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».

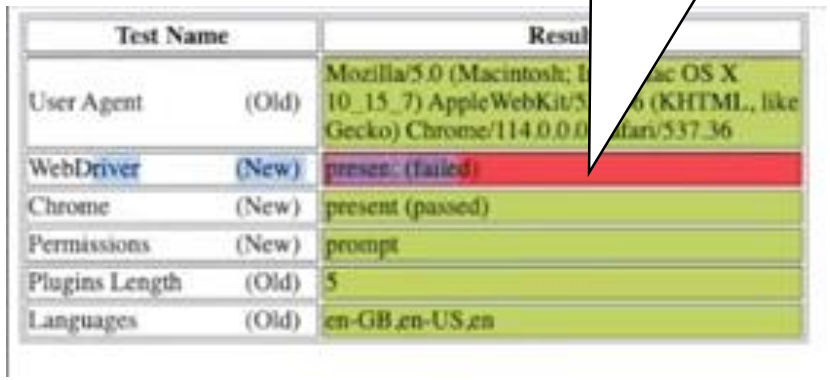
УПРАВЛЕНИЕ WEBDRIVER-МОДОМ И USER-AGENT

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

Сайт обнаруживает автотест



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

```
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

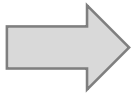
Pytest позволяет определять категории для тестов и предоставляет опции для включения или исключения категорий при запуске набора. Тест можно отметить любым количеством категорий.

```
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.