bok-choy Documentation

Release 0.9.2

edX Inc.

Nov 20, 2018

Contents

UI-level acceptance test framework.

CHAPTER 1

Introduction

Bok Choy is a UI-level acceptance test framework for writing robust Selenium tests in Python.

1.1 Installation

As Bok Choy is a Python framework, you first need to install Python. If you're running Linux or Mac OS X, you probably already have it installed. We recommend that you use pip to install your Python packages:

```
pip install bok_choy
```

CHAPTER 2

Tutorial

For this tutorial, we will visit GitHub, execute a search for EdX's version of its open source MOOC platform, and verify the results returned.

2.1 Folder structure

Your test will be a Python module, so let's get started by defining it as such. Make a folder for your project, and inside that create an empty file named `__init__.py`.

```
/home/user/bok-choy-tutorial
    - __init__.py
```

```
mkdir ~/bok-choy-tutorial
cd ~/bok-choy-tutorial
touch __init__.py
```

2.2 Round 1 - The framework of a test

Let's set up and execute a simple test to make sure that all the pieces are installed and working properly.

2.2.1 Define the page

The first step is to define the page object for the page of the web application that you will be interacting with. This includes the name of the page and a method to check whether the browser is on the page. If it is possible to navigate directly to the page, we want to tell the page object how to do that too.

Create a file named pages.py in your project folder and define the GitHubSearchPage page object as follows:

```
/home/user/bok-choy-tutorial
    - __init__.py
    - pages.py
```

```
# -*- coding: utf-8 -*-
from bok_choy.page_object import PageObject


class GitHubSearchPage(PageObject):
    """
    GitHub's search page
    """

    url = 'http://www.github.com/search'

    def is_browser_on_page(self):
        return 'code search' in self.browser.title.lower()
```

2.2.2 Write a test for the page

Write the first test, which will open up a browser, navigate to the page we just defined, and verify that we got there.

Create a file named test_search.py in your project folder and use it to visit the page as follows:

```
/home/user/bok-choy-tutorial
    - __init__.py
    - pages.py
    - test_search.py
```

```
import unittest
from bok_choy.web_app_test import WebAppTest
from pages import GitHubSearchPage


class TestGitHub(WebAppTest):
    """
    Tests for the GitHub site.
    """

    def test_page_existence(self):
        """
        Make sure that the page is accessible.
        """
        GitHubSearchPage(self.browser).visit()


if __name__ == '__main__':
    unittest.main()
```

2.2.3 Execute the test

Execute the test from the command line with the following.

```
python test_search.py
```

```
.
----------------------------------------------------------------------
Ran 1 test in 3.417s

OK
```

2.2.4 What just happened?

You should have seen your default browser launch and navigate to the GitHub search page. It knew how to get there because of the page object's 'url' property.

Once the browser navigated to the page, it knew it was on the right page because the page's 'is_browser_on_page' method returned True.

2.3 Round 2 - Interacting with a page

Let's circle back around to improve the definition of the page and have the test do something more interesting, like searching for something.

2.3.1 Improve the page definition

Tip: A Best Practice for Bok Choy tests is to use css locators to identify objects.

Hint: Get to know how to use the developer tools for your favorite browser. Here are links to articles to get you started with Chrome and Firefox.

Edit your pages.py file to add in the input field where you type in text and the search button. Using the Developer Tools for my browser, I see that the input field can be identified by combining form tags id (#search_form) and input tags type (text), so its css locator would be '#search_form > input[type="text"]'.

```
<form accept-charset="UTF-8" action="/search" class="search_repos" id="search_form"␣
↪method="get">
    <input type="text" data-hotkey="s" name="q" placeholder="Search GitHub" tabindex=
↪"1" autocapitalize="off" autofocus="" autocomplete="off" spellcheck="false">
```

Add a method for filling in the search term to the page object definition like this:

```
    def enter_search_terms(self, text):
        """
        Fill the text into the input field
        """
        self.q(css='#search_form input[type="text"]').fill(text)
```

What's next? I see that type (button) and class (button) are good way to identify the search button. Its css locator would be "button.button".

```
<button class="button" type="submit" tabindex="3">Search</button>
```

We will need to define how to press the button. But we also want to define how we know that pressing the button really worked. Try it yourself in a browser. While I'm writing this tutorial, the way the GitHub search currently works is to bring you to a search results page (as long as you entered text into the input field).

So before we add the method for clicking the Search button, we should add the definition for the search results page to pages.py. If we want to use the page title again, we can see that when you search for "foo bar" it will be:

```
<title>Search · foo bar</title>
```

2.3.2 Add another page's definition

So we add the search results page definition to pages.py:

```
# -*- coding: utf-8 -*-
import re
from bok_choy.page_object import PageObject


class GitHubSearchResultsPage(PageObject):
    """
    GitHub's search results page
    """

    # You do not navigate to this page directly
    url = None

    def is_browser_on_page(self):
        # This should be something like: u'Search · foo bar · GitHub'
        title = self.browser.title
        matches = re.match(u'^Search .+ GitHub$', title)
        return matches is not None
```

2.3.3 Define the search method

Back to defining a method for pressing the button and knowing that you have arrived at the results page: We want to press the button, then wait and make sure that you have arrived at the results page before continuing on. Page objects in Bok Choy have a `wait_for_page` method that does just that.

Let's see how the method definition for pressing the search button would look.

```
class GitHubSearchPage(PageObject):
    """
    GitHub's search page
    """

    url = 'http://www.github.com/search'

    def is_browser_on_page(self):
        return 'code search' in self.browser.title.lower()

    def enter_search_terms(self, text):
        """
```

(continues on next page)

(continued from previous page)

```
        Fill the text into the input field
        """
        self.q(css='#search_form input[type="text"]').fill(text)

    def search(self):
        """
        Click on the Search button and wait for the
        results page to be displayed
        """
        self.q(css='button.btn').click()
        GitHubSearchResultsPage(self.browser).wait_for_page()

    def search_for_terms(self, text):
        """
        Fill in the search terms and click the
        Search button
        """
        self.enter_search_terms(text)
        self.search()
```

2.3.4 Add the new test

Now let's add the new test to test_search.py:

```
import unittest
from bok_choy.web_app_test import WebAppTest
from pages import GitHubSearchPage, GitHubSearchResultsPage

class TestGitHub(WebAppTest):
    """
    Tests for the GitHub site.
    """

    def setUp(self):
        """
        Instantiate the page object.
        """
        super(TestGitHub, self).setUp()
        self.github_search_page = GitHubSearchPage(self.browser)

    def test_page_existence(self):
        """
        Make sure that the page is accessible.
        """
        self.github_search_page.visit()

    def test_search(self):
        """
        Make sure that you can search for something.
        """
        self.github_search_page.visit().search_for_terms('user:edx repo:edx-platform')


if __name__ == '__main__':
    unittest.main()
```

2.3.5 Run it!

```
python test_search.py
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 8.478s

OK
```

2.3.6 What just happened?

The first test ran, just as before. Now the second test ran too: it entered the search term, hit the search button, and verified that it got to the results page.

2.4 Round 3 - Search and verify results

In the test version that we just completed we entered some search terms and then verified that we got to the right page, but not that the correct results were returned. Let's improve our test to verify the search results.

2.4.1 Improve the page definitions

Since we want to verify the results of the search, we need to add a property for the results returned to the page object for the search results page.

```
# -*- coding: utf-8 -*-
import re
from bok_choy.page_object import PageObject


class GitHubSearchResultsPage(PageObject):
    """
    GitHub's search results page
    """

    url = None

    def is_browser_on_page(self):
        # This should be something like: u'Search · foo bar · GitHub'
        title = self.browser.title
        matches = re.match(u'^Search .+ GitHub$', title)
        return matches is not None

    @property
    def search_results(self):
        """
        Return a list of results returned from a search
        """
        return self.q(css='ul.repo-list > li > div > h3 > a').text
```

Also maybe we want a better way to determine that we are on the search page than just the words "code search" the title. Let's use a query to make sure that the search button exists.

```
class GitHubSearchPage(PageObject):
    """
    GitHub's search page
    """

    url = 'http://www.github.com/search'

    def is_browser_on_page(self):
        return self.q(css='button.btn').is_present()
```

2.4.2 Improve the search test

Now we want to verify that edx-platform repo for the EdX account was returned in the search results. And not only that, but also that it was the first result. Modify the test_search.py file to do these assertions:

```
import unittest
from bok_choy.web_app_test import WebAppTest
from pages import GitHubSearchPage, GitHubSearchResultsPage

class TestGitHub(WebAppTest):
    """
    Tests for the GitHub site.
    """

    def setUp(self):
        """
        Instantiate the page object.
        """
        super(TestGitHub, self).setUp()
        self.github_search_page = GitHubSearchPage(self.browser)
        self.github_results_page = GitHubSearchResultsPage(self.browser)

    def test_page_existence(self):
        """
        Make sure that the page is accessible.
        """
        self.github_search_page.visit()

    def test_search(self):
        """
        Make sure that you can search for something.
        """
        self.github_search_page.visit().search_for_terms('user:edx repo:edx-platform')
        search_results = self.github_results_page.search_results
        assert 'edx/edx-platform' in search_results
        assert search_results[0] == 'edx/edx-platform'


if __name__ == '__main__':
    unittest.main()
```

2.4.3 Run it!

```
python test_search.py
```

```
..
----------------------------------------------------------------------
Ran 2 tests in 7.692s

OK
```

2.4.4 What just happened?

Both tests ran. We verified that we could get to the GitHub search page, then we searched for the EdX user's edx-platform repo and verified that it was the first result returned.

2.5 Take it from here!

This tutorial should have gotten you going with defining page objects for a web application and how to start to write tests against the app. Now it's up to you to take it from here and start testing your own web application. Have fun!

CHAPTER 3

Test-Design Guidelines

To ensure that your tests are robust and maintainable, you should follow these guidelines:

1. Put browser interactions in the page object, not the test.
2. Put assertions in the test, not the page object.
3. Never use `time.sleep()`
4. Always make pages wait for actions to complete.
5. Wait for JavaScript to load.

3.1 Put browser interactions in the page object, not the test

When writing tests, it is sometimes tempting to query the browser directly. For example, you might write a test like this:

```
class BarTest(WebAppTest):
    def test_bar(self):
        bar_text = self.browser.find_elements_by_css_selector('div.bar').text
        self.assertEqual(bar_text, "Bar")
```

Don't do this! There are a number of problems with this approach:

1. If the CSS selector on the page changes, you will need to change every test that uses the CSS selector.
2. Selenium calls are notoriously unreliable. They provide no retry logic to protect you from timing issues, which can cause intermittent test failures. In contrast, `bok-choy`'s higher-level interface for browser interactions include robust error-checking and retry logic.

Instead, encapsulate the browser interaction within a page object:

```
class BarPage(PageObject):
    def is_browser_on_page(self):
        return self.q(css='section#bar').is_present()
```

(continues on next page)

(continued from previous page)

```
    @property
    def text(self):
        return self.q(css='div.bar').text
        if len(text_items) > 0:
            return text_items[0]
        else:
            return ""
```

Then use the page object in a test:

```
class BarTest(WebAppTest):
    def test_bar(self):
        bar_page = BarPage(self.browser)
        self.assertEqual(bar_page.text, "Bar")
```

The page object will first check that the browser is on the correct page before trying to use the page. It will also retry if, for example, JavaScript modifies the `<div>` in between the time we retrieve it and when we get the element's text (this would result in a run-time exception otherwise). Finally, if the CSS selector on the page changes, we can modify the page object, thus updating every test that interacts with the page.

3.2 Put assertions in the test, not the page object

Page objects allow tests to interact with the pages on a site. But page objects should **not** make assertions about the page; that's the responsibility of the test.

For example, don't do this:

```
class BarPage(PageObject):
    def check_section_title(self):
        assert self.q(css='div.bar').text == ['Test Section']
```

Because the page object contains the assertion, the page object is less re-usable. If another test expects the page title to be something other than "Test Section", it cannot re-use `check_section_title()`.

Instead, do this:

```
class BarPage(PageObject):
    def section_title(self):
        text_items = self.q(css='div.bar').text
        if len(text_items) > 0:
            return text_items[0]
        else:
            return ""
```

Each test can then access the section title and assert that it matches what the test expects.

3.3 Never use `time.sleep()`

Sometimes, tests fail because when they check the page too soon. Often, tests must wait for JavaScript on the page to finish manipulating the DOM, such as when adding elements or even attaching event listeners. In these cases, it is tempting to insert an explicit wait using `time.sleep()`. For example:

```
class FooPage(PageObject):
    def do_foo(self):
        time.sleep(10)
        self.q(css='button.foo').click()
```

There are two problems with this approach:

1. Tests run more slowly, because they will always wait, even if the page is ready.
2. No matter how long the test waits, at some point it will not wait long enough. This leads to intermittent test failures.

`bok-choy` provides two mechanisms for dealing with timing issues. First, each page object checks that the browser is on the correct page *before* you can interact with the page:

```
class FooPage(PageObject):
    def is_browser_on_page(self):
        return self.q(css='section.bar').is_present()

    def do_foo(self):
        self.q(css='button.foo').click()
```

When you call `do_foo()`, the page will wait for `section.bar` to be present in the DOM.

Second, the page object can use a `Promise` to wait for the DOM to be in a certain state. For example, suppose that the page is ready when a "loading" message is no longer visible. You could check this condition using a `Promise`:

```
class FooPage(PageObject):
    def is_browser_on_page(self):
        return self.q(css='button.foo').is_present()

    def do_foo(self):
        ready_promise = EmptyPromise(
            lambda: 'Loading...' not in self.q(css='div.msg').text,
            "Page finished loading"
        ).fulfill()

        self.q(css='button.foo').click()
```

3.4 Always make pages wait for actions to complete

Page objects generally provide two ways of interacting with a page: 1. Querying the page for information. 2. Performing an action on the page.

In the second case, page objects should wait for the action to complete before returning. For example, suppose a page object has a method `save_document()` that clicks a `Save` button. The page then redirects to a different page. In this case, the page object should wait for the next page to load before returning control to the caller.

```
class FooPage(PageObject):
    def save_document():
        self.q(css='button.save').click()
        return BarPage(self.browser).wait_for_page()
```

Tests can then use this page without worrying about whether the next page has loaded:

```
def test_save(self):
    bar = FooPage(self.browser).save_document()
    self.assertEqual(bar.text, "Bar")
```

3.5 Wait for JavaScript to load

Sometimes, a page is not ready until JavaScript on the page has finished loading. This is especially problematic for pages that load JavaScript asynchronously (for example, when using RequireJS).

`bok-choy` provides a simple mechanism for waiting for RequireJS modules to load:

```
@requirejs('foo')
class FooPage(PageObject):

    @wait_for_js
    def text(self):
        return self.q(css='div.foo').text
```

This will ensure that the RequireJS module `foo` has loaded before executing `text()`.

More generally, you can wait for JavaScript variables to be defined:

```
@js_defined('window.Foo')
class FooPage(PageObject):

    @wait_for_js
    def text(self):
        return self.q(css='div.foo').text
```

CHAPTER 4

Performing Accessibility Audits

The bok-choy framework includes the ability to perform accessibility audits on web pages using either Google Accessibility Developer Tools or Dequelabs Axe Core Accessibility Engine.

In each page object's definition you can define the audit rules to use for checking that page and optionally, the scope of the audit within the webpage itself.

The general methodology for enabling accessibility auditing consists of the following steps.

- *Define the Accessibility Rules to Check for a Page*
- *(Optional) Define the Scope of Accessibility Auditing for a Page*
- Perform the audits either actively or passively.
  - Actively: *Trigger an Audit Actively and Assert on the Results Returned*
  - Passively: *Leverage Your Existing Tests and Fail on Accessibility Errors*

4.1 Define the Accessibility Rules to Check for a Page

A page object's list of audit rules to use in the accessibility audit for a page are defined in the `rules` attribute of an A11yAuditConfig object. This can be updated after instantiating the page object to be tested via the `set_rules` method.

The default is to check all the rules. To set this explicitly, pass an empty dictionary to `set_rules`.

```
page.a11y_audit.config.set_rules({})
```

To skip automatic accessibility checking for a particular page, update the page object's `page.verify_accessibility` attribute to return `False`.

To check only a specific set of rules on a particular page, pass the list of the names of the rules to that page's `A11yAudit` object's `set_rules` method as the *apply* key.

```
page.a11y_audit.config.set_rules({
    "apply": ['badAriaAttributeValue', 'imagesWithoutAltText'],
})
```

To skip checking a specific set of rules on a particular page, pass the list of the names of the rules as the first argument to that page's `A11yAudit` object's `set_rules` method as the *ignore* key.

```
page.a11y_audit.config.set_rules({
    "ignore": ['badAriaAttributeValue', 'imagesWithoutAltText'],
})
```

4.2 (Optional) Define the Scope of Accessibility Auditing for a Page

You can limit the scope of an accessibility audit to only a portion of a page. The default scope is the entire document.

To limit the scope, configure the page object's `A11yAuditConfig` object via the `set_scope` method.

For instance, to start the accessibility audit in the `div` with id `foo`, you can follow this example.

```
page.a11y_audit.config.set_scope(["div#foo"])
```

Please see the rulset specific documentation for the `set_scope` method for more details.

4.3 Trigger an Audit Actively and Assert on the Results Returned

To trigger an accessibility audit actively, call the page object class's `a11y_audit.do_audit` method and then assert on the results returned.

Here is an example of how you might write a test case that actively performs an accessibility audit.

```
from bok_choy.page_object import PageObject


class MyPage(PageObject):
    def __init__(self, *args, **kwargs):
        super(MyPage, self).__init__(*args, **kwargs)

        self.a11y_audit.config.set_rules({
            "apply": ['badAriaAttributeValue', 'imagesWithoutAltText'],
        })

    def url(self):
        return 'https://www.mysite.com/page'


class AccessibilityTest(WebAppTest):

    def test_accessibility_on_page(self):
        page = MyPage(self.browser)
        page.visit()
        report = page.a11y_audit.do_audit()

        # There was one page in this session
```

(continues on next page)

(continued from previous page)

```
        self.assertEqual(1, len(report))
        result = report[0]

        # I have already corrected any accessibility errors on my page
        # for the rules I defined in the page object, so I will assert
        # that none exist.
        self.assertEqual(0, len(result.errors))
        self.assertEqual(0, len(result.warnings))
```

4.4 Leverage Your Existing Tests and Fail on Accessibility Errors

To trigger accessibility audits passively, set the `VERIFY_ACCESSIBILITY` environment variable to `True`. Doing so triggers an accessibility audit whenever a page object's `wait_for_page` method is called. If errors are found on the page, an AccessibilityError is raised.

Note: An AccessibilityError is raised only on errors, not on warnings.

You might already have some bok-choy tests written for your web application. Here is an example of a bok-choy test that will implicity check for two specific accessibility rules.

```
from bok_choy.page_object import PageObject


class MyPage(PageObject):
    def __init__(self, *args, **kwargs):
        super(MyPage, self).__init__(*args, **kwargs)

        self.a11y_audit.config.set_rules({
           "apply": ['badAriaAttributeValue', 'imagesWithoutAltText']
        })

    def url(self):
        return 'https://www.mysite.com/page'

    def click_button(self):
        """
        Click on the button element (id="button").
        On my example page this will trigger an ajax call
        that updates the #output div with the text "yes!"
        """
        self.q(css='div#fixture button').first.click()
        self.wait_for_ajax()

    @property
    def output(self):
        """
        Return the contents of the "#output" div on the page.
        """
        text_list = self.q(css='#output').text

        if len(text_list) < 1:
            return None
```

(continues on next page)

(continued from previous page)

```
        else:
            return text_list[0]

class MyPageTest(WebAppTest):

    def test_button_click_output(self):
        page = MyPage(self.browser)
        page.visit()
        page.click_button()

        self.assertEqual(page.output, 'yes!')
```

You can reuse your existing bok-choy tests in order to navigate through the application while at the same time verifying that it is accessibile.

Before running your bok-choy tests, set the environment variable `VERIFY_ACCESSIBILITY` to `True`.

```
export VERIFY_ACCESSIBILITY=True
```

This will trigger an audit, using the rules (and optionally the scope) set in the page object definition, whenever a call to `wait_for_page()` is made.

In the case of the `test_button_click_output` test case in the example above, an audit will be done at the end of the `visit()` and `click_button()` method calls, as each of those will call out to `wait_for_page()`.

If any assessibility errors are found, then the testcase will fail with an AccessibilityError.

Note: An AccessibilityError is raised only on errors, not on warnings.

CHAPTER 5

Visual Diff Testing

The bok-choy framework uses Needle to provide the ability to capture portions of a rendered page in the browser and assert that the image captured matches that of a baseline. Needle is an optional dependency of bok-choy, which you can install via either of the following commands:

```
pip install bok-choy[visual_diff]
pip install needle
```

The general methodology for creating a test with a screenshot assertion consists of the following steps.

- *Write Your Page Object and Test Case Code to Navigate the System Under Test*
- *Add The Call to assertScreenshot*
- *Create the Initial Baseline Screenshot*
- *Execute Your Test Cases After Changes to the System Under Test*
- *Advanced Features*

5.1 Write Your Page Object and Test Case Code to Navigate the System Under Test

If you are not familiar with how to write a bok-choy page object and test case, first check out the Tutorial.

Here is an example of a test that navigates to the edx.org home page.

page.py, which contains the page object.

```
from bok_choy.page_object import PageObject

class EdxHomePage(PageObject):
    url = 'http://www.edx.org'
```

(continues on next page)

(continued from previous page)

```
    def is_browser_on_page(self):
        return 'edx' in self.browser.title.lower()
```

my_test.py, which contains the test code.

```
from bok_choy.web_app_test import WebAppTest
from page import EdxHomePage

class TestEdxHomePage(WebAppTest):

    def test_page_existence(self):
        EdxHomePage(self.browser).visit()
```

5.2 Add the Call to assertScreenshot

`assertScreenshot()` takes two arguments: a CSS selector for the element to capture, and a filename for the image.

The following example uses the same **my_test.py** test case shown in the previous section, with an assertion added to check that the site logo for the edx.org home page has not changed.

- The first argument, `img.site-logo` is the css locator for the element that we want to capture and compare.
- The second argument, `edx_logo_header` is the filename that will be used for both the baseline and the actual results. The .png extension is appended automatically.

Note: For test reliability and synchronization purposes, a bok-choy best practice is to employ Promises to ensure that the page has been fully rendered before you take the screenshot. At the very least, you should first assert that the element you want to capture is present and visible on the screen.

---

my_test.py, with the screenshot assertion.

```
from bok_choy.web_app_test import WebAppTest
from page import EdxHomePage

class TestEdxHomePage(WebAppTest):

    def test_page_existence(self):
        homepage = EdxHomePage(self.browser).visit()
        css_locator = 'img.site-logo'
        self.assertTrue(homepage.q(css=css_locator).first.visible)
        self.assertScreenshot(css_locator, 'edx_logo_header')
```

5.3 Create the Initial Baseline Screenshot

To create an initial screenshot of the logo, run the test case in "baseline saving" mode by specifying the nose parameter `--with-save-baseline`.

```
$ nosetests my_test.py --with-save-baseline
```

If using pytest, you can instead set the environment variable `NEEDLE_SAVE_BASELINE`.

```
$ NEEDLE_SAVE_BASELINE=true py.test my_test.py
```

The folder in which the baseline and actual (output) screenshots are saved is determined using the following environment variables.

- NEEDLE_OUTPUT_DIR - defaults to "screenshots"
- NEEDLE_BASELINE_DIR - defaults to "screenshots/baseline"

In our example, we would execute the test once with the save baseline parameter to create screenshots/baseline/edx_logo_header.png. We would then open it up and check that it looks okay.

5.4 Execute Your Test Cases After Changes to the System Under Test

Now if we run our tests, it will take the same screenshot and check it against the saved baseline screenshot on disk.

```
$ nosetests my_test.py
```

If a regression causes them to become significantly different, then the test will fail.

5.5 Advanced Features

See the Needle documentation for more information on the following advanced features.

- Setting the viewport's size - This feature is particularly useful for predicting the size of the resulting screenshots when taking full screen captures, and for testing responsive sites.
- Difference engine - Instead of PIL (the default), you might want to use PerceptualDiff. In addition to being much faster than PIL, PerceptualDiff generates a diff PNG file when a test fails, highlighting the differences between the baseline image and the new screenshot.
- File cleanup - Each time you run tests, new screenshot images are saved to disk, for comparison with the baseline screenshots. You might want to set your configuration to delete these files for all successful tests.

CHAPTER 6

Performing XSS Vulnerability Audits

The bok-choy framework includes the ability to perform XSS (cross-site scripting) audits on web pages using a short XSS locator defined in https://www.owasp.org/index.php/XSS_Filter_Evasion_Cheat_Sheet#XSS_Locator.

6.1 Trigger XSS Vulnerability Audits in Existing Tests

You might already have some bok-choy tests written for your web application. To leverage existing bok-choy tests and have them fail on finding XSS vulnerabilities, follow these steps.

1. Insert the `XSS_INJECTION` string defined in `bok_choy.page_object` into your page content.
2. Set the `VERIFY_XSS` environment variable to `True`.

```
export VERIFY_XSS=True
```

With this environment variable set, an XSS audit is triggered whenever a page object's `q` method is called. The audit will detect improper escaping both in HTML and in Javascript that is embedded within HTML.

If errors are found on the page, an XSSExposureError is raised.

Here is an example of a bok-choy test that will check for XSS vulnerabilities. It clicks a button on the page, and the user's name is inserted into the page. If the user name is not properly escaped, the display of the name (which is data provided by the user and thus potentially malicious) can cause XSS issues.

In the case of the `test_button_click_output` test case in the example below, an audit will be done in the `click_button()`, `output()`, and `visit()` method calls, as each of those will call out to `q`.

If any XSS errors are found, then the test case will fail with an XSSExposureError.

```
from bok_choy.page_object import PageObject, XSS_INJECTION


class MyPage(PageObject):
    def url(self):
        return 'https://www.mysite.com/page'
```

(continues on next page)

(continued from previous page)

```
    def is_browser_on_page(self):
        return self.q(css='div#fixture button').present

    def click_button(self):
        """
        Click on the button element (id="button").
        On my example page this will trigger an ajax call
        that updates the #output div with the user's name.
        """
        self.q(css='div#fixture button').first.click()
        self.wait_for_ajax()

    @property
    def output(self):
        """
        Return the contents of the "#output" div on the page.

        In the example page, it will contain the user's name after being
        updated by the ajax call that is triggered by clicking the button.
        """
        text_list = self.q(css='#output').text

        if len(text_list) < 1:
            return None
        else:
            return text_list[0]

class MyPageTest(WebAppTest):
    def setUp(self):
        """
        Log in as a particular user.
        """
        super(MyPageTest, self).setUp()
        self.user_name = XSS_INJECTION
        self.log_in_as_user(self.user_name)

    def test_button_click_output(self):
        page = MyPage(self.browser)
        page.visit()
        page.click_button()

        self.assertEqual(page.output, self.user_name)

    def log_in_as_user(self, user_name):
        """
        Would be implemented to log in as a particular user
        with a potentially malicious, user-provided name.
        """
        pass
```

CHAPTER 7

Browser Customization

Although the default browser configurations provided by bok-choy should be sufficient for most needs, sometimes you'll need to customize it a little for particular tests or even an entire test suite. Here are some of the options bok-choy provides for doing that.

7.1 Firefox Profile Preferences

Whether you use a custom profile or not, you can customize the profile's preferences before the browser is launched. To do this, create a function which takes a FirefoxProfile as a parameter and add it via the `bok_choy.browser.add_profile_customizer()` function. For example, to suppress the "unresponsive script" warning dialog that normally interrupts a test case in Firefox when running accessibility tests on a particularly long page:

```
def customize_preferences(profile):
    profile.set_preference('dom.max_chrome_script_run_time', 0)
    profile.set_preference('dom.max_script_run_time', 0)
bok_choy.browser.add_profile_customizer(customize_preferences)
```

This customization can be done in any of the normal places that test setup occurs: `setUpClass()`, a pytest fixture, the test case itself, etc. You can clear any previously-added profile customizers via the `bok_choy.browser.clear_profile_customizers()` function.

7.2 Firefox Profile Directory

Normally, selenium launches Firefox using a new, anonymous user profile. If you have a specific Firefox profile that you'd like to use instead, you can specify the path to its directory in the `FIREFOX_PROFILE_PATH` environment variable anytime before the call to `bok_choy.browser.browser()`. This passes the path to the FirefoxProfile constructor so the browser can be launched with any customizations that have been made to that profile.

CHAPTER 8

Testing Environment Configuration

8.1 Testing via TravisCI

`bok-choy` can be used along with Travis CI to test changes remotely. One way to accomplish this testing is to use the headless version of Chrome or Firefox. bok-choy does this when the `BOKCHOY_HEADLESS` environment is set to "true".

```
before_script:
  - export BOKCHOY_HEADLESS=true
```

Another option is to use the X Virtual Framebuffer (xvfb) to imitate a display. Headless versions of Chrome and Firefox are relatively new developments, so you may want to use xvfb if you encounter a bug with headless browser usage. To use xvfb, you'll start it up via a `before_script` section in your `.travis.yml` file, like this:

```
before_script:
  - "export DISPLAY=:99.0"
  - "sh -e /etc/init.d/xvfb start"
  - sleep 3 # give xvfb some time to start
```

For more details, see this code example and the Travis docs.

8.2 Testing via tox

`bok-choy` can be used along with tox to test against multiple Python virtual environments containing different versions of requirements.

An important detail when using tox in a Travis CI environment: tox passes along only a fixed list of environment variables to each tox-created virtual environment. When using `bok-choy` via xvfb in tox, the DISPLAY environment variable is needed but is not automatically passed-in. The tox.ini file needs to specify the DISPLAY variable like this:

```
[testenv]
passenv =
    DISPLAY
```

For more details, see the tox docs.

CHAPTER 9

API Reference

9.1 browser

Use environment variables to configure Selenium remote WebDriver. For use with SauceLabs (via SauceConnect) or local browsers.

exception bok_choy.browser.**BrowserConfigError**
Misconfiguration error in the environment variables.

bok_choy.browser.**add_profile_customizer**(*func*)
Add a new function that modifies the preferences of the firefox profile object it receives as an argument

bok_choy.browser.**browser**(*tags=None*, *proxy=None*, *other_caps=None*)
Interpret environment variables to configure Selenium. Performs validation, logging, and sensible defaults.

There are three cases:

1. **Local browsers: If the proper environment variables are not all set for the second case,** then we use a local browser.
   - The environment variable *SELENIUM_BROWSER* can be set to specify which local browser to use. The default is Firefox.
   - Additionally, if a proxy instance is passed and the browser choice is either Chrome or Firefox, then the browser will be initialized with the proxy server set.
   - The environment variable *SELENIUM_FIREFOX_PATH* can be used for specifying a path to the Firefox binary. Default behavior is to use the system location.
   - The environment variable *FIREFOX_PROFILE_PATH* can be used for specifying a path to the Firefox profile. Default behavior is to use a barebones default profile with a few useful preferences set.
2. **Remote browser (not SauceLabs): Set all of the following environment variables, but not all of** the ones needed for SauceLabs:
   - SELENIUM_BROWSER

- SELENIUM_HOST
- SELENIUM_PORT

3. SauceLabs: Set all of the following environment variables:
   - SELENIUM_BROWSER
   - SELENIUM_VERSION
   - SELENIUM_PLATFORM
   - SELENIUM_HOST
   - SELENIUM_PORT
   - SAUCE_USER_NAME
   - SAUCE_API_KEY

NOTE: these are the environment variables set by the SauceLabs Jenkins plugin.

Optionally provide Jenkins info, used to identify jobs to Sauce:

- JOB_NAME
- BUILD_NUMBER

tags is a list of string tags to apply to the SauceLabs job. If not using SauceLabs, these will be ignored.

Keyword Arguments

- **tags** (*list of str*) – Tags to apply to the SauceLabs job. If not using SauceLabs, these will be ignored.
- **proxy** – A proxy instance.
- **other_caps** (*dict of str*) – Additional desired capabilities to provide to remote WebDriver instances. Note
- **these values will be overwritten by environment variables described above. This is only used for** (*that*) –
- **driver instances, where such info is usually used by services for additional configuration and** (*remote*) –
- **metadata.** –

Returns The configured browser object used to drive tests

Return type selenium.webdriver

Raises *BrowserConfigError* – The environment variables are not correctly specified.

bok_choy.browser.**clear_profile_customizers**()
Remove any previously-configured functions for customizing the firefox profile

bok_choy.browser.**save_driver_logs**(*driver*, *prefix*)
Save the selenium driver logs.

The location of the driver log files can be configured by the environment variable *SELENIUM_DRIVER_LOG_DIR*. If not set, this defaults to the current working directory.

Parameters

- **driver** (*selenium.webdriver*) – The Selenium-controlled browser.
- **prefix** (*str*) – A prefix which will be used in the output file names for the logs.

Returns None

bok_choy.browser.**save_screenshot**(*driver*, *name*)
Save a screenshot of the browser.

The location of the screenshot can be configured by the environment variable *SCREENSHOT_DIR*. If not set, this defaults to the current working directory.

Parameters

- **driver** (*selenium.webdriver*) – The Selenium-controlled browser.
- **name** (*str*) – A name for the screenshot, which will be used in the output file name.

Returns None

bok_choy.browser.**save_source**(*driver*, *name*)
Save the rendered HTML of the browser.

The location of the source can be configured by the environment variable *SAVED_SOURCE_DIR*. If not set, this defaults to the current working directory.

Parameters

- **driver** (*selenium.webdriver*) – The Selenium-controlled browser.
- **name** (*str*) – A name to use in the output file name. Note that ".html" is appended automatically

Returns None

9.2 javascript

Helpers for dealing with JavaScript synchronization issues.

bok_choy.javascript.**js_defined**(**js_vars*)
Class decorator that ensures JavaScript variables are defined in the browser.

This adds a *wait_for_js* method to the class, which will block until all the expected JavaScript variables are defined.

Parameters **js_vars** (*list of str*) – List of JavaScript variable names to wait for.

Returns Decorated class

bok_choy.javascript.**requirejs**(**modules*)
Class decorator that ensures RequireJS modules are loaded in the browser.

This adds a *wait_for_js* method to the class, which will block until all the expected RequireJS modules are loaded.

Parameters **modules** (*list of str*) –

Returns Decorated class

bok_choy.javascript.**wait_for_js**(*function*)
Method decorator that waits for JavaScript dependencies before executing *function*. If the function is not a method, the decorator has no effect.

Parameters **function** (*callable*) – Method to decorate.

Returns Decorated method

9.3 page_object

Base implementation of the Page Object pattern. See https://github.com/SeleniumHQ/selenium/wiki/PageObjects and http://www.seleniumhq.org/docs/06_test_design_considerations.jsp#page-object-design-pattern

exception bok_choy.page_object.**PageLoadError**
An error occurred while loading the page.

class bok_choy.page_object.**PageObject**(*browser*, **args*, ***kwargs*)
Encapsulates user interactions with a specific part of a web application.

The most important thing is this: Page objects encapsulate Selenium.

If you find yourself writing CSS selectors in tests, manipulating forms, or otherwise interacting directly with the web UI, stop!

Instead, put these in a *PageObject* subclass :)

PageObjects do their best to verify that they are only used when the browser is on a page containing the object. To do this, they will call *is_browser_on_page()* before executing any of their methods, and raise a *WrongPageError* if the browser isn't on the correct page.

Generally, this is the right behavior. However, at times it will be useful to not verify the page before executing a method. In those cases, the method can be marked with the *unguarded()* decorator. Additionally, private methods (those beginning with _) are always unguarded.

Class or instance properties are never guarded. However, methods marked with the `property()` are candidates for being guarded. To make them unguarded, you must mark the getter, setter, and deleter as *unguarded()* separately, and those decorators must be applied before the `property()` decorator.

Correct:

```
@property
@unguarded
def foo(self):
    return self._foo
```

Incorrect:

```
@unguarded
@property
def foo(self):
    return self._foo
```

Initialize the page object to use the specified browser instance.

Parameters **browser** (*selenium.webdriver*) – The Selenium-controlled browser.

Returns PageObject

a11y_audit
Initializes the a11y_audit attribute.

handle_alert(**args*, ***kwargs*)
Context manager that ensures alerts are dismissed.

Example usage:

```
with self.handle_alert():
    self.q(css='input.submit-button').first.click()
```

Keyword Arguments **confirm** (*bool*) – Whether to confirm or cancel the alert.

Returns None

is_browser_on_page()
Check that we are on the right page in the browser. The specific check will vary from page to page, but usually this amounts to checking the:

1. browser URL
2. page title
3. page headings

Returns A *bool* indicating whether the browser is on the correct page.

q(***kwargs*)
Construct a query on the browser.

Example usages:

```
self.q(css="div.foo").first.click()
self.q(xpath="/foo/bar").text
```

Keyword Arguments

- **css** – A CSS selector.
- **xpath** – An XPath selector.

Returns BrowserQuery

scroll_to_element(*element_selector*, *timeout=60*)
Scrolls the browser such that the element specified appears at the top. Before scrolling, waits for the element to be present.

Example usage:

```
self.scroll_to_element('.far-down', 'Scroll to far-down')
```

Parameters

- **element_selector** (*str*) – css selector of the element.
- **timeout** (*float*) – Maximum number of seconds to wait for the element to be present on the page before timing out.

Raises: BrokenPromise if the element does not exist (and therefore scrolling to it is not possible)

url
Return the URL of the page. This may be dynamic, determined by configuration options passed to the page object's constructor.

Some pages may not be directly accessible: perhaps the page object represents a "navigation" component that occurs on multiple pages. If this is the case, subclasses can return *None* to indicate that you can't directly visit the page object.

classmethod validate_url(*url*)
Return a boolean indicating whether the URL has a protocol and hostname. If a port is specified, ensure it is an integer.

Parameters **url** (*str*) – The URL to check.

Returns Boolean indicating whether the URL has a protocol and hostname.

visit()

Open the page containing this page object in the browser.

Some page objects may not provide a URL, in which case a *NotImplementedError* will be raised.

Raises

- *PageLoadError* – The page did not load successfully.
- NotImplementedError – The page object does not provide a URL to visit.

Returns PageObject

wait_for(*promise_check_func*, *description*, *result=False*, *timeout=60*)

Calls the method provided as an argument until the Promise satisfied or BrokenPromise. Retries if a WebDriverException is encountered (until the timeout is reached).

Parameters

- **promise_check_func** (*callable*) –
  - **If *result* is False Then** Function that accepts no arguments and returns a boolean indicating whether the promise is fulfilled
  - **If *result* is True Then** Function that accepts no arguments and returns a *(is_satisfied, result)* tuple, where *is_satisfied* is a boolean indicating whether the promise was satisfied, and *result* is a value to return from the fulfilled *Promise*
- **description** (*str*) – Description of the Promise, used in log messages
- **result** (*bool*) – Indicates whether we need result
- **timeout** (*float*) – Maximum number of seconds to wait for the Promise to be satisfied before timing out

Raises BrokenPromise – the *Promise* was not satisfied

wait_for_ajax(*timeout=30*)

Wait for jQuery to be loaded and for all ajax requests to finish. Note that we have to wait for jQuery to load first because it is used to check that ajax requests are complete.

Important: If you have an ajax requests that results in a page reload, you will need to use wait_for_page or some other method to confirm that the page has finished reloading after wait_for_ajax has returned.

Example usage:

```
self.q(css='input#email').fill("foo")
self.wait_for_ajax()
```

Keyword Arguments

- **timeout** (*int*) – The number of seconds to wait before timing out with
- **BrokenPromise exception.** (*a*) –

Returns None

Raises

- BrokenPromise – The timeout is exceeded before (1) jQuery is defined
- and (2) all ajax requests are completed.

wait_for_element_absence (*element_selector*, *description*, *timeout=60*)
Waits for element specified by *element_selector* until it disappears from DOM.

Example usage:

```
self.wait_for_element_absence('.submit', 'Submit Button is not Present')
```

Parameters

- **element_selector** (*str*) – css selector of the element.
- **description** (*str*) – Description of the Promise, used in log messages.
- **timeout** (*float*) – Maximum number of seconds to wait for the Promise to be satisfied before timing out

wait_for_element_invisibility (*element_selector*, *description*, *timeout=60*)
Waits for element specified by *element_selector* until it disappears from the web page.

Example usage:

```
self.wait_for_element_invisibility('.submit', 'Submit Button Disappeared')
```

Parameters

- **element_selector** (*str*) – css selector of the element.
- **description** (*str*) – Description of the Promise, used in log messages.
- **timeout** (*float*) – Maximum number of seconds to wait for the Promise to be satisfied before timing out

wait_for_element_presence (*element_selector*, *description*, *timeout=60*)
Waits for element specified by *element_selector* to be present in DOM.

Example usage:

```
self.wait_for_element_presence('.submit', 'Submit Button is Present')
```

Parameters

- **element_selector** (*str*) – css selector of the element.
- **description** (*str*) – Description of the Promise, used in log messages.
- **timeout** (*float*) – Maximum number of seconds to wait for the Promise to be satisfied before timing out

wait_for_element_visibility (*element_selector*, *description*, *timeout=60*)
Waits for element specified by *element_selector* until it is displayed on web page.

Example usage:

```
self.wait_for_element_visibility('.submit', 'Submit Button is Visible')
```

Parameters

- **element_selector** (*str*) – css selector of the element.
- **description** (*str*) – Description of the Promise, used in log messages.

- **timeout** (*float*) – Maximum number of seconds to wait for the Promise to be satisfied before timing out

wait_for_page(*timeout=30*)
Block until the page loads, then returns the page. Useful for ensuring that we navigate successfully to a particular page.

Keyword Arguments **timeout** (*int*) – The number of seconds to wait for the page before timing out with an exception.

Raises BrokenPromise – The timeout is exceeded without the page loading successfully.

warning(*msg*)
Subclasses call this to indicate that something unexpected occurred while interacting with the page.

Page objects themselves should never make assertions or raise exceptions, but they can issue warnings to make tests easier to debug.

Parameters **msg** (*str*) – The message to log as a warning.

Returns None

exception bok_choy.page_object.**WrongPageError**
The page object reports that we're on the wrong page!

exception bok_choy.page_object.**XSSExposureError**
An XSS issue has been found on the current page.

bok_choy.page_object.**no_selenium_errors**(*func*)
Decorator to create an *EmptyPromise* check function that is satisfied only when *func* executes without a Selenium error.

This protects against many common test failures due to timing issues. For example, accessing an element after it has been modified by JavaScript ordinarily results in a *StaleElementException*. Methods decorated with *no_selenium_errors* will simply retry if that happens, which makes tests more robust.

Parameters **func** (*callable*) – The function to execute, with retries if an error occurs.

Returns Decorated function

bok_choy.page_object.**pre_verify**(*method*)
Decorator that calls self._verify_page() before executing the decorated method

Parameters **method** (*callable*) – The method to decorate.

Returns Decorated method

bok_choy.page_object.**unguarded**(*method*)
Mark a PageObject method as unguarded.

Unguarded methods don't verify that the PageObject is on the current browser page before they execute

Parameters **method** (*callable*) – The method to decorate.

Returns Decorated method

9.4 accessibility

9.4.1 A11yAudit and A11yAuditConfig (Abstract Classes)

Interface for running accessibility audits on a PageObject.

class bok_choy.a11y.a11y_audit.**A11yAudit**(*browser*, *url*, *config=None*, **args*, ***kwargs*)
Allows auditing of a page for accessibility issues.

The ruleset to use can be specified by the environment variable *BOKCHOY_A11Y_RULESET*. Currently, there are two ruleset implemented:

axe_core:

- Ruleset class: AxeCoreAudit
- Ruleset config: AxeCoreAuditConfig
- This is default ruleset.

google_axs:

- Ruleset class: AxsAudit
- Ruleset config: AxsAuditConfig

Sets ruleset to be used.

Parameters

- **browser** – A browser instance
- **url** – URL of the page to test
- **config** – (optional) A11yAuditConfig or subclass of A11yAuditConfig

check_for_accessibility_errors()
Run an accessibility audit, parse the results, and raise a single exception if there are violations.

Note that an exception is only raised on errors, not on warnings.

Returns None

Raises *AccessibilityError*

default_config
Return an instance of a subclass of A11yAuditConfig.

do_audit()
Audit the page for accessibility problems using the enabled ruleset.

Returns A list (one for each browser session) of results returned from the audit. See documentation of *_check_rules* in the enabled ruleset for the format of each result.

static report_errors(*audit*, *url*)

Parameters

- **audit** – results of an accessibility audit.
- **url** – the url of the page being audited.

Raises

- *AccessibilityError* if errors are found in the audit.
- *'NotImplementedError' if this isn't overwritten in the ruleset* – specific implementation.

class bok_choy.a11y.a11y_audit.**A11yAuditConfig**(**args*, ***kwargs*)
The *A11yAuditConfig* object defines the options available in an accessibility ruleset.

customize_ruleset(*custom_ruleset_file=None*)
Allows customization of the ruleset. (e.g. adding custom rules, extending the implementation of an existing rule.)

Raises *'NotImplementedError' if this isn't overwritten in the ruleset* – specific implementation.

set_rules(*rules*)
Overrides the default rules to be run.

Raises *'NotImplementedError' if this isn't overwritten in the ruleset* – specific implementation.

set_rules_file(*path=None*)
Sets *self.rules_file* to the passed file.

Parameters **filepath where the JavaScript for the ruleset can be found.** (*A*) –

This is intended to be used in the case of using an extended or modified version of the ruleset. The interface and response format are expected to be unmodified.

set_scope(*include=None*, *exclude=None*)
Overrides the default scope (part of the DOM) to inspect.

Raises *'NotImplementedError' if this isn't overwritten in the ruleset* – specific implementation.

exception bok_choy.a11y.a11y_audit.**A11yAuditConfigError**
An error in A11yAuditConfig.

exception bok_choy.a11y.a11y_audit.**AccessibilityError**
The page violates one or more accessibility rules.

9.4.2 AxsAudit and AxsAuditConfig

Interface for using the google accessibility ruleset. See: https://github.com/GoogleChrome/accessibility-developer-tools

class bok_choy.a11y.axs_ruleset.**AuditResults**(*errors*, *warnings*)
Create new instance of AuditResults(errors, warnings)

errors
Alias for field number 0

warnings
Alias for field number 1

class bok_choy.a11y.axs_ruleset.**AxsAudit**(*browser*, *url*, *config=None*, **args*, ***kwargs*)
Use Google's Accessibility Developer Tools to audit a page for accessibility problems.

See https://github.com/GoogleChrome/accessibility-developer-tools

Sets ruleset to be used.

Parameters

- **browser** – A browser instance
- **url** – URL of the page to test
- **config** – (optional) A11yAuditConfig or subclass of A11yAuditConfig

default_config
Returns an instance of AxsAuditConfig.

static get_errors(*audit_results*)

Parameters **audit_results** – results of *AxsAudit.do_audit()*.

Returns: a list of errors.

static report_errors(*audit*, *url*)

Parameters

- **audit** – results of *AxsAudit.do_audit()*.
- **url** – the url of the page being audited.

Raises: *AccessibilityError*

class bok_choy.a11y.axs_ruleset.**AxsAuditConfig**(**args*, ***kwargs*)
The *AxsAuditConfig* object defines the options available when running an *AxsAudit*.

customize_ruleset(*custom_ruleset_file=None*)
This has not been implemented for the google_axs ruleset.

Raises *NotImplementedError*

set_rules(*rules*)
Sets the rules to be run or ignored for the audit.

Parameters **rules** – a dictionary of the format *{"ignore": [], "apply": []}*.

See https://github.com/GoogleChrome/accessibility-developer-tools/tree/master/src/audits

Passing *{"apply": []}* or *{}* means to check for all available rules.

Passing *{"apply": None}* means that no audit should be done for this page.

Passing *{"ignore": []}* means to run all otherwise enabled rules. Any rules in the "ignore" list will be ignored even if they were also specified in the "apply".

Examples

To check only *badAriaAttributeValue*:

```
page.a11y_audit.config.set_rules({
    "apply": ['badAriaAttributeValue']
})
```

To check all rules except *badAriaAttributeValue*:

```
page.a11y_audit.config.set_rules({
    "ignore": ['badAriaAttributeValue'],
})
```

set_scope(*include=None*, *exclude=None*)
Sets *scope*, the "start point" for the audit.

Parameters

- **include** – A list of css selectors specifying the elements that contain the portion of the page that should be audited. Defaults to auditing the entire document.
- **exclude** – This arg is not implemented in this ruleset.

Examples

To check only the *div* with id *foo*:

```
page.a11y_audit.config.set_scope(["div#foo"])
```

To reset the scope to check the whole document:

```
page.a11y_audit.config.set_scope()
```

9.4.3 AxeCoreAudit and AxeCoreAuditConfig

Interface for using the axe-core ruleset. See: https://github.com/dequelabs/axe-core

class bok_choy.a11y.axe_core_ruleset.**AxeCoreAudit**(*browser*, *url*, *config=None*, **args*, ***kwargs*)

Use Deque Labs' axe-core engine to audit a page for accessibility issues.

Related documentation:

https://github.com/dequelabs/axe-core/blob/master/doc/API.md

Sets ruleset to be used.

Parameters

- **browser** – A browser instance
- **url** – URL of the page to test
- **config** – (optional) A11yAuditConfig or subclass of A11yAuditConfig

default_config

Returns an instance of AxeCoreAuditConfig.

static format_errors(*errors*)

Parameters **errors** – results of *AxeCoreAudit.get_errors()*.

Returns: The errors as a formatted string.

static get_errors(*audit_results*)

Parameters **audit_results** – results of *AxeCoreAudit.do_audit()*.

Returns A dictionary with keys "errors" and "total".

static report_errors(*audit*, *url*)

Parameters

- **audit** – results of *AxeCoreAudit.do_audit()*.
- **url** – the url of the page being audited.

Raises: *AccessibilityError*

class bok_choy.a11y.axe_core_ruleset.**AxeCoreAuditConfig**(**args*, ***kwargs*)

The *AxeCoreAuditConfig* object defines the options available when running an *AxeCoreAudit*.

customize_ruleset(*custom_ruleset_file=None*)

Updates the ruleset to include a set of custom rules. These rules will be _added_ to the existing ruleset or replace the existing rule with the same ID.

Parameters **custom_ruleset_file** (*optional*) – The filepath to the custom rules. Defaults to *None*. If *custom_ruleset_file* isn't passed, the environment variable *BOKCHOY_A11Y_CUSTOM_RULES_FILE* will be checked. If a filepath isn't specified by either of these methods, the ruleset will not be updated.

Raises *IOError* if the specified file does not exist.

Examples

To include the rules defined in *axe-core-custom-rules.js*:

```
page.a11y_audit.config.customize_ruleset(
    "axe-core-custom-rules.js"
)
```

Alternatively, use the environment variable *BOKCHOY_A11Y_CUSTOM_RULES_FILE* to specify the path to the file containing the custom rules.

Documentation for how to write rules:

https://github.com/dequelabs/axe-core/blob/master/doc/developer-guide.md

An example of a custom rules file can be found at https://github.com/edx/bok-choy/tree/master/tests/a11y_custom_rules.js

set_rules(*rules*)

Set rules to ignore XOR limit to when checking for accessibility errors on the page.

Parameters **rules** – a dictionary one of the following formats. If you want to run all of the rules except for some:

```
{"ignore": []}
```

If you want to run only a specific set of rules:

```
{"apply": []}
```

If you want to run only rules of a specific standard:

```
{"tags": []}
```

Examples

To run only "bad-link" and "color-contrast" rules:

```
page.a11y_audit.config.set_rules({
    "apply": ["bad-link", "color-contrast"],
})
```

To run all rules except for "bad-link" and "color-contrast":

```
page.a11y_audit.config.set_rules({
    "ignore": ["bad-link", "color-contrast"],
})
```

To run only WCAG 2.0 Level A rules:

```
page.a11y_audit.config.set_rules({
    "tags": ["wcag2a"],
})
```

To run all rules: page.a11y_audit.config.set_rules({})

Related documentation:

- https://github.com/dequelabs/axe-core/blob/master/doc/API.md#options-parameter-examples
- https://github.com/dequelabs/axe-core/doc/rule-descriptions.md

set_scope(*include=None*, *exclude=None*)

Sets *scope* (refered to as *context* in ruleset documentation), which defines the elements on a page to include or exclude in the audit. If neither *include* nor *exclude* are passed, the entire document will be included.

Parameters

- **include** (*optional*) – a list of css selectors for elements that
- **be included in the audit. By, default, the entire document** (*should*) –
- **included.** (*is*) –
- **exclude** (*optional*) – a list of css selectors for elements that should not
- **included in the audit.** (*be*) –

Examples

To include all items in *#main-content* except *#some-special-elm*:

```
page.a11y_audit.config.set_scope(
    exclude=["#some-special-elm"],
    include=["#main-content"]
)
```

To include all items in the document except *#some-special-elm*:

```
page.a11y_audit.config.set_scope(
    exclude=["#some-special-elm"],
)
```

To include only children of *#some-special-elm*:

```
page.a11y_audit.config.set_scope(
    include=["#some-special-elm"],
)
```

Context documentation:

https://github.com/dequelabs/axe-core/blob/master/doc/API.md#a-context-parameter

Note that this implementation only supports css selectors. It does not accept nodes as described in the above documentation resource.

9.5 promise

Variation on the "promise" design pattern. Promises make it easier to handle asynchronous operations correctly.

exception bok_choy.promise.**BrokenPromise**(*promise*)
The promise was not satisfied within the time constraints.

Configure the broken promise error.

Parameters **promise** (Promise) – The promise that was not satisfied.

class bok_choy.promise.**EmptyPromise**(*check_func*, *description*, ***kwargs*)
A promise that has no result value.

Configure the promise.

Unlike a regular *Promise*, the *check_func()* does NOT return a tuple with a result value. That's why the promise is "empty" – you don't get anything back.

Example usage:

```
# This will block until `is_done` returns `True` or we reach the timeout limit.
EmptyPromise(lambda: is_done('test'), "Test operation is done").fulfill()
```

Parameters

- **check_func** (*callable*) – Function that accepts no arguments and returns a boolean indicating whether the promise is fulfilled.
- **description** (*str*) – Description of the Promise, used in log messages.

Returns EmptyPromise

class bok_choy.promise.**Promise**(*check_func*, *description*, *try_limit=None*, *try_interval=0.5*, *timeout=30*)
Check that an asynchronous action completed, blocking until it does or timeout / try limits are reached.

Configure the *Promise*.

The *Promise* will poll *check_func()* until either:

- The promise is satisfied
- The promise runs out of tries (checks more than *try_limit* times)
- The promise runs out of time (takes longer than *timeout* seconds)

If the try_limit or timeout is reached without success, then the promise is "broken" and an exception will be raised.

Note that if you specify a try_limit but not a timeout, the default timeout is still used. This is to prevent an inadvertent infinite loop. If you want to make sure that the try_limit expires first (and thus that many attempts will be made), then you should also pass in a larger value for timeout.

description is a string that will be included in the exception to make debugging easier.

Example:

```
# Dummy check function that indicates the promise is always satisfied
check_func = lambda: (True, "Hello world!")

# Check up to 5 times if the operation has completed
result = Promise(check_func, "Operation has completed", try_limit=5).fulfill()
```

Parameters

- **check_func** (*callable*) – A function that accepts no arguments and returns a *(is_satisfied, result)* tuple, where *is_satisfied* is a boolean indiating whether the promise was satisfied, and *result* is a value to return from the fulfilled *Promise*.
- **description** (*str*) – Description of the *Promise*, used in log messages.

Keyword Arguments

- **try_limit** (*int or None*) – Number of attempts to make to satisfy the *Promise*. Can be *None* to disable the limit.
- **try_interval** (*float*) – Number of seconds to wait between attempts.
- **timeout** (*float*) – Maximum number of seconds to wait for the *Promise* to be satisfied before timing out.

Returns Promise

fulfill()

Evaluate the promise and return the result.

Returns The result of the *Promise* (second return value from the *check_func*)

Raises *BrokenPromise* – the *Promise* was not satisfied within the time or attempt limits.

9.6 query

Tools for interacting with the DOM inside a browser.

class bok_choy.query.**BrowserQuery**(*browser*, ***kwargs*)

A Query that operates on a browser.

Generate a query over a browser.

Parameters **browser** (*selenium.webdriver*) – A Selenium-controlled browser.

Keyword Arguments

- **css** (*str*) – A CSS selector.
- **xpath** (*str*) – An XPath selector.

Returns BrowserQuery

Raises TypeError – The query must be passed either a CSS or XPath selector, but not both.

attrs(*attribute_name*)

Retrieve HTML attribute values from the elements matched by the query.

Example usage:

```
# Assume that the query matches html elements:
# <div class="foo"> and <div class="bar">
>> q.attrs('class')
['foo', 'bar']
```

Parameters **attribute_name** (*str*) – The name of the attribute values to retrieve.

Returns A list of attribute values for *attribute_name*.

click()
Click each matched element.

Example usage:

```
# Click the first element matched by the query
q.first.click()
```

Returns None

fill(*text*)
Set the text value of each matched element to *text*.

Example usage:

```
# Set the text of the first element matched by the query to "Foo"
q.first.fill('Foo')
```

Parameters **text** (*str*) – The text used to fill the element (usually a text field or text area).

Returns None

focused
Checks that *at least one* matched element is focused. More specifically, it checks whether the element is document.activeElement. If no matching element is focused, this returns *False*.

Returns bool

html
Retrieve the inner HTML of each element matched by the query.

Example usage:

```
# Assume that the query matches html elements:
# <div><span>Foo</span></div> and <div>Bar</div>
>> q.html
['<span>Foo</span>', 'Bar']
```

Returns The inner HTML for each element matched by the query.

invisible
Check whether all matched elements are present, but not visible.

Returns bool

is_focused()
Checks that *at least one* matched element is focused. More specifically, it checks whether the element is document.activeElement. If no matching element is focused, this returns *False*.

Returns bool

selected
Check whether all the matched elements are selected.

Returns bool

text
Retrieve text from each matched element.

Example usage:

```
# Assume that the query matches html elements:
# <div>Foo</div> and <div>Bar</div>
>> q.text
['Foo', 'Bar']
```

Returns The text of each element matched by the query.

visible
Check whether all matched elements are visible.

Returns bool

class bok_choy.query.**Query**(*seed_fn*, *desc=None*)
General mechanism for selecting and transforming values.

Configure the *Query*.

Parameters **seed_fn** (*callable*) – Callable with no arguments that produces a list of values.

Keyword Arguments **desc** (*str*) – A description of the query, used in log messages. If not provided, defaults to the name of the seed function.

Returns Query

execute(*try_limit=5*, *try_interval=0.5*, *timeout=30*)
Execute this query, retrying based on the supplied parameters.

Keyword Arguments

- **try_limit** (*int*) – The number of times to retry the query.
- **try_interval** (*float*) – The number of seconds to wait between each try (float).
- **timeout** (*float*) – The maximum number of seconds to spend retrying (float).

Returns The transformed results of the query.

Raises BrokenPromise – The query did not execute without a Selenium error after one or more attempts.

filter(*filter_fn=None*, *desc=None*, ***kwargs*)
Return a copy of this query, with some values removed.

Example usages:

```
# Returns a query that matches even numbers
q.filter(filter_fn=lambda x: x % 2)

# Returns a query that matches elements with el.description == "foo"
q.filter(description="foo")
```

Keyword Arguments

- **filter_fn** (*callable*) – If specified, a function that accepts one argument (the element) and returns a boolean indicating whether to include that element in the results.
- **kwargs** – Specify attribute values that an element must have to be included in the results.
- **desc** (*str*) – A description of the filter, for use in log messages. Defaults to the name of the filter function or attribute.

Raises `TypeError` – neither or both of *filter_fn* and *kwargs* are provided.

first
Return a Query that selects only the first element of this Query. If no elements are available, returns a query with no results.

Example usage:

```
>> q = Query(lambda: list(range(5)))
>> q.first.results
[0]
```

Returns Query

is_present()
Check whether the query returns any results.

Returns Boolean indicating whether the query contains any results.

map(*map_fn*, *desc=None*)
Return a copy of this query, with the values mapped through *map_fn*.

Parameters **map_fn** (*callable*) – A callable that takes a single argument and returns a new value.

Keyword Arguments **desc** (*str*) – A description of the mapping transform, for use in log message. Defaults to the name of the map function.

Returns Query

nth(*index*)
Return a query that selects the element at *index* (starts from 0). If no elements are available, returns a query with no results.

Example usage:

```
>> q = Query(lambda: list(range(5)))
>> q.nth(2).results
[2]
```

Parameters **index** (*int*) – The index of the element to select (starts from 0)

Returns Query

present
Check whether the query returns any results.

Returns Boolean indicating whether the query contains any results.

replace(***kwargs*)
Return a copy of this *Query*, but with attributes specified as keyword arguments replaced by the keyword values.

Keyword Arguments **to replace in the copy.** (*Attributes/values*) –

Returns A copy of the query that has its attributes updated with the specified values.

Raises `TypeError` – The *Query* does not have the specified attribute.

results
A list of the results of the query, which are cached. If you call *results* multiple times on the same query, you will always get the same results. Use *reset()* to clear the cache and re-run the query.

Returns The results from executing the query.

transform(*transform*, *desc=None*)
Create a copy of this query, transformed by *transform*.

Parameters **transform** (*callable*) – Callable that takes an iterable of values and returns an iterable of transformed values.

Keyword Arguments **desc** (*str*) – A description of the transform, to use in log messages. Defaults to the name of the *transform* function.

Returns Query

bok_choy.query.**no_error**(*func*)
Decorator to create a *Promise* check function that is satisfied only when *func* executes without a Selenium error.

This protects against many common test failures due to timing issues. For example, accessing an element after it has been modified by JavaScript ordinarily results in a *StaleElementException*. Methods decorated with *no_error* will simply retry if that happens, which makes tests more robust.

Parameters **func** (*callable*) – The function to execute, with retries if an error occurs.

Returns Decorated function

9.7 web_app_test

Base class for testing a web application.

class bok_choy.web_app_test.**WebAppTest**(**args*, ***kwargs*)
Base class for testing a web application.

get_web_driver()
Override NeedleTestCases's get_web_driver class method to return the WebDriver instance that is already being used, instead of starting up a new one.

quit_browser()
Terminate the web browser which was launched to run the tests.

setUp()
Start the browser for use by the test. You *must* call this in the *setUp* method of any subclasses before using the browser!

Returns None

classmethod setUpClass()
Override NeedleTestCase's setUpClass method so that it does not start up the browser once for each testcase class. Instead we start up the browser once per TestCase instance, in the setUp method.

set_viewport_size(*width*, *height*)
Override NeedleTestCases's set_viewport_size class method because we need it to operate on the instance not the class.

See the Needle documentation at http://needle.readthedocs.org/ for information on this feature. It is particularly useful to predict the size of the resulting screenshots when taking fullscreen captures, or to test responsive sites.

classmethod tearDownClass()
Override NeedleTestCase's tearDownClass method because it would quit the browser. This is not needed as we have already quit the browser after each TestCase, by virtue of a cleanup that we add in the setUp method.

unique_id
Helper method to return a uuid.

Returns 39-char UUID string

CHAPTER 10

Indices and tables

- genindex
- modindex
- search

Python Module Index

b

Index

X