Marionette Python Client Documentation

Release

Mozilla Automation and Tools team

July 17, 2015

Contents

The Marionette python client library allows you to remotely control a Gecko-based browser or device which is running a Marionette server. This includes desktop Firefox and FirefoxOS (support for Firefox for Android is planned, but not yet fully implemented).

The Marionette server is built directly into Gecko and can be started by passing in a command line option to Gecko, or by using a Marionette-enabled build. The server listens for connections from various clients. Clients can then control Gecko by sending commands to the server.

This is the official python client for Marionette. There also exists a NodeJS client maintained by the Firefox OS automation team.

CHAPTER 1

Getting the Client

The python client is officially supported. To install it, first make sure you have pip installed then run:

```
pip install marionette_client
```

It's highly recommended to use virtualenv when installing Marionette to avoid package conflicts and other general nastiness.

You should now be ready to start using Marionette. The best way to learn is to play around with it. Start a Marionette-enabled instance of Firefox, fire up a python shell and follow along with the interactive tutorial!

CHAPTER 2

Using the Client for Testing

Please visit the Marionette Tests section on MDN for information regarding testing with Marionette.

CHAPTER 3

Session Management

A session is a single instance of a Marionette client connected to a Marionette server. Before you can start executing commands, you need to start a session with `start_session()`:

```
client = Marionette('localhost', port=2828)
client.start_session()
```

This returns a session id and an object listing the capabilities of the Marionette server. For example, a server running on a Firefox OS device will have the ability to rotate the window, while a server running from Firefox won't. It's also possible to access the capabilities using the `session_capabilities` attribute. After finishing with a session, you can delete it with `delete_session()`. Note that this will also happen automatically when the Marionette object is garbage collected.

CHAPTER 4

Context Management

Commands can only be executed in a single window, frame and scope at a time. In order to run commands elsewhere, it's necessary to explicitly switch to the appropriate context.

Use `switch_to_window()` to execute commands in the context of a new window:

```
original_window = client.current_window_handle
for handle in client.window_handles:
    if handle != original_window:
        client.switch_to_window(handle)
        print("Switched to window with '{}' loaded.".format(client.get_url()))
client.switch_to_window(original_window)
```

Similarly, use `switch_to_frame()` to execute commands in the context of a new frame (e.g an <iframe> element):

```
iframe = client.find_element(By.TAG_NAME, 'iframe')
client.switch_to_frame(iframe)
assert iframe == client.get_active_frame()
```

Finally Marionette can switch between *chrome* and *content* scope. Chrome is a privileged scope where you can access things like the Firefox UI itself or the system app in Firefox OS. Content scope is where things like webpages or normal Firefox OS apps live. You can switch between *chrome* and *content* using the `set_context()` and `using_context()` functions:

```
client.set_context(client.CONTEXT_CONTENT)
# content scope
with client.using_context(client.CONTEXT_CHROME):
    #chrome scope
    ... do stuff ...
# content scope restored
```

CHAPTER 5

Navigation

Use `navigate()` to open a new website. It's also possible to move through the back/forward cache using `go_forward()` and `go_back()` respectively. To retrieve the currently open website, use `get_url()`:

```
url = 'http://mozilla.org'
client.navigate(url)
client.go_back()
client.go_forward()
assert client.get_url() == url
```

CHAPTER 6

DOM Elements

In order to inspect or manipulate actual DOM elements, they must first be found using the `find_element()` or `find_elements()` methods:

```
from marionette import HTMLElement
element = client.find_element(By.ID, 'my-id')
assert type(element) == HTMLElement
elements = client.find_elements(By.TAG_NAME, 'a')
assert type(elements) == list
```

For a full list of valid search strategies, see Finding Elements.

Now that an element has been found, it's possible to manipulate it:

```
element.click()
element.send_keys('hello!')
print(element.get_attribute('style'))
```

For the full list of possible commands, see the `HTMLElement` reference.

Be warned that a reference to an element object can become stale if it was modified or removed from the document. See Dealing with Stale Elements for tips on working around this limitation.

CHAPTER 7

Script Execution

Sometimes Marionette's provided APIs just aren't enough and it is necessary to run arbitrary javascript. This is accomplished with the `execute_script()` and `execute_async_script()` functions. They accomplish what their names suggest, the former executes some synchronous JavaScript, while the latter provides a callback mechanism for running asynchronous JavaScript:

```
result = client.execute_script("return arguments[0] + arguments[1];",
                               script_args=[2, 3])
assert result == 5
```

The async method works the same way, except it won't return until a special *marionetteScriptFinished()* function is called:

```
result = client.execute_async_script("""
    setTimeout(function() {
      marionetteScriptFinished("all done");
    }, arguments[0]);
""", script_args=[1000])
assert result == "all done"
```

Beware that running asynchronous scripts can potentially hang the program indefinitely if they are not written properly. It is generally a good idea to set a script timeout using `set_script_timeout()` and handling *ScriptTimeoutException*.

CHAPTER 8

Indices and tables

- genindex
- modindex
- search

8.1 Marionette Python Client

The Marionette python client library allows you to remotely control a Gecko-based browser or device which is running a Marionette server. This includes desktop Firefox and FirefoxOS (support for Firefox for Android is planned, but not yet fully implemented).

The Marionette server is built directly into Gecko and can be started by passing in a command line option to Gecko, or by using a Marionette-enabled build. The server listens for connections from various clients. Clients can then control Gecko by sending commands to the server.

This is the official python client for Marionette. There also exists a NodeJS client maintained by the Firefox OS automation team.

8.1.1 Getting the Client

The python client is officially supported. To install it, first make sure you have pip installed then run:

```
pip install marionette_client
```

It's highly recommended to use virtualenv when installing Marionette to avoid package conflicts and other general nastiness.

You should now be ready to start using Marionette. The best way to learn is to play around with it. Start a Marionette-enabled instance of Firefox, fire up a python shell and follow along with the interactive tutorial!

8.1.2 Using the Client for Testing

Please visit the Marionette Tests section on MDN for information regarding testing with Marionette.

8.1.3 Session Management

A session is a single instance of a Marionette client connected to a Marionette server. Before you can start executing commands, you need to start a session with `start_session()`:

```
client = Marionette('localhost', port=2828)
client.start_session()
```

This returns a session id and an object listing the capabilities of the Marionette server. For example, a server running on a Firefox OS device will have the ability to rotate the window, while a server running from Firefox won't. It's also possible to access the capabilities using the `session_capabilities` attribute. After finishing with a session, you can delete it with `delete_session()`. Note that this will also happen automatically when the Marionette object is garbage collected.

8.1.4 Context Management

Commands can only be executed in a single window, frame and scope at a time. In order to run commands elsewhere, it's necessary to explicitly switch to the appropriate context.

Use `switch_to_window()` to execute commands in the context of a new window:

```
original_window = client.current_window_handle
for handle in client.window_handles:
    if handle != original_window:
        client.switch_to_window(handle)
        print("Switched to window with '{}' loaded.".format(client.get_url()))
client.switch_to_window(original_window)
```

Similarly, use `switch_to_frame()` to execute commands in the context of a new frame (e.g an <iframe> element):

```
iframe = client.find_element(By.TAG_NAME, 'iframe')
client.switch_to_frame(iframe)
assert iframe == client.get_active_frame()
```

Finally Marionette can switch between *chrome* and *content* scope. Chrome is a privileged scope where you can access things like the Firefox UI itself or the system app in Firefox OS. Content scope is where things like webpages or normal Firefox OS apps live. You can switch between *chrome* and *content* using the `set_context()` and `using_context()` functions:

```
client.set_context(client.CONTEXT_CONTENT)
# content scope
with client.using_context(client.CONTEXT_CHROME):
    #chrome scope
    ... do stuff ...
# content scope restored
```

8.1.5 Navigation

Use `navigate()` to open a new website. It's also possible to move through the back/forward cache using `go_forward()` and `go_back()` respectively. To retrieve the currently open website, use `get_url()`:

```
url = 'http://mozilla.org'
client.navigate(url)
client.go_back()
client.go_forward()
assert client.get_url() == url
```

8.1.6 DOM Elements

In order to inspect or manipulate actual DOM elements, they must first be found using the `find_element()` or `find_elements()` methods:

```
from marionette import HTMLElement
element = client.find_element(By.ID, 'my-id')
assert type(element) == HTMLElement
elements = client.find_elements(By.TAG_NAME, 'a')
assert type(elements) == list
```

For a full list of valid search strategies, see Finding Elements.

Now that an element has been found, it's possible to manipulate it:

```
element.click()
element.send_keys('hello!')
print(element.get_attribute('style'))
```

For the full list of possible commands, see the `HTMLElement` reference.

Be warned that a reference to an element object can become stale if it was modified or removed from the document. See Dealing with Stale Elements for tips on working around this limitation.

8.1.7 Script Execution

Sometimes Marionette's provided APIs just aren't enough and it is necessary to run arbitrary javascript. This is accomplished with the `execute_script()` and `execute_async_script()` functions. They accomplish what their names suggest, the former executes some synchronous JavaScript, while the latter provides a callback mechanism for running asynchronous JavaScript:

```
result = client.execute_script("return arguments[0] + arguments[1];",
                               script_args=[2, 3])
assert result == 5
```

The async method works the same way, except it won't return until a special *marionetteScriptFinished()* function is called:

```
result = client.execute_async_script("""
    setTimeout(function() {
      marionetteScriptFinished("all done");
    }, arguments[0]);
""", script_args=[1000])
assert result == "all done"
```

Beware that running asynchronous scripts can potentially hang the program indefinitely if they are not written properly. It is generally a good idea to set a script timeout using `set_script_timeout()` and handling *ScriptTimeoutException*.

8.2 Using the Client Interactively

Once you installed the client and have Marionette running, you can fire up your favourite interactive python environment and start playing with Marionette. Let's use a typical python shell:

```
python
```

First, import Marionette:

```
from marionette import Marionette
```

Now create the client for this session. Assuming you're using the default port on a Marionette instance running locally:

```
client = Marionette(host='localhost', port=2828)
client.start_session()
```

This will return some id representing your session id. Now that you've established a connection, let's start doing interesting things:

```
client.execute_script("alert('o hai there!');")
```

You should now see this alert pop up! How exciting! Okay, let's do something practical. Close the dialog and try this:

```
client.navigate("http://www.mozilla.org")
```

Now you're at mozilla.org! You can even verify it using the following:

```
client.get_url()
```

You can even find an element and click on it. Let's say you want to get the first link:

```
from marionette import By
first_link = client.find_element(By.TAG_NAME, "a")
```

first_link now holds a reference to the first link on the page. You can click it:

```
first_link.click()
```

8.3 Advanced Topics

Here are a collection of articles explaining some of the more complicated aspects of Marionette.

8.3.1 Finding Elements

One of the most common and yet often most difficult tasks in Marionette is finding a DOM element on a webpage or in the chrome UI. Marionette provides several different search strategies to use when finding elements. All search strategies work with both `find_element()` and `find_elements()`, though some strategies are not implemented in chrome scope.

In the event that more than one element is matched by the query, `find_element()` will only return the first element found. In the event that no elements are matched by the query, `find_element()` will raise *NoSuchElementException* while `find_elements()` will return an empty list.

Search Strategies

Search strategies are defined in the `By` class:

```
from marionette import By
print(By.ID)
```

The strategies are:

- *id* - The easiest way to find an element is to refer to its id directly:

```
container = client.find_element(By.ID, 'container')
```

- *class name* - To find elements belonging to a certain class, use *class name*:

```
buttons = client.find_elements(By.CLASS_NAME, 'button')
```

- *css selector* - It's also possible to find elements using a css selector:

```
container_buttons = client.find_elements(By.CSS_SELECTOR, '#container .buttons')
```

- *name* - Find elements by their name attribute (not implemented in chrome scope):

```
form = client.find_element(By.NAME, 'signup')
```

- *tag name* - To find all the elements with a given tag, use *tag name*:

```
paragraphs = client.find_elements(By.TAG_NAME, 'p')
```

- *link text* - A convenience strategy for finding link elements by their innerHTML (not implemented in chrome scope):

```
link = client.find_element(By.LINK_TEXT, 'Click me!')
```

- *partial link text* - Same as *link text* except substrings of the innerHTML are matched (not implemented in chrome scope):

```
link = client.find_element(By.PARTIAL_LINK_TEXT, 'Clic')
```

- *xpath* - Find elements using an xpath query:

```
elem = client.find_element(By.XPATH, './/*[@id="foobar"]')
```

Chaining Searches

In addition to the methods on the Marionette object, HTMLElement objects also provide `find_element()` and `find_elements()` methods. The difference is that only child nodes of the element will be searched. Consider the following html snippet:

```
<div id="content">
    <span id="main"></span>
</div>
<div id="footer"></div>
```

Doing the following will work:

```
client.find_element(By.ID, 'container').find_element(By.ID, 'main')
```

But this will raise a *NoSuchElementException*:

```
client.find_element(By.ID, 'container').find_element(By.ID, 'footer')
```

Finding Anonymous Nodes

When working in chrome scope, for example manipulating the Firefox user interface, you may run into something called an anonymous node.

Firefox uses a markup language called XUL for its interface. XUL is similar to HTML in that it has a DOM and tags that render controls on the display. One ability of XUL is to create re-useable widgets that are made up out of several

smaller XUL elements. These widgets can be bound to the DOM using something called the XML binding language (XBL).

The end result is that the DOM sees the widget as a single entity. It doesn't know anything about how that widget is made up. All of the smaller XUL elements that make up the widget are called anonymous content. It is not possible to query such elements using traditional DOM methods like *getElementById*.

Marionette provides two special strategies used for finding anonymous content. Unlike normal elements, anonymous nodes can only be seen by their parent. So it's necessary to first find the parent element and then search for the anonymous children from there.

- *anon* - Finds all anonymous children of the element, there is no search term so *None* must be passed in:

```
anon_children = client.find_element('id', 'parent').find_elements('anon', None)
```

- *anon attribute* - Find an anonymous child based on an attribute. An unofficial convention is for anonymous nodes to have an *anonid* attribute:

```
anon_child = client.find_element('id', 'parent').find_element('anon attribute', {'anonid': 'cont
```

8.3.2 Dealing with Stale Elements

Marionette does not keep a live representation of the DOM saved. All it can do is send commands to the Marionette server which queries the DOM on the client's behalf. References to elements are also not passed from server to client. A unique id is generated for each element that gets referenced and a mapping of id to element object is stored on the server. When commands such as `click()` are run, the client sends the element's id along with the command. The server looks up the proper DOM element in its reference table and executes the command on it.

In practice this means that the DOM can change state and Marionette will never know until it sends another query. For example, look at the following HTML:

```
<head>
<script type=text/javascript>
    function addDiv() {
       var div = document.createElement("div");
       document.getElementById("container").appendChild(div);
    }
</script>
</head>

<body>
    <div id="container">
    </div>
    <input id="button" type=button onclick="addDiv();">
</body>
```

Care needs to be taken as the DOM is being modified after the page has loaded. The following code has a race condition:

```
button = client.find_element('id', 'button')
button.click()
assert len(client.find_elements('css selector', '#container div')) > 0
```

Explicit Waiting and Expected Conditions

To avoid the above scenario, manual synchronisation is needed. Waits are used to pause program execution until a given condition is true. This is a useful technique to employ when documents load new content or change after

`Document.readyState`'s value changes to "complete".

The `Wait` helper class provided by Marionette avoids some of the caveats of `time.sleep(n)`. It will return immediately once the provided condition evaluates to true.

To avoid the race condition in the above example, one could do:

```
button = client.find_element('id', 'button')
button.click()

def find_divs():
    return client.find_elements('css selector', '#container div')

divs = Wait(client).until(find_divs)
assert len(divs) > 0
```

This avoids the race condition. Because finding elements is a common condition to wait for, it is built in to Marionette. Instead of the above, you could write:

```
button = client.find_element('id', 'button')
button.click()
assert len(Wait(client).until(expected.elements_present('css selector', '#container div'))) > 0
```

For a full list of built-in conditions, see `expected`.

8.3.3 Actions

Action Sequences

`Actions` are designed as a way to simulate user input as closely as possible on a touch device like a smart phone. A common operation is to tap the screen and drag your finger to another part of the screen and lift it off.

This can be simulated using an Action:

```
from marionette import Actions

start_element = marionette.find_element('id', 'start')
end_element = marionette.find_element('id', 'end')

action = Actions(marionette)
action.press(start_element).wait(1).move(end_element).release()
action.perform()
```

This will simulate pressing an element, waiting for one second, moving the finger over to another element and then lifting the finger off the screen. The wait is optional in this case, but can be useful for simulating delays typical to a users behaviour.

Multi-Action Sequences

Sometimes it may be necessary to simulate multiple actions at the same time. For example a user may be dragging one finger while tapping another. This is where `MultiActions` come in. MultiActions are simply a way of combining two or more actions together and performing them all at the same time:

```
action1 = Actions(marionette)
action1.press(start_element).move(end_element).release()

action2 = Actions(marionette)
```

```
action2.press(another_element).wait(1).release()

multi = MultiActions(marionette)
multi.add(action1)
multi.add(action2)
multi.perform()
```

8.3.4 Debugging

Sometimes when working with Marionette you'll run into unexpected behaviour and need to do some debugging. This page outlines some of the Marionette methods that can be useful to you.

Please note that the best tools for debugging are the ones that ship with Gecko. This page doesn't describe how to use those with Marionette. Also see a related topic about using the debugger with Marionette on MDN.

Storing Logs on the Server

By calling *~Marionette.log* it is possible to store a message on the server. Logs can later be retrieved using *~Marionette.get_logs*. For example:

```
try:
    marionette.log("Sending a click event") # logged at INFO level
    elem.click()
except:
    marionette.log("Something went wrong!", "ERROR")

print(marionette.get_logs())
```

Disclaimer: Example for illustrative purposes only, don't actually hide tracebacks like that!

Seeing What's on the Page

Sometimes it's difficult to tell what is actually on the page that is being manipulated. Either because it happens too fast, the window isn't big enough or you are manipulating a remote server! There are two methods that can help you out. The first is *~Marionette.screenshot*:

```
marionette.screenshot() # takes screenshot of entire frame
elem = marionette.find_element(By.ID, 'some-div')
marionette.screenshot(elem) # takes a screenshot of only the given element
```

Sometimes you just want to see the DOM layout. You can do this with the *~Marionette.page_source* property. Note that the page source depends on the context you are in:

```
print(marionette.page_source)
marionette.set_context('chrome')
print(marionette.page_source)
```

8.4 API Reference

8.4.1 Marionette

class marionette_driver.marionette.**Marionette**(*host='localhost'*, *port=2828*, *app=None*, *app_args=None*, *bin=None*, *profile=None*, *emulator=None*, *sdcard=None*, *emulator_img=None*, *emulator_binary=None*, *emulator_res=None*, *connect_to_running_emulator=False*, *gecko_log=None*, *homedir=None*, *baseurl=None*, *no_window=False*, *logdir=None*, *busybox=None*, *symbols_path=None*, *timeout=None*, *socket_timeout=360*, *device_serial=None*, *adb_path=None*, *process_args=None*, *adb_host=None*, *adb_port=None*, *prefs=None*)

Represents a Marionette connection to a browser or device.

absolute_url(*relative_url*)

Returns an absolute url for files served from Marionette's www directory.

Parameters **relative_url** – The url of a static file, relative to Marionette's www directory.

add_cookie(*cookie*)

Adds a cookie to your current session.

Parameters **cookie** – A dictionary object, with required keys - "name" and "value"; optional keys - "path", "domain", "secure", "expiry".

Usage example:

```
driver.add_cookie({'name': 'foo', 'value': 'bar'})
driver.add_cookie({'name': 'foo', 'value': 'bar', 'path': '/'})
driver.add_cookie({'name': 'foo', 'value': 'bar', 'path': '/',
                   'secure': True})
```

chrome_window_handles

Get a list of currently open chrome windows.

Each window handle is assigned by the server, and the list of strings returned does not have a guaranteed ordering.

Returns unordered list of unique window handles as strings

clear_imported_scripts()

Clears all imported scripts in this context, ie: calling clear_imported_scripts in chrome context will clear only scripts you imported in chrome, and will leave the scripts you imported in content context.

close()

Close the current window, ending the session if it's the last window currently open.

On B2G this method is a noop and will return immediately.

close_chrome_window()

Close the currently selected chrome window, ending the session if it's the last window open.

On B2G this method is a noop and will return immediately.

current_chrome_window_handle
Get the current chrome window's handle. Corresponds to a chrome window that may itself contain tabs identified by window_handles.

Returns an opaque server-assigned identifier to this window that uniquely identifies it within this Marionette instance. This can be used to switch to this window at a later point.

Returns unique window handle

Return type string

current_window_handle
Get the current window's handle.

Returns an opaque server-assigned identifier to this window that uniquely identifies it within this Marionette instance. This can be used to switch to this window at a later point.

Returns unique window handle

Return type string

delete_all_cookies()
Delete all cookies in the scope of the current session.

Usage example:

```
driver.delete_all_cookies()
```

delete_cookie(*name*)
Delete a cookie by its name.

Parameters **name** – Name of cookie to delete.

Usage example:

```
driver.delete_cookie('foo')
```

delete_session()
Close the current session and disconnect from the server.

enforce_gecko_prefs(*prefs*)
Checks if the running instance has the given prefs. If not, it will kill the currently running instance, and spawn a new instance with the requested preferences.

: param prefs: A dictionary whose keys are preference names.

execute_async_script(*script*, *script_args=None*, *new_sandbox=True*, *sandbox='default'*, *script_timeout=None*, *debug_script=False*)
Executes an asynchronous JavaScript script, and returns the result (or None if the script does return a value).

The script is executed in the context set by the most recent set_context() call, or to the CONTEXT_CONTENT context if set_context() has not been called.

Parameters

- **script** – A string containing the JavaScript to execute.
- **script_args** – A list of arguments to pass to the script.
- **sandbox** – A tag referring to the sandbox you wish to use; if you specify a new tag, a new sandbox will be created. If you use the special tag 'system', the sandbox will be created using the system principal which has elevated privileges.

- **new_sandbox** – If False, preserve global variables from the last execute_*script call. This is True by default, in which case no globals are preserved.
- **debug_script** – Capture javascript exceptions when in CONTEXT_CHROME context.

Usage example:

```
marionette.set_script_timeout(10000) # set timeout period of 10 seconds
result = self.marionette.execute_async_script("""
  // this script waits 5 seconds, and then returns the number 1
  setTimeout(function() {
    marionetteScriptFinished(1);
  }, 5000);
""")
assert result == 1
```

execute_script(*script*, *script_args=None*, *new_sandbox=True*, *sandbox='default'*, *script_timeout=None*)

Executes a synchronous JavaScript script, and returns the result (or None if the script does return a value).

The script is executed in the context set by the most recent set_context() call, or to the CONTEXT_CONTENT context if set_context() has not been called.

Parameters

- **script** – A string containing the JavaScript to execute.
- **script_args** – A list of arguments to pass to the script.
- **sandbox** – A tag referring to the sandbox you wish to use; if you specify a new tag, a new sandbox will be created. If you use the special tag 'system', the sandbox will be created using the system principal which has elevated privileges.
- **new_sandbox** – If False, preserve global variables from the last execute_*script call. This is True by default, in which case no globals are preserved.

Simple usage example:

```
result = marionette.execute_script("return 1;")
assert result == 1
```

You can use the *script_args* parameter to pass arguments to the script:

```
result = marionette.execute_script("return arguments[0] + arguments[1];",
                                   script_args=[2, 3])
assert result == 5
some_element = marionette.find_element("id", "someElement")
sid = marionette.execute_script("return arguments[0].id;", script_args=[some_element])
assert some_element.get_attribute("id") == sid
```

Scripts wishing to access non-standard properties of the window object must use window.wrappedJSObject:

```
result = marionette.execute_script("""
  window.wrappedJSObject.test1 = 'foo';
  window.wrappedJSObject.test2 = 'bar';
  return window.wrappedJSObject.test1 + window.wrappedJSObject.test2;
  """)
assert result == "foobar"
```

Global variables set by individual scripts do not persist between script calls by default. If you wish to persist data between script calls, you can set new_sandbox to False on your next call, and add any new variables to a new 'global' object like this:

```
marionette.execute_script("global.test1 = 'foo';")
result = self.marionette.execute_script("return global.test1;", new_sandbox=False)
assert result == 'foo'
```

find_element(*method*, *target*, *id=None*)

Returns an HTMLElement instances that matches the specified method and target in the current context.

An HTMLElement instance may be used to call other methods on the element, such as click(). If no element is immediately found, the attempt to locate an element will be repeated for up to the amount of time set by set_search_timeout(). If multiple elements match the given criteria, only the first is returned. If no element matches, a NoSuchElementException will be raised.

Parameters

- **method** – The method to use to locate the element; one of: "id", "name", "class name", "tag name", "css selector", "link text", "partial link text", "xpath", "anon" and "anon attribute". Note that the "name", "link text" and "partial link test" methods are not supported in the chrome dom.
- **target** – The target of the search. For example, if method = "tag", target might equal "div". If method = "id", target would be an element id.
- **id** – If specified, search for elements only inside the element with the specified id.

find_elements(*method*, *target*, *id=None*)

Returns a list of all HTMLElement instances that match the specified method and target in the current context.

An HTMLElement instance may be used to call other methods on the element, such as click(). If no element is immediately found, the attempt to locate an element will be repeated for up to the amount of time set by set_search_timeout().

Parameters

- **method** – The method to use to locate the elements; one of: "id", "name", "class name", "tag name", "css selector", "link text", "partial link text", "xpath", "anon" and "anon attribute". Note that the "name", "link text" and "partial link test" methods are not supported in the chrome dom.
- **target** – The target of the search. For example, if method = "tag", target might equal "div". If method = "id", target would be an element id.
- **id** – If specified, search for elements only inside the element with the specified id.

get_active_frame()

Returns an HTMLElement representing the frame Marionette is currently acting on.

get_cookie(*name*)

Get a single cookie by name. Returns the cookie if found, None if not.

Parameters **name** – Name of cookie to get.

get_cookies()

Get all the cookies for the current domain.

This is the equivalent of calling *document.cookie* and parsing the result.

Returns A set of cookies for the current domain.

get_logs()
Returns the list of logged messages.

Each log message is an array with three string elements: the level, the message, and a date.

Usage example:

```
marionette.log("I AM INFO")
marionette.log("I AM ERROR", "ERROR")
logs = marionette.get_logs()
assert logs[0][1] == "I AM INFO"
assert logs[1][1] == "I AM ERROR"
```

get_url()
Get a string representing the current URL.

On Desktop this returns a string representation of the URL of the current top level browsing context. This is equivalent to document.location.href.

When in the context of the chrome, this returns the canonical URL of the current resource.

Returns string representation of URL

get_window_position()
Get the current window's position Return a dictionary with the keys x and y :returns: a dictionary with x and y

get_window_type()
Gets the windowtype attribute of the window Marionette is currently acting on.

This command only makes sense in a chrome context. You might use this method to distinguish a browser window from an editor window.

go_back()
Causes the browser to perform a back navigation.

go_forward()
Causes the browser to perform a forward navigation.

import_script(*js_file*)
Imports a script into the scope of the execute_script and execute_async_script calls.

This is particularly useful if you wish to import your own libraries.

Parameters **js_file** – Filename of JavaScript file to import.

For example, Say you have a script, importfunc.js, that contains:

```
let testFunc = function() { return "i'm a test function!";};
```

Assuming this file is in the same directory as the test, you could do something like:

```
js = os.path.abspath(os.path.join(__file__, os.path.pardir, "importfunc.js"))
marionette.import_script(js)
assert "i'm a test function!" == self.marionette.execute_script("return testFunc();")
```

log(*msg*, *level=None*)
Stores a timestamped log message in the Marionette server for later retrieval.

Parameters

- **msg** – String with message to log.
- **level** – String with log level (e.g. "INFO" or "DEBUG"). If None, defaults to "INFO".

maximize_window()
Resize the browser window currently receiving commands. The action should be equivalent to the user pressing the the maximize button

navigate(*url*)
Navigate to given *url*.

Navigates the current top-level browsing context's content frame to the given URL and waits for the document to load or the session's page timeout duration to elapse before returning.

The command will return with a failure if there is an error loading the document or the URL is blocked. This can occur if it fails to reach the host, the URL is malformed, the page is restricted (about:* pages), or if there is a certificate issue to name some examples.

The document is considered successfully loaded when the *DOMContentLoaded* event on the frame element associated with the *window* triggers and *document.readState* is "complete".

In chrome context it will change the current *window*'s location to the supplied URL and wait until *document.readState* equals "complete" or the page timeout duration has elapsed.

Parameters **url** – The URL to navigate to.

orientation
Get the current browser orientation.

Will return one of the valid primary orientation values portrait-primary, landscape-primary, portrait-secondary, or landscape-secondary.

page_source
A string representation of the DOM.

refresh()
Causes the browser to perform to refresh the current page.

restart(*clean=False*, *in_app=False*)
This will terminate the currently running instance, and spawn a new instance with the same profile and then reuse the session id when creating a session again.

: param clean: If False the same profile will be used after the restart. Note that the in app initiated restart always maintains the same profile.

: param in_app: If True, marionette will cause a restart from within the browser. Otherwise the browser will be restarted immediately by killing the process.

screenshot(*element=None*, *highlights=None*, *format='base64'*, *full=True*)
Takes a screenshot of a web element or the current frame.

The screen capture is returned as a lossless PNG image encoded as a base 64 string by default. If the *element* argument is defined the capture area will be limited to the bounding box of that element. Otherwise, the capture area will be the bounding box of the current frame.

Parameters

- **element** – The element to take a screenshot of. If None, will take a screenshot of the current frame.
- **highlights** – A list of HTMLElement objects to draw a red box around in the returned screenshot.
- **format** – if "base64" (the default), returns the screenshot as a base64-string. If "binary", the data is decoded and returned as raw binary.
- **full** – If True (the default), the capture area will be the complete frame. Else only the viewport is captured. Only applies when *element* is None.

session_capabilities
A JSON dictionary representing the capabilities of the current session.

set_context(*context*)
Sets the context that Marionette commands are running in.

Parameters **context** – Context, may be one of the class properties *CONTEXT_CHROME* or *CONTEXT_CONTENT*.

Usage example:

```
marionette.set_context(marionette.CONTEXT_CHROME)
```

set_orientation(*orientation*)
Set the current browser orientation.

The supplied orientation should be given as one of the valid orientation values. If the orientation is unknown, an error will be raised.

Valid orientations are "portrait" and "landscape", which fall back to "portrait-primary" and "landscape-primary" respectively, and "portrait-secondary" as well as "landscape-secondary".

Parameters **orientation** – The orientation to lock the screen in.

set_script_timeout(*timeout*)
Sets the maximum number of ms that an asynchronous script is allowed to run.

If a script does not return in the specified amount of time, a ScriptTimeoutException is raised.

Parameters **timeout** – The maximum number of milliseconds an asynchronous script can run without causing an ScriptTimeoutException to be raised

set_search_timeout(*timeout*)
Sets a timeout for the find methods.

When searching for an element using either `Marionette.find_element` or `Marionette.find_elements`, the method will continue trying to locate the element for up to timeout ms. This can be useful if, for example, the element you're looking for might not exist immediately, because it belongs to a page which is currently being loaded.

Parameters **timeout** – Timeout in milliseconds.

set_window_position(*x*, *y*)

Set the position of the current window

param x x coordinate for the top left of the window

param y y coordinate for the top left of the window

set_window_size(*width*, *height*)
Resize the browser window currently in focus.

The supplied width and height values refer to the window outerWidth and outerHeight values, which include scroll bars, title bars, etc.

An error will be returned if the requested window size would result in the window being in the maximised state.

Parameters

- **width** – The width to resize the window to.
- **height** – The height to resize the window to.

start_session(*desired_capabilities=None*, *session_id=None*, *timeout=60*)
Create a new Marionette session.

This method must be called before performing any other action.

Parameters

- **desired_capabilities** – An optional dict of desired capabilities. This is currently ignored.
- **timeout** – Timeout in seconds for the server to be ready.
- **session_id** – unique identifier for the session. If no session id is passed in then one will be generated by the marionette server.

Returns A dict of the capabilities offered.

switch_to_alert()
Returns an Alert object for interacting with a currently displayed alert.

```
alert = self.marionette.switch_to_alert()
text = alert.text
alert.accept()
```

switch_to_default_content()
Switch the current context to page's default content.

switch_to_frame(*frame=None*, *focus=True*)
Switch the current context to the specified frame. Subsequent commands will operate in the context of the specified frame, if applicable.

Parameters

- **frame** – A reference to the frame to switch to. This can be an `HTMLElement`, an integer index, string name, or an ID attribute. If you call `switch_to_frame` without an argument, it will switch to the top-level frame.
- **focus** – A boolean value which determins whether to focus the frame that we just switched to.

switch_to_window(*window_id*)
Switch to the specified window; subsequent commands will be directed at the new window.

Parameters **window_id** – The id or name of the window to switch to.

timeouts(*timeout_type*, *ms*)
An interface for managing timeout behaviour of a Marionette instance.

Setting timeouts specifies the type and amount of time the Marionette instance should wait during requests.

There are three types of timeouts that can be set: implicit, script and page load.

•An implicit timeout specifies the amount of time a Marionette instance should wait when searching for elements. Here, marionette polls a page until an element is found or the timeout expires, whichever occurs first. When searching for multiple elements, the driver should poll the page until at least one element is found or the timeout expires, at which point it should return an empty list.

•A script timeout specifies the amount of time the Marionette instance should wait after calling executeAsyncScript for the callback to have executed before returning a timeout response.

•A page load timeout specifies the amount of time the Marionette instance should wait for a page load operation to complete. If this limit is exceeded, the Marionette instance will return a "timeout" response status.

Parameters

- **timeout_type** – A string value specifying the timeout type. This must be one of three types: 'implicit', 'script' or 'page load'
- **ms** – A number value specifying the timeout length in milliseconds (ms)

title
Current title of the active window.

using_context(**args*, ***kwds*)
Sets the context that Marionette commands are running in using a *with* statement. The state of the context on the server is saved before entering the block, and restored upon exiting it.

Parameters **context** – Context, may be one of the class properties *CONTEXT_CHROME* or *CONTEXT_CONTENT*.

Usage example:

```
with marionette.using_context(marionette.CONTEXT_CHROME):
    # chrome scope
    ... do stuff ...
```

using_permissions(**args*, ***kwds*)
Sets permissions for code being executed in a *with* block, and restores them on exit.

Parameters **perms** – A dict containing one or more perms and their

values to be set.

Usage example:

```
with marionette.using_permissions({'systemXHR': True}):
    ... do stuff ...
```

window_handles
Get list of windows in the current context.

If called in the content context it will return a list of references to all available browser windows. Called in the chrome context, it will list all available windows, not just browser windows (e.g. not just navigator.browser).

Each window handle is assigned by the server, and the list of strings returned does not have a guaranteed ordering.

Returns unordered list of unique window handles as strings

window_size
Get the current browser window size.

Will return the current browser window size in pixels. Refers to window outerWidth and outerHeight values, which include scroll bars, title bars, etc.

Returns dictionary representation of current window width and height

8.4.2 HTMLElement

class marionette_driver.marionette.**HTMLElement**(*marionette*, *id*)
Represents a DOM Element.

clear()
Clears the input of the element.

find_element(*method*, *target*)
Returns an HTMLElement instance that matches the specified method and target, relative to the current element.

For more details on this function, see the find_element method in the Marionette class.

find_elements(*method*, *target*)
Returns a list of all HTMLElement instances that match the specified method and target in the current context.

For more details on this function, see the find_elements method in the Marionette class.

get_attribute(*attribute*)
Returns the requested attribute (or None, if no attribute is set).

Parameters **attribute** – The name of the attribute.

is_displayed()
Returns True if the element is displayed.

is_enabled()
This command will return False if all the following criteria are met otherwise return True:

- A form control is disabled.
- A HtmlElement has a disabled boolean attribute.

is_selected()
Returns True if the element is selected.

location
Get an element's location on the page.

The returned point will contain the x and y coordinates of the top left-hand corner of the given element. The point (0,0) refers to the upper-left corner of the document.

Returns a dictionary containing x and y as entries

rect
Gets the element's bounding rectangle.

This will return a dictionary with the following:

- x and y represent the top left coordinates of the `HTMLElement` relative to top left corner of the document.
- height and the width will contain the height and the width of the DOMRect of the `HTMLElement`.

send_keys(**string*)
Sends the string via synthesized keypresses to the element.

size
A dictionary with the size of the element.

tag_name
The tag name of the element.

tap(*x=None*, *y=None*)
Simulates a set of tap events on the element.

Parameters

- **x** – X-coordinate of tap event. If not given, default to the center of the element.
- **y** – Y-coordinate of tap event. If not given, default to the center of the element.

text
Returns the visible text of the element, and its child elements.

value_of_css_property(*property_name*)
Gets the value of the specified CSS property name.

Parameters **property_name** – Property name to get the value of.

8.4.3 DateTimeValue

class marionette_driver.**DateTimeValue**(*element*)
Interface for setting the value of HTML5 "date" and "time" input elements.

Simple usage example:

```
element = marionette.find_element("id", "date-test")
dt_value = DateTimeValue(element)
dt_value.date = datetime(1998, 6, 2)
```

date
Retrieve the element's string value

time
Retrieve the element's string value

8.4.4 Actions

class marionette_driver.marionette.**Actions**(*marionette*)
An Action object represents a set of actions that are executed in a particular order.

All action methods (press, etc.) return the Actions object itself, to make it easy to create a chain of events.

Example usage:

```
# get html file
testAction = marionette.absolute_url("testFool.html")
# navigate to the file
marionette.navigate(testAction)
# find element1 and element2
element1 = marionette.find_element("id", "element1")
element2 = marionette.find_element("id", "element2")
# create action object
action = Actions(marionette)
# add actions (press, wait, move, release) into the object
action.press(element1).wait(5). move(element2).release()
# fire all the added events
action.perform()
```

cancel()
Sends 'touchcancel' event to the target of the original 'touchstart' event.

May only be called if press() has already be called.

click(*element*, *button=0*, *count=1*)
Performs a click with additional parameters to allow for double clicking, right click, middle click, etc.

Parameters

- **element** – The element to click.

- **button** – The mouse button to click (indexed from 0, left to right).
- **count** – Optional, the count of clicks to synthesize (for double click events).

context_click(*element*)
Performs a context click on the specified element.

Parameters **element** – The element to context click.

double_click(*element*)
Performs a double click on the specified element.

Parameters **element** – The element to double click.

double_tap(*element*, *x=None*, *y=None*)
Performs a double tap on the target element.

Parameters

- **element** – The element to double tap.
- **x** – Optional, x-coordinate of double tap, relative to the top-left corner of the element.
- **y** – Optional, y-coordinate of double tap, relative to the top-left corner of the element.

flick(*element*, *x1*, *y1*, *x2*, *y2*, *duration=200*)
Performs a flick gesture on the target element.

Parameters

- **element** – The element to perform the flick gesture on.
- **x1** – Starting x-coordinate of flick, relative to the top left corner of the element.
- **y1** – Starting y-coordinate of flick, relative to the top left corner of the element.
- **x2** – Ending x-coordinate of flick, relative to the top left corner of the element.
- **y2** – Ending y-coordinate of flick, relative to the top left corner of the element.
- **duration** – Time needed for the flick gesture for complete (in milliseconds).

key_down(*key_code*)
Perform a "keyDown" action for the given key code. Modifier keys are respected by the server for the course of an action chain.

Parameters **key_code** – The key to press as a result of this action.

key_up(*key_code*)
Perform a "keyUp" action for the given key code. Modifier keys are respected by the server for the course of an action chain. :param key_up: The key to release as a result of this action.

long_press(*element*, *time_in_seconds*, *x=None*, *y=None*)
Performs a long press gesture on the target element.

Parameters

- **element** – The element to press.
- **time_in_seconds** – Time in seconds to wait before releasing the press.
- **x** – Optional, x-coordinate to tap, relative to the top-left corner of the element.
- **y** – Optional, y-coordinate to tap, relative to the top-left corner of the element.

This is equivalent to calling:

```
        action.press(element, x, y).wait(time_in_seconds).release()
```

middle_click(*element*)
Performs a middle click on the specified element.

Parameters **element** – The element to middle click.

move(*element*)
Sends a 'touchmove' event at the center of the target element.

Parameters **element** – Element to move towards.

May only be called if press() has already be called.

move_by_offset(*x*, *y*)
Sends 'touchmove' event to the given x, y coordinates relative to the top-left of the currently touched element.

May only be called if press() has already be called.

Parameters

- **x** – Specifies x-coordinate of move event, relative to the top-left corner of the element.
- **y** – Specifies y-coordinate of move event, relative to the top-left corner of the element.

perform()
Sends the action chain built so far to the server side for execution and clears the current chain of actions.

press(*element*, *x=None*, *y=None*)
Sends a 'touchstart' event to this element.

If no coordinates are given, it will be targeted at the center of the element. If given, it will be targeted at the (x,y) coordinates relative to the top-left corner of the element.

Parameters

- **element** – The element to press on.
- **x** – Optional, x-coordinate to tap, relative to the top-left corner of the element.
- **y** – Optional, y-coordinate to tap, relative to the top-left corner of the element.

release()
Sends a 'touchend' event to this element.

May only be called if press() has already be called on this element.

If press and release are chained without a move action between them, then it will be processed as a 'tap' event, and will dispatch the expected mouse events ('mousemove' (if necessary), 'mousedown', 'mouseup', 'mouseclick') after the touch events. If there is a wait period between press and release that will trigger a contextmenu, then the 'contextmenu' menu event will be fired instead of the touch/mouse events.

tap(*element*, *x=None*, *y=None*)
Performs a quick tap on the target element.

Parameters

- **element** – The element to tap.
- **x** – Optional, x-coordinate of tap, relative to the top-left corner of the element. If not specified, default to center of element.
- **y** – Optional, y-coordinate of tap, relative to the top-left corner of the element. If not specified, default to center of element.

This is equivalent to calling:

```
action.press(element, x, y).release()
```

wait(*time=None*)
Waits for specified time period.

Parameters **time** – Time in seconds to wait. If time is None then this has no effect for a single action chain. If used inside a multi-action chain, then time being None indicates that we should wait for all other currently executing actions that are part of the chain to complete.

8.4.5 MultiActions

class marionette_driver.marionette.**MultiActions**(*marionette*)
A MultiActions object represents a sequence of actions that may be performed at the same time. Its intent is to allow the simulation of multi-touch gestures. Usage example:

```
# create multiaction object
multitouch = MultiActions(marionette)
# create several action objects
action_1 = Actions(marionette)
action_2 = Actions(marionette)
# add actions to each action object/finger
action_1.press(element1).move_to(element2).release()
action_2.press(element3).wait().release(element3)
# fire all the added events
multitouch.add(action_1).add(action_2).perform()
```

add(*action*)
Adds a set of actions to perform.

Parameters **action** – An Actions object.

perform()
Perform all the actions added to this object.

8.4.6 Wait

class marionette_driver.**Wait**(*marionette*, *timeout=None*, *interval=0.1*, *ignored_exceptions=None*, *clock=None*)
An explicit conditional utility class for waiting until a condition evaluates to true or not null.

This will repeatedly evaluate a condition in anticipation for a truthy return value, or its timeout to expire, or its waiting predicate to become true.

A *Wait* instance defines the maximum amount of time to wait for a condition, as well as the frequency with which to check the condition. Furthermore, the user may configure the wait to ignore specific types of exceptions whilst waiting, such as *errors.NoSuchElementException* when searching for an element on the page.

__init__(*marionette*, *timeout=None*, *interval=0.1*, *ignored_exceptions=None*, *clock=None*)
Configure the Wait instance to have a custom timeout, interval, and list of ignored exceptions. Optionally a different time implementation than the one provided by the standard library (time) can also be provided.

Sample usage:

```
# Wait 30 seconds for window to open, checking for its presence once
# every 5 seconds.
wait = Wait(marionette, timeout=30, interval=5,
```

```
                    ignored_exceptions=errors.NoSuchWindowException)
        window = wait.until(lambda m: m.switch_to_window(42))
```

Parameters

- **marionette** – The input value to be provided to conditions, usually a Marionette instance.
- **timeout** – How long to wait for the evaluated condition to become true. The default timeout is the *timeout* property on the *Marionette* object if set, or *wait.DEFAULT_TIMEOUT*.
- **interval** – How often the condition should be evaluated. In reality the interval may be greater as the cost of evaluating the condition function is not factored in. The default polling interval is *wait.DEFAULT_INTERVAL*.
- **ignored_exceptions** – Ignore specific types of exceptions whilst waiting for the condition. Any exceptions not whitelisted will be allowed to propagate, terminating the wait.
- **clock** – Allows overriding the use of the runtime's default time library. See *wait.SystemClock* for implementation details.

__weakref__
list of weak references to the object (if defined)

until(*condition*, *is_true=None*, *message=''*)
Repeatedly runs condition until its return value evaluates to true, or its timeout expires or the predicate evaluates to true.

This will poll at the given interval until the given timeout is reached, or the predicate or conditions returns true. A condition that returns null or does not evaluate to true will fully elapse its timeout before raising an *errors.TimeoutException*.

If an exception is raised in the condition function and it's not ignored, this function will raise immediately. If the exception is ignored, it will continue polling for the condition until it returns successfully or a *TimeoutException* is raised.

Parameters

- **condition** – A callable function whose return value will be returned by this function if it evaluates to true.
- **is_true** – An optional predicate that will terminate and return when it evaluates to False. It should be a function that will be passed clock and an end time. The default predicate will terminate a wait when the clock elapses the timeout.
- **message** – An optional message to include in the exception's message if this function times out.

Built-in Conditions

class marionette_driver.expected.**element_displayed**(**args*)
An expectation for checking that an element is visible.

Visibility means that the element is not only displayed, but also has a height and width that is greater than 0 pixels.

Stale elements, meaning elements that have been detached from the DOM of the current context are treated as not being displayed, meaning this expectation is not analogous to the behaviour of calling *is_displayed()* on an *HTMLElement*.

You can select which element to be checked for visibility by supplying a locator:

```
displayed = Wait(marionette).until(expected.element_displayed(By.ID, "foo"))
```

Or by supplying an element:

```
el = marionette.find_element(By.ID, "foo")
displayed = Wait(marionette).until(expected.element_displayed(el))
```

Parameters **args** – locator or web element

Returns True if element is displayed, False if hidden

class marionette_driver.expected.**element_enabled**(*element*)
An expectation for checking that the given element is enabled.

Parameters **element** – the element to check if enabled

Returns True if element is enabled, False otherwise

class marionette_driver.expected.**element_not_displayed**(**args*)
An expectation for checking that an element is not visible.

Visibility means that the element is not only displayed, but also has a height and width that is greater than 0 pixels.

Stale elements, meaning elements that have been detached fom the DOM of the current context are treated as not being displayed, meaning this expectation is not analogous to the behaviour of calling *is_displayed()* on an *HTMLElement*.

You can select which element to be checked for visibility by supplying a locator:

```
hidden = Wait(marionette).until(expected.element_not_displayed(By.ID, "foo"))
```

Or by supplying an element:

```
el = marionette.find_element(By.ID, "foo")
hidden = Wait(marionette).until(expected.element_not_displayed(el))
```

Parameters **args** – locator or web element

Returns True if element is hidden, False if displayed

class marionette_driver.expected.**element_not_enabled**(*element*)
An expectation for checking that the given element is disabled.

Parameters **element** – the element to check if disabled

Returns True if element is disabled, False if enabled

class marionette_driver.expected.**element_not_present**(**args*)
Checks that a web element is not present in the DOM of the current context.

You can select which element to be checked for lack of presence by supplying a locator:

```
r = Wait(marionette).until(expected.element_not_present(By.ID, "foo"))
```

Or by using a function/lambda returning an element:

```
r = Wait(marionette).until(expected.element_present(lambda m: m.find_element(By.ID, "foo")))
```

Parameters **args** – locator or function returning web element

Returns True if element is not present, or False if it is present

class marionette_driver.expected.**element_not_selected**(*element*)
An expectation for checking that the given element is not selected.

Parameters **element** – the element to not be selected

Returns True if element is not selected, False if selected

class marionette_driver.expected.**element_present**(**args*)
Checks that a web element is present in the DOM of the current context. This does not necessarily mean that the element is visible.

You can select which element to be checked for presence by supplying a locator:

```
el = Wait(marionette).until(expected.element_present(By.ID, "foo"))
```

Or by using a function/lambda returning an element:

```
el = Wait(marionette).until(expected.element_present(lambda m: m.find_element(By.ID, "foo")))
```

Parameters **args** – locator or function returning web element

Returns the web element once it is located, or False

class marionette_driver.expected.**element_selected**(*element*)
An expectation for checking that the given element is selected.

Parameters **element** – the element to be selected

Returns True if element is selected, False otherwise

class marionette_driver.expected.**element_stale**(*element*)
Check that the given element is no longer attached to DOM of the current context.

This can be useful for waiting until an element is no longer present.

Sample usage:

```
el = marionette.find_element(By.ID, "foo")
# ...
Wait(marionette).until(expected.element_stale(el))
```

Parameters **element** – the element to wait for

Returns False if the element is still attached to the DOM, True otherwise

class marionette_driver.expected.**elements_not_present**(**args*)
Checks that web elements are not present in the DOM of the current context.

You can select which elements to be checked for not being present by supplying a locator:

```
r = Wait(marionette).until(expected.elements_not_present(By.TAG_NAME, "a"))
```

Or by using a function/lambda returning a list of elements:

```
r = Wait(marionette).until(expected.elements_not_present(lambda m: m.find_elements(By.TAG_NAME,
```

Parameters **args** – locator or function returning a list of web elements

Returns True if elements are missing, False if one or more are present

class marionette_driver.expected.**elements_present**(**args*)

Checks that web elements are present in the DOM of the current context. This does not necessarily mean that the elements are visible.

You can select which elements to be checked for presence by supplying a locator:

```
els = Wait(marionette).until(expected.elements_present(By.TAG_NAME, "a"))
```

Or by using a function/lambda returning a list of elements:

```
els = Wait(marionette).until(expected.elements_present(lambda m: m.find_elements(By.TAG_NAME, "a
```

Parameters **args** – locator or function returning a list of web elements

Returns list of web elements once they are located, or False

m

Symbols

A

C

D

E