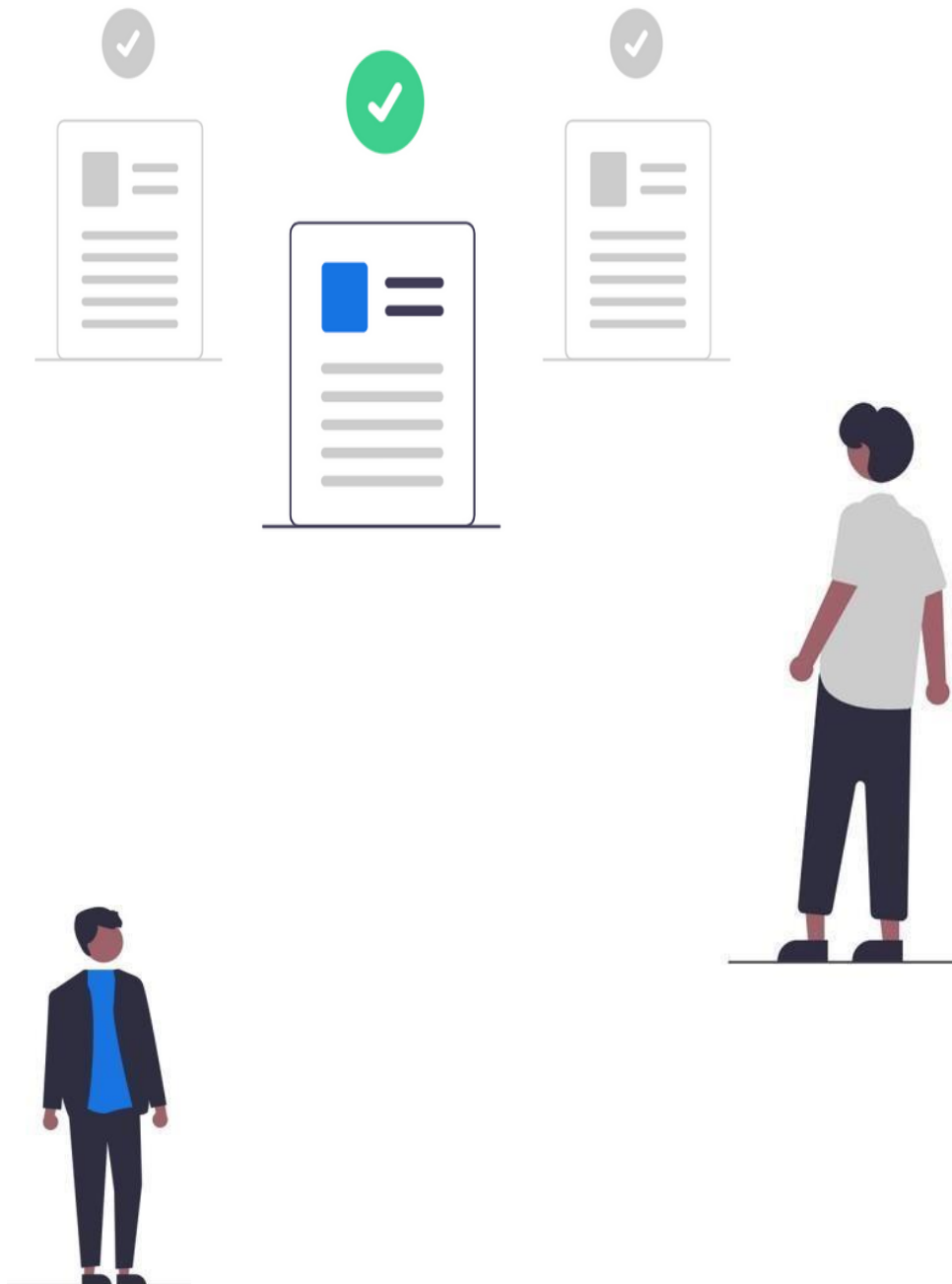


POM with Selenium in Python - Learning Guide



1. What is POM (Page Object Model)?

Page Object Model (POM) is a design pattern in Selenium that creates an object repository for storing all web elements. It helps in making the code reusable, maintainable, and easy to understand. The basic idea is to separate the logic of interaction with web elements from the tests themselves.

2. Why Use POM in Selenium?

- **Maintainability:** Changes in the UI can be handled by updating only the respective pageobject class, reducing redundancy.
- **Reusability:** Common actions are defined in page objects and can be reused across multiple test scripts.
- **Readability:** Code becomes easier to read and understand because the tests are independent of the object properties and locators.

3. Installing Selenium for Python

To start using Selenium with Python, you need to install it:

```
bash
```

```
Copy code
```

```
pip install selenium
```

You also need to install the appropriate WebDriver (e.g., ChromeDriver, GeckoDriver for Firefox) and set it up.

4. Key Components of POM

1. **Page Classes:** These represent individual pages in the application under test.
2. **Web Elements:** Each page class will contain locators for the elements on the page.
3. **Methods for Interaction:** Page classes will have methods to interact with the elements, like clicking buttons or entering text.
4. **Test Classes:** Test scripts that call these page classes to perform actions.

5. Creating a POM Structure

Here's how to create the basic structure using Python.

Step 1: Directory Structure

```
lua
```

```
Copy code
```

```
-- project_name/
```

```
-- pages/
```

```
-- home_page.py
```

```
|-- login_page.py  
|-- tests/  
    |-- test_login.py  
|-- base.py
```

6. Example Implementation

a. base.py

The base class for handling the WebDriver setup and teardown:python

Copy code

```
from selenium import webdriver  
  
class BasePage:  
    def __init__(self, driver: webdriver.Chrome):  
        self.driver = driver  
        self.driver.maximize_window()  
  
    def open(self, url: str):  
        self.driver.get(url)  
  
    def quit(self):  
        self.driver.quit()
```

b. login_page.py

A page class that contains elements and methods related to the login page:python

Copy code

```
from selenium.webdriver.common.by import By  
from base import BasePage  
  
class LoginPage(BasePage):  
    USERNAME = (By.ID, 'username')  
    PASSWORD = (By.ID, 'password')  
    LOGIN_BUTTON = (By.ID, 'loginBtn')  
  
    def enter_username(self, username: str):  
        self.driver.find_element(*self.USERNAME).send_keys(username)
```

```
def enter_password(self, password: str):  
    self.driver.find_element(*self.PASSWORD).send_keys(password)  
  
def click_login(self):  
    self.driver.find_element(*self.LOGIN_BUTTON).click()
```

c. test_login.py

The test case using the login page:

python Copy code

```
import unittest  
  
from selenium import webdriver  
  
from pages.login_page import LoginPage  
  
class TestLogin(unittest.TestCase):  
    def setUp(self):  
        self.driver = webdriver.Chrome()  
        self.login_page = LoginPage(self.driver)  
  
    def test_valid_login(self):  
        self.login_page.open("https://example.com/login")  
        self.login_page.enter_username("your_username")  
        self.login_page.enter_password("your_password")  
        self.login_page.click_login()  
        # Add assertions to validate successful login  
  
    def tearDown(self):  
        self.driver.quit()  
  
if __name__ == "__main__":  
    unittest.main()
```

7. Adding Assertions

To verify whether the actions are successful, use assertions in the test class:python

Copy code

```
self.assertTrue("Dashboard" in self.driver.title)
```

This will ensure the test passes only when the expected result is achieved.

8. Best Practices

- **DRY Principle:** Avoid repeating the same code in multiple tests.
- **Keep Locators Separate:** Store locators in a separate class if necessary.
- **Use Explicit Waits:** Instead of static waits (`time.sleep()`), use WebDriver's explicit waits for better synchronization.

python Copy code

```
from selenium.webdriver.support.ui import WebDriverWait
```

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

```
element = WebDriverWait(driver, 10).until(EC.presence_of_element_located((By.ID, 'element_id')))
```

9. Advanced Topics

Once you're comfortable with the basics, you can explore:

- **Handling Alerts and Popups**
- **File Uploads and Downloads**
- **Working with Frames**
- **Headless Browser Testing**
- **Running Tests in Parallel (e.g., with pytest-xdist)**

10. Resources for Further Learning

- **Selenium Documentation:** Selenium HQ
- **Python WebDriver API:** [Python Selenium WebDriver API](#)
- **Learn PyTest:** Explore how to use pytest for better test management and reporting.

This document should give you a strong foundation for learning and implementing POM in Selenium with Python. You can extend and refine the framework as your testing needs grow!

40

You said:

basics documentation i want

ChatGPT said:

ChatGPT

If you're looking for a basic document to understand **Page Object Model (POM)** with **Selenium in Python**, here's a straightforward guide focusing on the fundamentals:

POM with Selenium in Python – Basic Documentation

1. What is Selenium?

Selenium is a popular open-source tool used for automating web browsers. It's widely used for testing web applications across different browsers and platforms.

2. What is POM (Page Object Model)?

Page Object Model (POM) is a design pattern in Selenium that helps create a structured, modular, and maintainable way of organizing web element locators and actions. Each web page in the application is represented as a class, and the functionalities and elements on that page are encapsulated within that class.

3. Why Use POM in Selenium?

- **Maintainability:** Easier to update and maintain tests when web elements change.
- **Reusability:** Methods defined in one place can be reused across multiple tests.
- **Readability:** Simplifies test cases by abstracting out the web element locators and actions.

4. Installing Selenium for Python

To use Selenium in Python, you need to install it via **pip**:

Bash

Copy code

```
pip install selenium
```

You will also need to download a WebDriver (like ChromeDriver or GeckoDriver for Firefox) for the browser you're testing.

5. Basic Selenium Commands

Here are some common Selenium commands you'll need:

- **Launching a browser:**

python Copy code

```
from selenium import webdriver
```

```
driver = webdriver.Chrome() # or webdriver.Firefox()
```

- **Navigating to a URL:**

python Copy code

```
driver.get('https://www.example.com')
```

- **Locating elements:**

python Copy code

```
element = driver.find_element_by_id('element_id')
```

```
element = driver.find_element_by_name('element_name')
```

```
element = driver.find_element_by_xpath('//*[@name="q"]')
```

- **Performing actions on elements:**

python Copy code

```
element.click() # Clicking
```

```
element.send_keys('text') # Typing
```

- **Closing the browser:**

python Copy code

```
driver.quit()
```

6. Basic POM Structure

When implementing POM, you separate the test logic from the web element definitions. This is achieved by creating **Page Classes**.

a. Directory Structure

Here's an example of how your project can be structured:lua

Copy code

```
|-- project/  
    |-- pages/  
    |-- login_page.py  
    |-- tests/  
    |-- test_login.py  
    |-- base.py
```

7. Writing a Base Class

The base class will handle common functionalities like browser setup and teardown:python

Copy code

```
from selenium import webdriver
```

```
class BasePage:

    def __init__(self, driver: webdriver.Chrome):self.driver =
        driver

    def open_url(self, url: str):

        self.driver.get(url)

    def quit(self):

        self.driver.quit()
```

8. Creating a Page Object Class

In the POM approach, each web page has its own class that encapsulates the web elements and actions.

For example, here's a **LoginPage** class that contains the locators and actions specific to a login page:python

Copy code

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

```
class LoginPage(BasePage):

    USERNAME_FIELD = (By.ID, 'username')
    PASSWORD_FIELD = (By.ID, 'password')
    LOGIN_BUTTON = (By.ID, 'loginButton')

    def enter_username(self, username: str):

        self.driver.find_element(*self.USERNAME_FIELD).send_keys(username)

    def enter_password(self, password: str):

        self.driver.find_element(*self.PASSWORD_FIELD).send_keys(password)

    def click_login(self):

        self.driver.find_element(*self.LOGIN_BUTTON).click()Here:
```

- We use **By.ID** to locate elements using their IDs.
- The `enter_username()`, `enter_password()`, and `click_login()` methods are actions we perform on the login page.

9. Writing a Test Script

The test script will use the page objects and methods to perform actions on the web application.python

Copy code

```
import unittest

from selenium import webdriver

from pages.login_page import LoginPage

class TestLogin(unittest.TestCase):

    def setUp(self):

        self.driver = webdriver.Chrome()

        self.login_page = LoginPage(self.driver)

        self.login_page.open_url('https://www.example.com/login')

    def test_valid_login(self):

        self.login_page.enter_username('user1')

        self.login_page.enter_password('password123')

        self.login_page.click_login()

        # You can add assertions to verify login success

    def tearDown(self):

        self.login_page.quit()

if __name__ == '__main__':

    unittest.main()
```

Explanation:

- We initialize the driver and open the login page in the setUp() method.
- In the test_valid_login() method, we call methods from the LoginPage class to enter credentials and click the login button.
- After each test, we close the browser in the tearDown() method.

10. Running the Tests

You can run the test by simply executing the script: bash

Copy code

```
python -m unittest tests/test_login.py
```

11. Advantages of POM

- **Code reusability:** You can reuse methods and locators for different test cases.
- **Separation of concerns:** Test logic and UI actions are separated, making it easier to maintain.
- **Cleaner and structured code:** POM provides a modular structure to your tests.