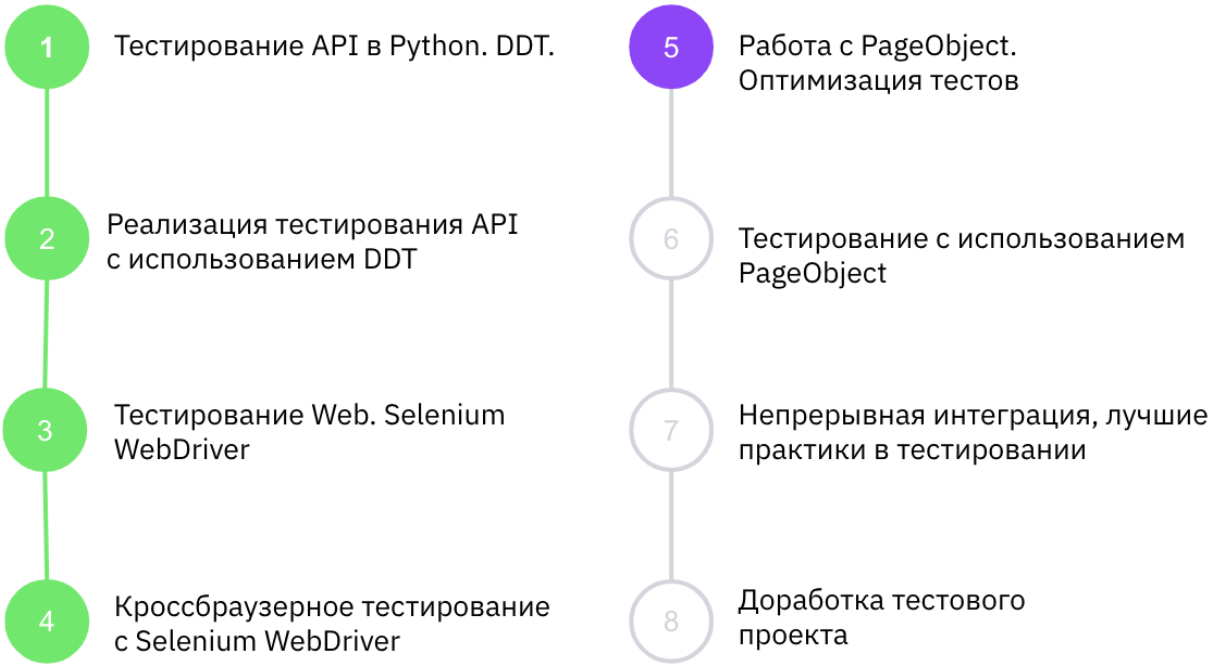
GeekBrains

Автоматизация тестирования веб-приложений на Python

Лекция 3
Работа с PageObject. Оптимизация тестов

Оглавление

Введение



На этом уроке мы узнаем:

- Как писать тесты веб-приложений с использованием общепринятого стандарта Page Object.
- Как добавить логи в проект.

Page Object — это паттерн автоматизации работы с веб-страницами и фактический стандарт в автоматизации тестирования веб-приложений. Он помогает оформить тесты стандартизировано: они будут понятны многим разработчикам, а это оптимизирует работу над проектом.

Ведение логов (то есть журналов операций) позволит быстрее отладить тесты и упростит диагностику, если при автотестах будут найдены баги.

💡 Для более детального понимания последовательности работы с примерами обязательно посмотрите видеолекцию.

Термины, используемые в лекции

Page Object — это паттерн автоматизации работы с веб-страницами, который фактически является стандартом в автоматизации тестирования веб-приложений.

Что такое Page Object

Page Object — это паттерн автоматизации работы с веб-страницами, который фактически является стандартом в автоматизации тестирования веб-приложений. Его основная идея — разделить логику тестов от реализации.

Преимущества использования:

- переиспользование кода;
- удобство понимания;
- шаблонизация для понятного чтения кода другими разработчиками тестов и так далее.

Использование паттерна Page Object подразумевает моделирование веб-страниц в виде объектов, которые взаимодействуют между собой. Сущностями внутри объекта страницы могут быть методы, функции и классы.

Каждая веб-страница тестируемого приложения описывается в виде объекта класса. Взаимодействие пользователя описывается в методах класса, а в тестах остается только бизнес-логика. Этот подход помогает избежать проблем с тестами при изменении верстки веб-приложения. В этом случае нужно поправить только класс, описывающий страницу.

Page Object, как правило, состоит из нескольких частей:

1. **Base Page** содержит необходимые методы для работы с webdriver.
2. **Page Object** содержит методы для работы с элементами на веб-страницах.
3. **Tests** реализует логику тест-кейсов.

Схема паттерна Page Object:

В примере, который мы рассматривали на предыдущей лекции, мы частично использовали ООП-подход, характерный как раз для паттерна Page Object. Но не совсем соответствовали ему.

Перепишем код с использованием Page Object. Начнем с создания фикстуры.

Фикстуры для Page Object, области действия фикстур

Фикстуры, требующие доступа к сети, зависят от подключения, а на их создание обычно уходит много времени.

Мы можем добавить параметр **scope** в декоратор фикстуры @pytest.fixture, чтобы функция-фикстура вызывалась только один раз для области действия.

Возможные значения параметра scope: function, class, module, package или session. По умолчанию он установлен в значение function. Если мы опишем фикстуру с параметром scope=”session”, то каждая тестовая функция модуля получит тот же самый объект, что позволит сэкономить время на создание подключения.

Перейдем к проекту.

Сначала нам нужно реализовать инициализацию для WebDriver. Описывать её будем в фикстуре. В файле conftest.py и реализуем функцию с именем browser.

Помечаем ее декоратором @pytest.fixture и передаем параметр scope со значением session. Это значит, что эта функция-фикстура будет исполняться только раз за тестовую сессию.

Далее описываем часть, которая будет выполняться перед тестами. В ней происходит инициализация webdriver. Далее используем конструкцию yield, которая разделяет функцию на части — до тестов и после тестов.

В части «после тестов» вызываем функцию quit, которая завершает сессию и закрывает экземпляр webdriver.

```python
import pytest, yaml
from selenium import webdriver
from selenium.webdriver.chrome.service import Service
from webdriver_manager.chrome import ChromeDriverManager
from webdriver_manager.firefox import GeckoDriverManager

with open("./testdata.yaml") as f:
    testdata = yaml.safe_load(f)
    browser = testdata["browser"]

@pytest.fixture(scope="session")
def browser():
    if browser == "firefox":
        service = Service(executable_path=GeckoDriverManager().install())
        options = webdriver.FirefoxOptions()
        driver = webdriver.Firefox(service=service, options=options)
    else:
        service = Service(executable_path=ChromeDriverManager().install())
        options = webdriver.ChromeOptions()
        driver = webdriver.Chrome(service=service, options=options)
    yield driver
    driver.quit()
```

Реализация класса Base Page

В классе BasePage определяем базовые методы для работы с WebDriver.

Создаем файл BaseApp.py.

В классе BasePage создаем конструктор, который принимает driver, — экземпляр webdriver. Указываем base_url, который будет использоваться для открытия страницы.

Далее создаем методы find_element (ищет один элемент и возвращает его) и get_element_property (получает свойство элемента).

Обернем find_element в WebdriverWait, который отвечает за явные ожидания в Selenium.

В функции определяем время, которое по умолчанию равно 10 секундам. Это время для поиска элемента. Метод go_to_site вызывает функцию get из WebDriver. Метод позволяет перейти на указываемую страницу. Передаем в него base_url.

```python
from selenium.webdriver.support import expected_conditions as EC

class BasePage:

    def __init__(self, driver):
        self.driver = driver
        self.base_url = "https://test-stand.gb.ru"

    def find_element(self, locator, time=10):
        return WebDriverWait(self.driver,
time).until(EC.presence_of_element_located(locator),

 message=f"Can't find element by locator {locator}")

    def get_element_property(self, locator, property):
        element = self.find_element(locator)
        return element.value_of_css_property(property)

    def go_to_site(self):
        return self.driver.get(self.base_url)
```

Реализация Page Object

Создаем класс TestSeacrhLocators. Он будет использоваться только для хранения локаторов. Дадим локаторам информативные названия.

В классе описываем локаторы:

- LOCATOR_LOGIN_FIELD — локатор поля ввода логина;
- LOCATOR_PASS_FIELD — локатор поля ввода пароля;
- LOCATOR_LOGIN_BTN — локатор кнопки логина;
- LOCATOR_ERROR_FIELD — локатор поля с ошибкой.

Создадим класс OperationsHelper, который наследуется от BasePage.

Реализуем вспомогательные методы:

- enter_login — ввод логина;
- enter_pass — ввод пароля;
- click_login_button — клик кнопки логина;

- get_error_text — ищет элемент с оповещением об ошибке и получает атрибут text.

Для примера переопределим время по умолчанию, установим 2 секунды.

Наш класс для веб-страницы реализуется в файле testpage.py.

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

class TestSeacrhLocators:
    LOCATOR_LOGIN_FIELD = (By.XPATH, """//*
[@id="login"]/div[1]/label/input""")
    LOCATOR_PASS_FIELD = (By.XPATH, """//*
[@id="login"]/div[2]/label/input""")
    LOCATOR_LOGIN_BTN = (By.CSS_SELECTOR, "button")
    LOCATOR_ERROR_FIELD = (By.XPATH, """//*
[@id="app"]/main/div/div/div[2]/h2""")

class OperationsHelper(BasePage):

    def enter_login(self, word):
        login_field =
self.find_element(TestSeacrhLocators.LOCATOR_LOGIN_FIELD)
        login_field.clear()
        login_field.send_keys(word)

    def enter_pass(self, word):
        pass_field =
self.find_element(TestSeacrhLocators.LOCATOR_PASS_FIELD)
        pass_field.clear()
        pass_field.send_keys(word)

    def click_login_button(self):

     self.find_element(TestSeacrhLocators.LOCATOR_LOGIN_BTN).click()

    def get_error_text(self):
        error_field =
self.find_element(TestSeacrhLocators.LOCATOR_ERROR_FIELD,time=2)
        return error_field.text
```

Оптимизация тестов

Изменим код ранее написанного теста.

```python
from testpage import OperationsHelper

def test_step1(browser):
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    testpage.enter_login("test")
    testpage.enter_pass("test")
    testpage.click_login_button()
    assert testpage.get_error_text() == "401"
```

Создаем тестовую функцию test_step1, которая будет принимать фикстуру browser.

Далее первой строчкой создаем объект страницы — testpage. Из объекта вызываем методы взаимодействия с элементами страницы.

В функции описывается верхнеуровневая логика действий пользователя.

Перенесем все, что реализовали, на схему, аналогично схеме Page Object.

Итак, мы переписали наш код в соответствии с практиками Page Object. Количество кода увеличилось, но тесты стали логичнее и понятнее.

Логирование Python

Отчет о тестах в Pytest можно настроить для вывода довольно подробной информации о тестовых шагах. Однако зачастую для отладки и формирования отчетности нам нужно вести журналы (логи) с информацией о прохождении тестов.

В этом разделе мы научимся добавлять в проект логи.

В языке Python основной инструмент для логирования — **библиотека logging.**

Модуль logging в Python — это набор функций и классов, которые позволяют регистрировать события, происходящие во время работы кода. Этот модуль входит

в стандартную библиотеку, поэтому для его использования достаточно написать одну строку:

```
1 import logging
```

Основная функция, для работы с этим модулем — **basicConfig()**. В ней можно указать все основные настройки (по крайней мере, на базовом уровне).

У функции basicConfig() есть три основных параметра:

- **level** — уровень логирования;
- **filename** — место, куда мы направляем логи;
- **format** — вид, в котором мы сохраняем результат.

Рассмотрим эти параметры подробнее.

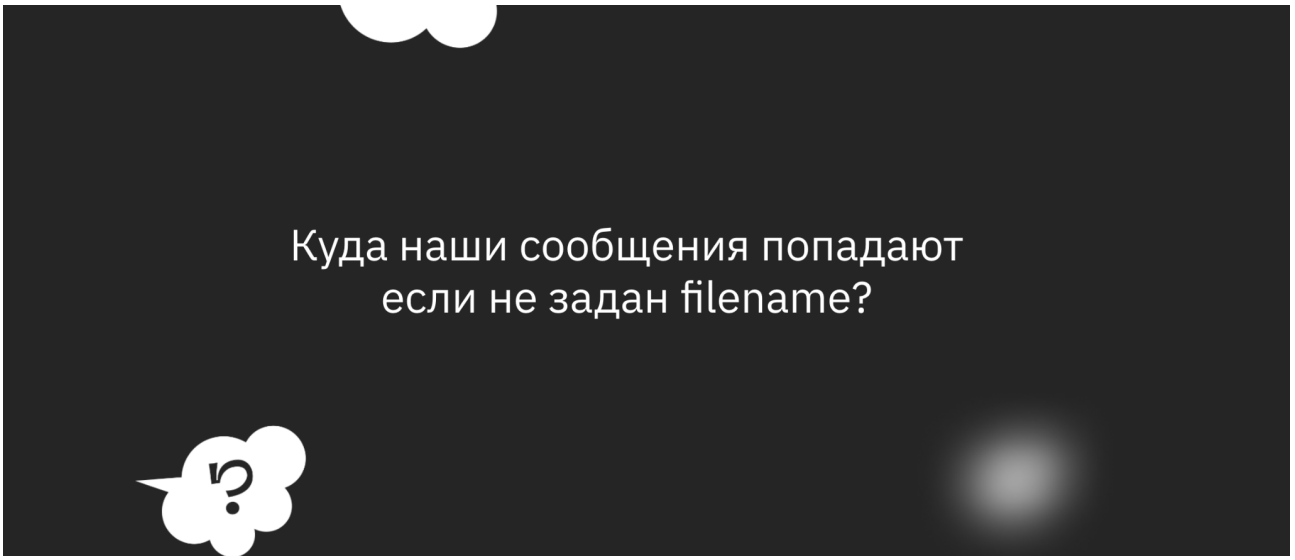
Уровень	Значение	Использование
logging.NOTSET	0	сообщения отключены
logging.DEBUG	10	подробная отладочная информация
logging.INFO	20	информационные сообщения
logging.WARNING	30	предупреждение
logging.ERROR	40	ошибка
logging.CRITICAL	50	серьезная ошибка, программа не может продолжать работу

События, которые генерирует наш код, кардинально могут отличаться по степени важности. Чтобы не засорять логи лишней информацией, в basicConfig() вы можете указать минимальный уровень фиксируемых событий.

По умолчанию фиксируются только предупреждения (WARNINGS) и события с более высоким приоритетом: ошибки (ERRORS) и критические ошибки (CRITICALS).

Чтобы записать информационное сообщение, достаточно написать код:

```
1 logging.debug('debug message')
2 logging.info('info message')
```



Ответ: за место, в которое попадают логи, отвечает параметр **filename** в basicConfig. По умолчанию все логи будут отображаться в консоли.

Направить запись лога в файл просто:

```
1 logging.basicConfig(filename = "mylog.log")
```

Последнее, с чем нам нужно разобраться, — форматирование лога. Эта опция позволяет дополнять лог полезной информацией: датой, названием файла с ошибкой, номером строки, названием метода и так далее. Сделать это можно, с помощью параметра **format**.

Например, если внутри basicConfig указать:

format = "%(asctime)s - %(levelname)s - %(funcName)s: %(lineno)d - %(message)s"

То вывод ошибки будет выглядеть так:

```
1 2023-01-16 10:35:12,468 - ERROR - <module>:1 - Hello, world!
```

По умолчанию формат такой:

```
<УРОВЕНЬ>: <ИМЯ_ЛОГГЕРА>: <СООБЩЕНИЕ>
```

Однако при использовании Pytest параметры, заданные в **logging.basicConfig,** не будут иметь силы, и файл лога не создастся. Дело в том, что pytest самостоятельно обрабатывает команды логирования. Чтобы настроить логирование в файл, мы должны создать файл pytest.ini и прописать в нем параметры:

- **log_file_format** — структура строк лога;
- **log_file_date_format** — структура метки времени;
- **log_file** — путь к файлу лога;
- **log_file_level** — уровень логирования.

Добавление логов в проект

файл pytest.ini

Добавим параметры лога.

```
[pytest]
log_file_format = %(asctime)s %(levelname)s %(message)s
log_file_date_format = %Y-%m-%d %H:%M:%S
log_file = log.txt
log_file_level = 20
```

файл test_1.py

Импортируем модуль logging и добавляем информационное сообщение logging.info.

```
from testpage import OperationsHelper
import logging

def test_step1(browser):
    logging.info("Test 1 Starting")
    testpage = OperationsHelper(browser)
    testpage.go_to_site()
    testpage.enter_login("test")
    testpage.enter_pass("test")
    testpage.click_login_button()
    assert testpage.get_error_text() == "401"
```

файл testpage.py

Добавим сообщения о том, какие действия со страницей происходят.

```python
from BaseApp import BasePage
from selenium.webdriver.common.by import By
import logging

class TestSeacrhLocators:
 ...

class OperationsHelper(BasePage):

    def enter_login(self, word):
        logging.info(f"Send '{word}' to element
{TestSeacrhLocators.LOCATOR_LOGIN_FIELD[1]}")
        login_field =
self.find_element(TestSeacrhLocators.LOCATOR_LOGIN_FIELD)
        login_field.clear()
        login_field.send_keys(word)

    def enter_pass(self, word):
        logging.info(f"Send '{word}' to element
{TestSeacrhLocators.LOCATOR_PASS_FIELD[1]}")
        pass_field =
self.find_element(TestSeacrhLocators.LOCATOR_PASS_FIELD)
        pass_field.clear()
        pass_field.send_keys(word)

    def click_login_button(self):
        logging.info("Click login button")

     self.find_element(TestSeacrhLocators.LOCATOR_LOGIN_BTN).click()

    def get_error_text(self):
        error_field =
self.find_element(TestSeacrhLocators.LOCATOR_ERROR_FIELD, time=2)
        text = error_field.text
        logging.info(f"We find text {text} in error field
{TestSeacrhLocators.LOCATOR_ERROR_FIELD[1]}")
        return text
```

Итоги

На этом уроке мы узнали:

- Как писать тесты веб-приложений с использованием общепринятого стандарта Page Object.
- Как добавить логи в проект.

Это сделало наши тесты более удобочитаемыми и упростило их отладку.

Что можно почитать еще?

1. Page Object — мнение Мартина Фаула (Martin Fowler) по поводу Page Object
2. Основы использования паттерна Page Object вместе с Selenium WebDriver

Используемая литература

1. Page Object — документация Selenium
2. Реализация паттерна Page Object на Python + pytest