PyPOM Documentation

Release latest

Mozilla

Dec 10, 2018

Contents

PyPOM, or Python Page Object Model, is a Python library that provides a base page object model for use with Selenium or Splinter functional tests.

It is tested on Python 2.7 and 3.6.

Contributions are welcome. Feel free to fork and contribute!

CHAPTER 1

Installation

1.1 Requirements

PyPOM requires Python >= 2.7.

1.2 Install PyPOM

To install PyPOM using pip:

```
$ pip install PyPOM
```

If you want to use PyPOM with Splinter install the optional splinter support:

```
$ pip install PyPOM[splinter]
```

To install from source:

```
$ python setup.py develop
```

CHAPTER 2

User Guide

Contents

- *Explicit waits*

2.1 Upgrading to 2.0

Plugin support was introduced in v2.0, and if you're upgrading from an earlier version you may need to make some changes to take advantage of any plugins. Before this version, to implement a custom wait for pages/regions to finish loading it was necessary to implement `wait_for_page_to_load` or `wait_for_region_to_load`. If you haven't implemented either of these, you don't need to do anything to upgrade. If you have, then whilst your custom waits will still work, we now support plugins that can be triggered after a page/region load, and these calls are made from the base classes. By overriding the default behaviour, you may be missing out on triggering any plugin behaviours. Rather than having to remember to always call the same method from the parent, you can simply change your custom wait to a new `loaded` property that returns `True` when the page/region has loaded.

So, if you have implemented your own *`wait_for_page_to_load()`* like this:

```
def wait_for_page_to_load(self):
    self.wait.until(lambda s: self.seed_url in s.current_url)
```

You will want to change it to use *`loaded`* like this:

```
@property
def loaded(self):
    return self.seed_url in self.selenium.current_url
```

Similarly, if you have implemented your own *`wait_for_region_to_load()`* like this:

```
def wait_for_region_to_load(self):
    self.wait.until(lambda s: self.root.is_displayed())
```

You will want to change it to use *`loaded`* like this:

```
@property
def loaded(self):
    return self.root.is_displayed()
```

2.2 Drivers

PyPOM requires a driver object to be instantiated, and supports multiple driver types. The examples in this guide will assume that you have a driver instance.

2.2.1 Selenium

To instantiate a Selenium driver you will need a `WebDriver` object:

```
from selenium.webdriver import Firefox
driver = Firefox()
```

2.2.2 Splinter

To instantiate a Splinter driver you will need a `Browser` object:

```
from splinter import Browser
driver = Browser()
```

2.3 Pages

Page objects are representations of web pages. They provide functions to allow simulating user actions, and providing properties that return state from the page. The *Page* class provided by PyPOM provides a simple implementation that can be sub-classed to apply to your project.

The following very simple example instantiates a page object representing the landing page of the Mozilla website:

```
from pypom import Page

class Mozilla(Page):
    pass

page = Mozilla(driver)
```

If a page has a seed URL then you can call the *open()* function to open the page in the browser. There are a number of ways to specify a seed URL.

2.3.1 Base URL

A base URL can be passed to a page object on instantiation. If no URL template is provided, then calling *open()* will open this base URL:

```
from pypom import Page

class Mozilla(Page):
    pass

base_url = 'https://www.mozilla.org'
page = Mozilla(driver, base_url).open()
```

2.3.2 URL templates

By setting a value for `URL_TEMPLATE`, pages can specify either an absolute URL or one that is relative to the base URL (when provided). In the following example, the URL https://www.mozilla.org/about/ will be opened:

```
from pypom import Page

class Mozilla(Page):
    URL_TEMPLATE = '/about/'

base_url = 'https://www.mozilla.org'
page = Mozilla(driver, base_url).open()
```

As this is a template, any additional keyword arguments passed when instantiating the page object will attempt to resolve any placeholders. In the following example, the URL https://www.mozilla.org/de/about/ will be opened:

```
from pypom import Page

class Mozilla(Page):
    URL_TEMPLATE = '/{locale}/about/'

base_url = 'https://www.mozilla.org'
page = Mozilla(driver, base_url, locale='de').open()
```

2.3.3 URL parameters

Any keyword arguments provided that are not used as placeholders in the URL template are added as query string parameters. In the following example, the URL https://developer.mozilla.org/fr/search?q=bold&topic=css will be opened:

```
from pypom import Page

class Search(Page):
    URL_TEMPLATE = '/{locale}/search'

base_url = 'https://developer.mozilla.org/'
page = Search(driver, base_url, locale='fr', q='bold', topic='css').open()
```

2.3.4 Waiting for pages to load

Whenever a driver detects that a page is loading, it does its best to block until it's complete. Unfortunately, as the driver does not know your application, it's quite common for it to return earlier than a user would consider the page to be ready. For this reason, the *loaded* property can be overridden and customised for your project's needs by returning `True` when the page has loaded. This property is polled by *wait_for_page_to_load()*, which is called by *open()* after loading the seed URL, and can be called directly by functions that cause a page to load.

The following example waits for the seed URL to be in the current URL. You can use this so long as the URL is not rewritten or redirected by your application:

```
from pypom import Page

class Mozilla(Page):

    @property
    def loaded(self):
        return self.seed_url in self.selenium.current_url
```

Other things to wait for might include when elements are displayed or enabled, or when an element has a particular class. This will be very dependent on your application.

2.4 Regions

Region objects represent one or more elements of a web page that are repeated multiple times on a page, or shared between multiple web pages. They prevent duplication, and can improve the readability and maintainability of your page objects.

2.4.1 Root elements

It's important for page regions to have a root element. This is the element that any child elements will be located within. This means that page region locators do not need to be unique on the page, only unique within the context of the root element.

If your page region contains a `_root_locator` attribute, this will be used to locate the root element every time an instance of the region is created. This is recommended for most page regions as it avoids issues when the root element becomes stale.

Alternatively, you can locate the root element yourself and pass it to the region on construction. This is useful when creating regions that are repeated on a single page.

The root element can later be accessed via the *`root`* attribute on the region, which may be necessary if you need to interact with it.

2.4.2 Repeating regions

Page regions are useful when you have multiple items on a page that share the same characteristics, such as a list of search results. By creating a page region, you can interact with any of these items in a common way:

The following example uses Selenium to locate all results on a page and return a list of `Result` regions. This can be used to determine the number of results, and each result can be accessed from this list for further state or interactions. Refer to *locating elements* for more information on how to write locators for your driver:

```
<!DOCTYPE html>
<html lang="en">
<body>
<h1>Repeated Regions Example</h1>

<ol>
    <li class="result">
        <span class="name">Result 1</span>
        <a href="./detail/1/">detail</a>
    </li>
    <li class="result">
        <span class="name">Result 2</span>
        <a href="./detail/2/">detail</a>
    </li>
    <li class="result">
        <span class="name">Result 3</span>
        <a href="./detail/3/">detail</a>
    </li>
    <li class="result">
        <span class="name">Result 4</span>
        <a href="./detail/4/">detail</a>
    </li>
</ol>

</body>
</html>
```

```
class Results(Page):
    _result_locator = (By.CLASS_NAME, "result")

    @property
    def results(self):
```

(continues on next page)

(continued from previous page)

```
        return [
            self.Result(self, el) for el in self.find_elements(*self._result_locator)
        ]

    class Result(Region):
        _name_locator = (By.CLASS_NAME, "name")
        _detail_locator = (By.TAG_NAME, "a")

        @property
        def name(self):
            return self.find_element(*self._name_locator).text

        @property
```

2.4.3 Nested regions

Regions can be nested inside other regions (i.e. a menu region with multiple entry regions). In the following example a main page contains two menu regions that include multiple repeated entry regions:

```
<!DOCTYPE html>
<html lang="en">
<body>
<h1>Nested Regions Example</h1>
<div id="page">Main Page

    <div id="menu1" class="menu">
        <ol>
            <li class="entry">Menu1-Entry1</li>
            <li class="entry">Menu1-Entry2</li>
            <li class="entry">Menu1-Entry3</li>
            <li class="entry">Menu1-Entry4</li>
            <li class="entry">Menu1-Entry5</li>
        </ol>
    </div>
    <div id="menu2" class="menu">
        <ol>
            <li class="entry">Menu2-Entry1</li>
            <li class="entry">Menu2-Entry2</li>
            <li class="entry">Menu2-Entry3</li>
        </ol>
    </div>
</div>
</body>
</html>
```

As a region requires a page object to be passed you need to pass `self.page` when instantiating nested regions:

```
class MainPage(Page):
    @property
    def menu1(self):
        root = self.find_element(By.ID, "menu1")
        return Menu(self, root=root)

    @property
    def menu2(self):
```

(continues on next page)

(continued from previous page)

```
        root = self.find_element(By.ID, "menu2")
        return Menu(self, root=root)


class Menu(Region):
    @property
    def entries(self):
        return [
            Entry(self.page, item) for item in self.find_elements(*Entry.entry_
↪locator)
        ]


class Entry(Region):
    entry_locator = (By.CLASS_NAME, "entry")

    @property
    def name(self):
        return self.root.text
```

2.4.4 Shared regions

Pages with common characteristics can use regions to avoid duplication. Examples of this include page headers, navigation menus, login forms, and footers. These regions can either be defined in a base page object that is inherited by the pages that contain the region, or they can exist in their own module:

In the following example, any page objects that extend `Base` will inherit the `header` property, and be able to check if it's displayed. Refer to *locating elements* for more information on how to write locators for your driver:

```
from pypom import Page, Region
from selenium.webdriver.common.by import By

class Base(Page):

    @property
    def header(self):
        return self.Header(self)

    class Header(Region):
        _root_locator = (By.ID, 'header')

        def is_displayed(self):
            return self.root.is_displayed()
```

2.4.5 Waiting for regions to load

The *loaded* property function can be overridden and customised for your project's needs by returning `True` when the region has loaded to ensure it's ready for interaction. This property is polled by *wait_for_region_to_load*, which is called whenever a region is instantiated, and can be called directly by functions that a region to reload.

The following example waits for an element within a page region to be displayed:

```
from pypom import Region

class Header(Region):

    @property
    def loaded(self):
        return self.root.is_displayed()
```

Other things to wait for might include when elements are displayed or enabled, or when an element has a particular class. This will be very dependent on your application.

2.5 Locating elements

Each driver has its own approach to locating elements. A suggested approach is to store your locators at the top of your page/region classes. Ideally these should be preceeded with a single underscore to indicate that they're primarily reserved for internal use. These attributes can be stored as a two item tuple containing both the strategy and locator, and can then be unpacked when passed to a method that requires the arguments to be separated.

2.5.1 Selenium

The `By` class covers the common locator strategies for Selenium. The following example shows a locator being defined and used in a page object:

```
from pypom import Page
from selenium.webdriver.common.by import By

class Mozilla(Page):
    _logo_locator = (By.ID, 'logo')

    @property
    def loaded(self):
        logo = self.find_element(*self._logo_locator)
        return logo.is_displayed()
```

2.5.2 Splinter

The available locator strategies for Splinter are:

- name
- id
- css
- xpath
- text
- value
- tag

The following example shows a locator being defined and used in a page object:

```
from pypom import Page
from selenium.webdriver.common.by import By

class Mozilla(Page):
    _logo_locator = ('id', 'logo')

    @property
    def loaded(self):
        logo = self.find_element(*self._logo_locator)
        return logo.is_displayed()
```

2.6 Explicit waits

For convenience, a `WebDriverWait` object is instantiated with an optional timeout (with a default of 10 seconds) for every page. This allows your page objects to define an explicit wait whenever an interaction causes a reponse that a real user would wait for before continuing. For example, checking a box might make a button become enabled. If we didn't wait for the button to become enabled we may try clicking on it too early, and nothing would happen. Another example of where explicit waits are common is when *waiting for pages to load* or *waiting for regions to load*.

The following example uses Selenium to demonstrate a wait that is necessary after checking a box that causes a button to become enabled. Refer to *locating elements* for more information on how to write locators for your driver:

```
from pypom import Page
from selenium.webdriver.common.by import By

class Mozilla(Page):
    _privacy_policy_locator = (By.ID, 'privacy')
    _sign_me_up_locator = (By.ID, 'sign_up')

    def accept_privacy_policy(self):
        self.find_element(*self._privacy_policy_locator).click()
        sign_me_up = self.find_element(*self._sign_me_up_locator)
        self.wait.until(lambda s: sign_me_up.is_enabled())
```

You can either specify a timeout by passing the optional `timeout` keyword argument when instantiating a page object, or you can override the `__init__()` method if you want your timeout to be inherited by a base project page class.

Note: The default timeout of 10 seconds may be considered excessive, and you may wish to reduce it. It it not recommended to increase the timeout however. If you have interactions that take longer than the default you may find that you have a performance issue that will considerably affect the user experience.

CHAPTER 3

Plugins

Plugin support was added in v2.0.

3.1 Writing plugins

PyPOM uses pluggy to enable support for plugins. In order to write a plugin you can create an installable Python package with a specific entry point. For example, the following (incomplete) `setup.py` will register a plugin named screenshot:

```
from setuptools import setup

setup(name='PyPOM-screenshot',
      description='plugin for PyPOM that takes a lot of screenshots',
      packages=['pypom_screenshot'],
      install_requires=['PyPOM'],
      entry_points={'pypom.plugin': ['screenshot = pypom_screenshot.plugin']})
```

Then, in your package implement one or more of the plugin *Hooks* provided by PyPOM. The following example will take a screenshot whenever a page or region has finished loading:

```
from pypom import hookimpl

@hookimpl
def pypom_after_wait_for_page_to_load(page):
    page.selenium.get_screenshot_as_file(page.__class__.__name__ + '.png')


@hookimpl
def pypom_after_wait_for_region_to_load(region):
    region.root.screenshot(region.__class__.__name__ + '.png')
```

CHAPTER 4

Developer Interface

This part of the documentation describes the interfaces for using PyPOM.

4.1 Page

class pypom.page.**Page**(*driver*, *base_url=None*, *timeout=10*, ***url_kwargs*)

A page object.

Used as a base class for your project's page objects.

Parameters

- **driver** (WebDriver or Browser) – A driver.
- **base_url** (*str*) – (optional) Base URL.
- **timeout** (*int*) – (optional) Timeout used for explicit waits. Defaults to 10.
- **url_kwargs** – (optional) Keyword arguments used when generating the *seed_url*.

Usage (Selenium):

```
from pypom import Page
from selenium.webdriver import Firefox

class Mozilla(Page):
    URL_TEMPLATE = 'https://www.mozilla.org/{locale}'

driver = Firefox()
page = Mozilla(driver, locale='en-US')
page.open()
```

Usage (Splinter):

```
from pypom import Page
from splinter import Browser

class Mozilla(Page):
    URL_TEMPLATE = 'https://www.mozilla.org/{locale}'

driver = Browser()
page = Mozilla(driver, locale='en-US')
page.open()
```

find_elements (*strategy*, *locator*)
Finds elements on the page.

Parameters

- **strategy** (*str*) – Location strategy to use. See By or ALLOWED_STRATEGIES.
- **locator** (*str*) – Location of target elements.

Returns List of WebElement or ElementList

Return type list

is_element_displayed (*strategy*, *locator*)
Checks whether an element is displayed.

Parameters

- **strategy** (*str*) – Location strategy to use. See By or ALLOWED_STRATEGIES.
- **locator** (*str*) – Location of target element.

Returns True if element is displayed, else False.

Return type bool

is_element_present (*strategy*, *locator*)
Checks whether an element is present.

Parameters

- **strategy** (*str*) – Location strategy to use. See By or ALLOWED_STRATEGIES.
- **locator** (*str*) – Location of target element.

Returns True if element is present, else False.

Return type bool

loaded
Loaded state of the page.

By default the driver will try to wait for any page loads to be complete, however it's not uncommon for it to return early. To address this you can override *loaded* to return True when the page has finished loading.

Returns True if page is loaded, else False.

Return type bool

Usage (Selenium):

```
from pypom import Page
from selenium.webdriver.common.by import By

```

(continues on next page)

(continued from previous page)

```
class Mozilla(Page):

    @property
    def loaded(self):
        body = self.find_element(By.TAG_NAME, 'body')
        return 'loaded' in body.get_attribute('class')
```

Usage (Splinter):

```
from pypom import Page

class Mozilla(Page):

    def loaded(self):
        body = self.find_element('tag', 'body')
        return 'loaded' in body['class']
```

Examples:

```
# wait for the seed_url value to be in the current URL
self.seed_url in self.selenium.current_url
```

open()
Open the page.

Navigates to *seed_url* and calls *wait_for_page_to_load()*.

Returns The current page object.

Return type *Page*

Raises UsageError

seed_url
A URL that can be used to open the page.

The URL is formatted from URL_TEMPLATE, which is then appended to base_url unless the template results in an absolute URL.

Returns URL that can be used to open the page.

Return type str

selenium
Backwards compatibility attribute

wait_for_page_to_load()
Wait for the page to load.

4.2 Region

class pypom.region.**Region**(*page*, *root=None*)
A page region object.

Used as a base class for your project's page region objects.

Parameters

- **page** (*Page*) – Page object this region appears in.

- **root** (WebElement or WebDriverElement) – (optional) element that serves as the root for the region.

Usage (Selenium):

```
from pypom import Page, Region
from selenium.webdriver import Firefox
from selenium.webdriver.common.by import By

class Mozilla(Page):
    URL_TEMPLATE = 'https://www.mozilla.org/'

    @property
    def newsletter(self):
        return Newsletter(self)

    class Newsletter(Region):
        _root_locator = (By.ID, 'newsletter-form')
        _submit_locator = (By.ID, 'footer_email_submit')

        def sign_up(self):
            self.find_element(*self._submit_locator).click()

driver = Firefox()
page = Mozilla(driver).open()
page.newsletter.sign_up()
```

Usage (Splinter):

```
from pypom import Page, Region
from splinter import Browser

class Mozilla(Page):
    URL_TEMPLATE = 'https://www.mozilla.org/'

    @property
    def newsletter(self):
        return Newsletter(self)

    class Newsletter(Region):
        _root_locator = ('id', 'newsletter-form')
        _submit_locator = ('id', 'footer_email_submit')

        def sign_up(self):
            self.find_element(*self._submit_locator).click()

driver = Browser()
page = Mozilla(driver).open()
page.newsletter.sign_up()
```

find_element(*strategy*, *locator*)

Finds an element on the page.

Parameters

- **strategy** (*str*) – Location strategy to use. See By or ALLOWED_STRATEGIES.
- **locator** (*str*) – Location of target element.

Returns An element.

Rytpe WebElement or WebDriverElement

find_elements (*strategy*, *locator*)
Finds elements on the page.

Parameters

- **strategy** (*str*) – Location strategy to use. See By or ALLOWED_STRATEGIES.
- **locator** (*str*) – Location of target elements.

Returns List of WebElement or ElementList

Return type list

is_element_displayed (*strategy*, *locator*)
Checks whether an element is displayed.

Parameters

- **strategy** (*str*) – Location strategy to use. See By or ALLOWED_STRATEGIES.
- **locator** (*str*) – Location of target element.

Returns True if element is displayed, else False.

Return type bool

is_element_present (*strategy*, *locator*)
Checks whether an element is present.

Parameters

- **strategy** (*str*) – Location strategy to use. See By or ALLOWED_STRATEGIES.
- **locator** (*str*) – Location of target element.

Returns True if element is present, else False.

Return type bool

loaded
Loaded state of the page region.

You may need to initialise your page region before it's ready for you to interact with it. If this is the case, you can override *loaded* to return True when the region has finished loading.

Returns True if page is loaded, else False.

Return type bool

Usage (Selenium):

```
from pypom import Page, Region
from selenium.webdriver.common.by import By

class Mozilla(Page):
    URL_TEMPLATE = 'https://www.mozilla.org/'

    @property
    def newsletter(self):
        return Newsletter(self)

    class Newsletter(Region):
        _root_locator = (By.ID, 'newsletter-form')
```

(continues on next page)

(continued from previous page)

```
        @property
        def loaded(self):
            return 'loaded' in self.root.get_attribute('class')
```

Usage (Splinter):

```
from pypom import Page, Region

class Mozilla(Page):
    URL_TEMPLATE = 'https://www.mozilla.org/'

    @property
    def newsletter(self):
        return Newsletter(self)

    class Newsletter(Region):
        _root_locator = ('id', 'newsletter-form')

        @property
        def loaded(self):
            return 'loaded' in self.root['class']
```

root
Root element for the page region.

Page regions should define a root element either by passing this on instantiation or by defining a `_root_locator` attribute. To reduce the chances of hitting `StaleElementReferenceException` or similar you should use `_root_locator`, as this is looked up every time the *root* property is accessed.

selenium
Backwards compatibility attribute

wait_for_region_to_load()
Wait for the page region to load.

4.3 Hooks

pypom.hooks.**pypom_after_wait_for_page_to_load**(*page*)
Called after waiting for the page to load

pypom.hooks.**pypom_after_wait_for_region_to_load**(*region*)
Called after waiting for the region to load

CHAPTER 5

Development

5.1 Automated Testing

All pull requests and merges are tested in Travis CI based on the `.travis.yml` file.

Usually, a link to your specific travis build appears in pull requests, but if not, you can find it on the pull requests page

The only way to trigger Travis CI to run again for a pull request, is to submit another change to the pull branch.

5.2 Running Tests

You will need Tox installed to run the tests against the supported Python versions.

```
$ pip install tox
$ tox
```

CHAPTER 6

Release Notes

CHAPTER 7

2.2.0 (2018-10-29)

7.1 Deprecations and Removals

- Removed PhantomJS support from Splinter driver due to removal in Splinter v0.9.0. (#93)

CHAPTER 8

2.1.0 (2018-08-13)

8.1 Bugfixes

- Replace use of `implprefix` with `HookimplMarker` due to deprecation.

  Existing PyPOM plugins will need to be updated to import the *hookimpl* and use it to decorate hook implementations rather than rely on the prefix of the function names.

  Before:

```
def pypom_after_wait_for_page_to_load(page):
    pass
```

  After:

```
from pypom import hookimpl

@hookimpl
def pypom_after_wait_for_page_to_load(page):
    pass (#90)
```

CHAPTER 9

2.0.0 (2018-04-17)

- Added support for plugins.
  - This introduces plugin hooks `pypom_after_wait_for_page_to_load` and `pypom_after_wait_for_region_to_load`.
  - In order to take advantage of plugin support you must avoid implementing `wait_for_page_to_load` or `wait_for_region_to_load` in your page objects.
  - This was previously the only way to implement a custom wait for your pages and regions, but now means the calls to plugin hooks would be bypassed.
  - Custom waits can now be achieved by implementing a `loaded` property on the page or region, which returns `True` when the page or region has finished loading.
  - See the user guide for more details.
- Any unused `url_kwargs` after formatting `URL_TEMPLATE` are added as URL query string parameters.

CHAPTER 10

1.3.0 (2018-02-28)

- Added support for EventFiringWebDriver
  - Thanks to @Greums for the PR

CHAPTER 11

1.2.0 (2017-06-20)

- Dropped support for Python 2.6

CHAPTER 12

1.1.1 (2016-11-21)

- Fixed packaging of `pypom.interfaces`

CHAPTER 13

1.1.0 (2016-11-17)

- Added support for Splinter
  - Thanks to @davidemoro for the PR

CHAPTER 14

1.0.0 (2016-05-24)

- Official release

Python Module Index

p

Index