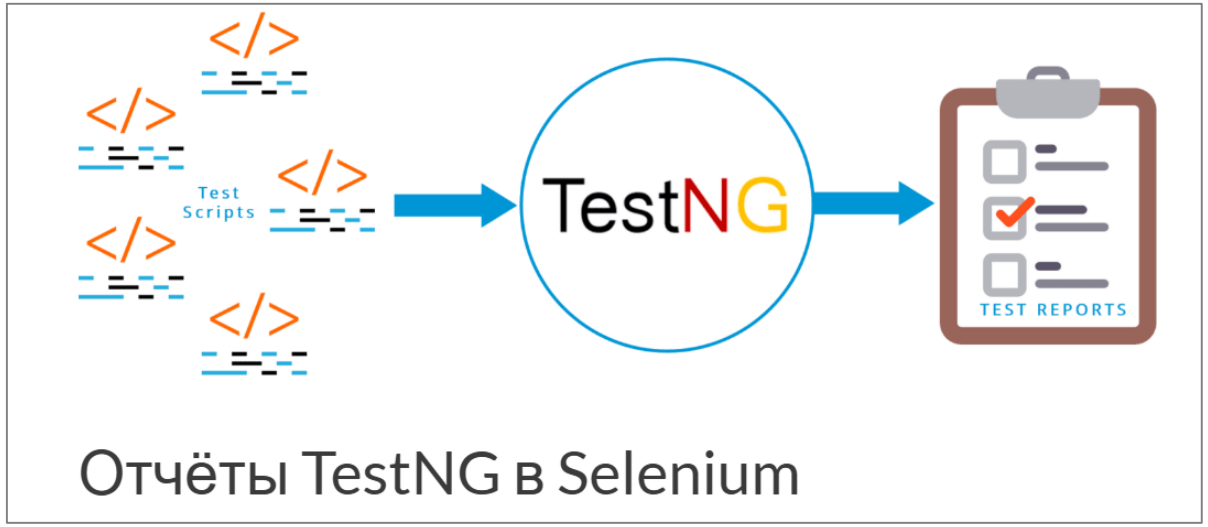
```
@pytest.mark.parametrize('language', ["ru", "en-gb"])
class TestLogin:
    def test_guest_should_see_login_link(self, browser, language):
        link = f"http://selenium1py.pythonanywhere.com/{language}/"
        browser.get(link)
        browser.find_element(By.CSS_SELECTOR, "#login_link")
        # этот тест запустится 2 раза

    def test_guest_should_see_navbar_element(self, browser, language):
        # этот тест тоже запустится дважды
```

Пример тестовой функции, реализующей проверку того, что определенный ввод приводит к ожидаемому выводу:

```
@pytest.mark.parametrize("test_input,expected", [("3+5", 8), ("2+4", 6), ("6*9", 42)])
def test_eval(test_input, expected):
    assert eval(test_input) == expected
```

Здесь декоратор определяет три разных кортежа, чтобы функция выполнялась три раза, используя их по очереди.

Отчёты



https://qarocks.ru/testng-reports-generation-in-selenium/