page-objects Documentation

Release 1.0.0

Edward Easton

March 09, 2015

Contents

Page Objects are a testing pattern for websites. Page Objects model a page on your site to provide accessors and methods for interacting with this page, both to reduce boilerplate and provide a single place for element locators.

This project is an implementation of this pattern for Python using Selenium webdriver. It is agnostic to test harnesses and designed to help you build up libraries of code to test your sites.

The Python Selenium API is documented here: http://selenium-python.readthedocs.org

CHAPTER 1

Quick Example

```
>>> from page_objects import PageObject, PageElement
>>> from selenium import webdriver
>>>
>>> class LoginPage(PageObject):
        username = PageElement(id_='username')
        password = PageElement(name='password')
        login = PageElement(css='input[type="submit"]')

>>> driver = webdriver.PhantomJS()
>>> driver.get("http://example.com")
>>> page = LoginPage(driver)
>>> page.username = 'secret'
>>> page.password = 'squirrel'
>>> assert page.username.text == 'secret'
>>> page.login.click()
```

CHAPTER 2

Installation

```
$ pip install page_objects
```

CHAPTER 3

Table Of Contents

3.1 Introduction

Browser-based testing is hard to get right. There are wonderful tools out there like Seleinum that allow us to drive a web browser around our sites with the minimum of effort to provide true integration tests.

With this great power however, comes a great way to create thousands of lines of complicated test code with finicky timeouts and brittle execution. Tests that are often easier to set fire to and re-write than fix when your designers decide to change the layout of the HTML.

What Page Objects are for is to encourage you to write your browser tests in a maintainable and supportable way that makes it easier to keep the tests alive as the underlying site changes.

The Page Object model isn't new, or something that I've invented here. There are numerous examples and discussions on the web: http://lmgtfy.com/?q=Page+Object+Model

This is an implementation of the pattern for Python using Selenium webdriver that I have found useful as my time as a developer. It is not tied to any particular test framework. The codebase is tiny and has remained largely unchanged for several years. I hope you enjoy using them as much as I have!

3.2 Tutorial

3.2.1 Example HTML

Throughout this tutorial, we'll use the following simple web page with a form to demonstrate the page objects pattern.

```
<html>
    <head>
        <title>Login Page</title>
    </head>
    <body>
        <form type="POST" action="/login">
            <input type="text" name="username" id="user-input"></input>
            <input type="password" name="password"></input>
            <input type="submit">Submit</input>
        </form>
    </body>
</html>
```

Let's assume also that this page is hosted on the url http://example.com/login

3.2.2 A simple Page Object

Here's a simple page object that models this page.

```
>>> from page_objects import PageObject, PageElement
>>>
>>> class LoginPage(PageObject):
        username = PageElement(id_='user-input')
        password = PageElement(name='password')
        login = PageElement(css='input[type="submit"]')
        form = PageElement(tag_name='form')
```

3.2.3 What is a Page Object?

It is a class that, when instantiated, models a single page on your website. It has attributes on it that model elements on the page. Accessing the attributes on the Page Object instance accesses the live elements on the page, thus removing the need for your test code to worry about how to go about finding them.

In our example, the `LoginPage` class is how the test will access the login page HTML, it has three attributes of type `PageElement` that refer to the three input elements, and one for the form.

3.2.4 Page Elements and locators

Page Elements allow you to specify in exactly one place how to find a particular element on your page. They are implemented using the Descriptor protocol which you can read more about here: https://docs.python.org/3.4/howto/descriptor.html. When constructing a Page Element, you specify the locator for the element, which can be any from the following table:

Keyword Arg	Description
id_	Element ID attribute
css	CSS Selector
name	Element name attribute
class_name	Element class name
tag_name	Element HTML tag name
link_text	Anchor Element text content
partial_link_text	Anchor Element partial text content
xpath	XPath

These map directly to Selenium Webdriver's element accessor API which is documented here: http://selenium-python.readthedocs.org/en/latest/locating-elements.html

3.2.5 Using Page Objects

Page Objects are constructed with an instantiated webdriver instance, and optinally a root URI:

```
>>> from selenium import webdriver
>>> driver = webdriver.PhantomJS()
>>> page = LoginPage(driver, root_uri="http://example.com")
```

Here I've used the PhantomJS webdriver which is a convenient way to test your site without needing a full browser stack installed. The root URI is purely to provide a base for the Page Object's one and only method, `get()`:

```
>>> page.get('/login')
```

This call above instructs the browser to load the url http://example.com/login.

3.2.6 Accessing Page Elements

To access elements on the page we *get* attributes on the page object. This will return a Selenium `WebElement` instance for the selector that was specified in the PageElement's constructor. If the element is not found, it will return `None`.

For example, to check that the form above was using POST instead of GET, we would do the following.

```
>>> page.form
<selenium.webdriver.remote.webelement.WebElement object at 0x2b089299a510>
>>> assert page.form.get_attribute('type') == 'POST'
```

Here, accessing `page.form` gets the form element off of the page, allowing us to run the `WebElement.get_attribute` function to return its form type.

3.2.7 Interacting with Page Elements

We can interact with page elements in our tests as well. To type in text inputs, we can *set* attributes on the Page Object. This sends the text that we set on the attribute to the Selenium `WebElement` using its `send_keys` method.

For example, to fill in the form above, and then click the login button we would do the following:

```
>>> page.username = 'secret'
>>> page.password = 'squirrel'
>>> page.login.click()
```

3.2.8 Multi Page Elements

Sometimes we we want to access lists of elements from the page. To do this there is a `MultiPageElement` class which is constructed exactly the same as `PageElement`.

```
>>> from page_objects import PageObject, MultiPageElement
>>>
>>> class LoginPage(PageObject):
        inputs = MultiPageElement(tag_name='input')
```

When accessed, they return a list of the matching elements, or an empty list if there was nothing found.

```
>>> page.inputs
[<selenium.webdriver.remote.webelement.WebElement object at 0x2b089299a510>,
 <selenium.webdriver.remote.webelement.WebElement object at 0x2b089299a520>,
 <selenium.webdriver.remote.webelement.WebElement object at 0x2b089299a530>]
```

You can send text to them all as well:

```
>>> page.inputs = 'squirrels'
```

3.2.9 Elements with context

By default, when the `PageElement` objects are searching on the page for their matching selector, they are searching from the root of the DOM. Or to put it another way, they are searching across the whole page. Sometimes, it might be better to search within the context of another element if you have many similar items on the page.

Take this example where there are more than one login form on the page:

```
<html>
    <body>
        <form id="login-1" type="POST" action="/login-1">
            <input type="text" name="username"></input>
            <input type="password" name="password"></input>
            <input type="submit">Submit</input>
        </form>
        <form id="login-2" type="POST" action="/login-2">
            <input type="text" name="username"></input>
            <input type="password" name="password"></input>
            <input type="submit">Submit</input>
        </form>
    </body>
</html>
```

You could have separate page elements for each form input, like this:

```
>>> class LoginPage(PageObject):
        submit_1 = PageElement(css='#form-1 input[type="submit"]')
        submit_2 = PageElement(css='#form-2 input[type="submit"]')
```

However, you can also construct the page elements with the `context` flag set like this:

```
>>> class LoginPage(PageObject):
        form1 = PageElement(id_='form-1')
        form2 = PageElement(id_='form-1')
        submit = PageElement(css='input[type="submit"]', context=True)
```

This allows you to access the submit element *within* a form element, by calling the submit element like you would a method:

```
>>> page.submit(page.form1).click()
```

In this way, Page Elements with context are like 'saved searches'.

3.2.10 Accessing the Webdriver directly

You can always access the webdriver that the Page Object was constructed with direcly, using the `w` attribute.

```
>>> page.w
<selenium.webdriver.phantomjs.webdriver.WebDriver object at 0x2b0891af39d0>
>> page.w.current_url
'http://example.com/login'
```

3.3 Project History

This was originally part of the pkglib project at http://github.com/ahlmss/pkglib, it has been forked to retain history.

3.3.1 Release History

1.1.0 (2014-10-15)

- Added feature: PageElements can now be contructed with context

- Deprecated page_element and mutli_page_element factory methods

1.0.1 (2014-09-30)

- Added PageObject.get(uri) method, based off of the page's root_uri attribute.

1.0.0 (2014-09-29)

- Initial export from http://github.com/ahlmss/pkglib
- *genindex*
- *modindex*