26.3. Библиотека Smart PageObject

Как известно, хороший разработчик стремится минимизировать лишние действия. Поэтому вместо создания каждый раз базовых классов для страниц и элементов можно использовать библиотеки.

Примеры библиотек и фреймворков применяемых для реализации PageObject:
"Page Objects for Python"
PyPOM — Python Page Object Model
Webium
Robot Framework Page Objects
Holmium.core

В этом модуле мы подробно рассмотрим библиотеку *Smart Page Object*, написанную нашим ведущим автором — Тимуром Нурлыгаяновым. Эта библиотека достаточно проста для понимания, но при этом вполне применима на реальных тестовых проектах.

Репозиторий библиотеки с примерами находится здесь. Скачайте и распакуйте файлы репозитория на своем ПК. Кроме библиотеки *selenium,* для её работы необходимо наличие *Python* библиотек *termcolor, allure-python-commons*. Все зависимости перечислены в файле requirements.txt.

Установить их можно консольной командой (из папки проекта):

В библиотеке есть три главных базовых класса — WebPage, WebElement и ManyWebElements (для работы со множеством элементов).

Класс WebPage предоставляет следующие методы:

- get — открытие страницы,
- go_back — возврат на предыдущую страницу,
- refresh — перегрузить страницу,
- screenshot — скриншот,
- scroll_down — прокрутка вниз,
- scroll_up — прокрутка вверх,
- switch_to_iframe — переключиться во фрейм,
- switch_out_iframe — выйти из фрейма,
- get_current_url — получить адрес текущей страницы,
- get_page_source — выдать исходный код страницы,
- check_js_errors — проверить наличие ошибок в JavaScript коде,
- wait_page_loaded — ждать загрузки.

Класс WebElement даёт нам следующие методы:

- find — поиск элементов на странице,
- delete — удалить элемент,
- wait_to_be_clickable — ждать состояния кликабельности,
- is_clickable — проверить возможно ли нажатие,
- is_presented — проверить наличие элемента,
- is_visible — проверить видимость элемента,
- wait_until_not_visible — ждать пока элемент будет невидим,
- send_keys — ввести текстовые данные в элемент,
- get_text — получить текстовое содержимое,
- get_attribute — получить атрибут тега,
- click — кликнуть по элементу,
- right_mouse_click — правый клик мыши,
- highlight_and_make_screenshot — сделать контрастность и скриншот,
- scroll_to_element — прокрутить до элемента

Несмотря на то, что многие методы по функционалу похожи на методы из библиотеки *selenium-python*, внутри них есть много полезностей и обработки исключений. Подробно рассматривать внутренности этих классов мы не будем, а сразу приступим к созданию класса страницы и тестов на ее основе. Для примера мы протестим сайт Яндекс.Маркета.

```
class MainPage(WebPage):

    def __init__(self, web_driver, url=''):
        if not url:
            url = os.getenv("MAIN_URL") or 'https://market.yandex.ru/'

        super().__init__(web_driver, url)

    # поле поиска
    search = WebElement(id='header-search')

    # кнопка поиска
    search_run_button =
WebElement(xpath='//button[@type="submit"]')

    # Название продуктов  в списке
    products_titles = ManyWebElements(xpath='//a[contains(@href,
"/product-") and @title!=""]')

    # Кнопка сортировки по цене
    sort_products_by_price = WebElement(css_selector='button[data-
autotest-id="dprice"]')

    # Цены продукта
    products_prices = ManyWebElements(xpath='//div[@data-zone-
name="price"]//span/*[1]')
```

При создании объекта мы оставляем возможность взять *url* из переданных в круглых скобках параметров, если адреса нет — проверяем наличие параметра внешнего окружения. Если нет и его — берём значение по умолчанию. Дальше мы создаем внутри класса основные элементы, с которыми предстоит работать, в виде свойств класса.

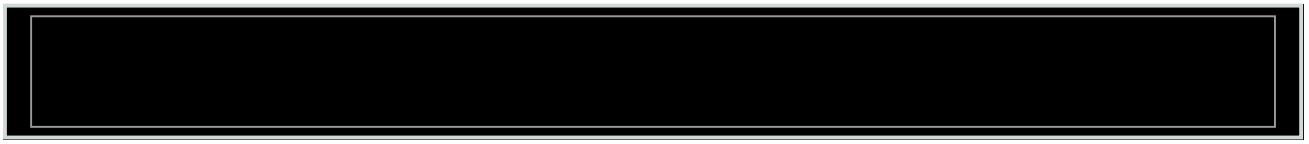
Код теста проверяющего работу поиска на странице:

```
def test_check_main_search(web_browser):
    """ Make sure main search works fine. """

    page = MainPage(web_browser)

    page.search = 'iPhone 12'
    page.search_run_button.click()

    # Проверяем  наличие продуктов на странице:
    assert page.products_titles.count() == 48

    # Проверяем наличие в названии слова iphone
    for title in page.products_titles.get_text():
        msg = 'Wrong product in search "{}"'.format(title)
        assert 'iphone' in title.lower(), msg
```

Как видно из кода, понять, что происходит на странице достаточно легко. Обратим внимание на следующие особенности решения:

Строка	Пояснение
`page.search = 'iPhone 12'`	`page.search` является объектом `WebElement`, но мы задаём ему текстовое значение, как переменной. Это возможно благодаря методу `_set_value` внутри этого класса. Этот метод запускается, когда классу назначается значение. На деле происходит поиск нужного элемента, его очистка и, если он позволяет, печать строки в текстовое поле.
`'Wrong product in search "{}"'. format(title)`	Метод `format` строкового объекта возвращает строку, где фигурные скобки заменены на переменную `title`.
`assert 'iphone' in title.lower(), msg`	`assert` проверяет истинность «зеленого» выражения и, если оно неверно, выводит `msg`.

Запустите тесты консольной командой из папки проекта:



Задание 26.3.1

1/1 point (graded)

Какие тесты завершились удачей?

- [x] test_check_main_search ✔
- [x] test_check_wrong_input_in_search ✔
- [ ] test_check_sort_by_price

✔

Отправить

Answers are displayed within the problem

Задание 26.3.2

1/1 point (graded)

Подумайте, какая логическая ошибка в условии проверки присутствует в строке:

```
assert page.products_titles.count() == 48
```

Введите исправленный ответ.

assert page.products_titles.count() >= 1 ✔

Отправить

✔ Correct (1/1 point)