Germanium

Web Testing API that doesn't disappoint

Germanium Static

The Germanium static package is for creating tests that revolve around running a single browser instance at a time, in the whole test process.

def open_browser(browser="Firefox", wd=None, iframe_selector=DefaultIFrameSelector(), screenshot_folder="screenshots", scripts=list())

Open the given browser:

```
open_browser("firefox")
```

or:

```
open_browser("ie")
```

This allows also connecting to remote drivers, for example:

```
open_browser("ff:http://10.2.1.1:5555/wd/hub")
```

def close_browser()

Close the currently running browser.

def go_to(url)

Go to the given URL, and wait for the page to load.

```
go_to("http://google.com/")
```

def type_keys(keys, selector = None)

Type the keys specified into the element, or the currently active element.

```
type_keys('germanium', Input('q'))
```

def click(selector)

Click the element with the given selector.

```
click(Button('OK'))
```

def hover(selector)

Hover the element with the given selector.

def double_click(selector)

Double click the element with the given selector.

def right_click(selector)

```
Right click the element with the given selector.

def get_web_driver()
--------------------

Return the WebDriver instance the global Germanium was built around.

def get_germanium()
-------------------

Returns the currently running Germanium instance.

def S(*argv, **kwargs)
```

Returns a deferred locator, using the `S`uper locator.

def iframe(target, keep_new_context = False)

Selects the current working iframe with the `target` name.

```
@iframe("editor")
def type_keys_into_editor(keys):
    type_keys(keys)

type_keys_into_editor('hello world') # will switch the iframe to 'editor' and back
click(Button("Save"))                # iframe is 'default'
```

Germanium API Documentation

There are three kinds of functions that are provided for easier support inside the browsers:

1. decorator:
    - `@iframe`
2. germanium functions:
    - `S`, super locator
    - `js`, `execute_script`
    - `take_screenshot`
    - `load_script`
3. static utility functions:
    - `type_keys`
    - `click`
    - `double_click`
    - `right_click`
    - `hover`
    - `get_attributes`
    - `wait`

germanium iframe decorator

@iframe(name, keep_new_context=False)

Switch the iframe when executing the code of the function. It will use the strategy provided when the Germanium instance was created.

For example if we would have an editor that is embedded in an IFrame, and we would want to call the saving of the document, we could implement that such as:

```
@iframe("default")
def close_dialog(germanium):
    germanium.S('"Save dialog" > button["Ok"]').click()

@iframe("editor")
def save_document(germanium):
    germanium.S('#save-button').element().click()
    close_dialog(germanium)
```

The `@iframe` decorator is going to find the current context by scanning the parameters of the function for the Germanium instance. If the first parameter is an object that contains a property named either: `germanium` or `_germanium` will be used.

germanium Instance Functions

Constructor GermaniumDriver(web_driver, ..)

Constructs a new GermaniumDriver utility object on top of whatever WebDriver object is given.

```
GermaniumDriver(web_driver,
                iframe_selector=DefaultIFrameSelector(),
                screenshot_folder="screenshots",
                scripts=list())
```

The only required parameter is the `web_driver` argument, that must be a WebDriver instance.

iframe_selector

The `iframe_selector` specifies the strategy to use whenever the iframe will be changed by the `@iframe` decorator. This class should have a method named `select_iframe(self, germanium, iframe_name)`.

Germanium uses `"default"` for the `switch_to_default_content`.

The default implementation is:

```
class DefaultIFrameSelector(object):
    """
    An implementation of the IFrameSelector strategy that does nothing.
    """
    def select_iframe(self, germanium, iframe_name):
        if iframe_name != "default":
            raise Exception("Unknown iframe name: '%s'. Make sure you create an IFrame
Selector "
                            "that you will pass when creating the GermaniumDriver,
e.g.:\n"
                            "GermaniumDriver(wd, iframe_selector=MyIFrameSelector())")

        germanium.switch_to_default_content()
        return iframe_name
```

This can easily be changed so depending on the `iframe_name` it will do a switch_to_frame on the germanium object.

```
class EditorIFrameSelector(object):
    def select_iframe(self, germanium, iframe_name):
        if iframe_name == "default":
            germanium.switch_to_default_content()
        elif iframe_name == "editor":
            editor_iframe = germanium.find_element_by_css_selector('iframe')
            germanium.switch_to_frame(editor_iframe)

        return iframe_name
```

In case you don't want a full class, you can pass also a callable that will be invoked with two parameters `germanium` and `iframe_name`:

```
def select_iframe(germanium, iframe_name):
    if iframe_name == "default":
        germanium.switch_to_default_content()
    elif iframe_name == "editor":
        editor_iframe = germanium.find_element_by_css_selector('iframe')
        germanium.switch_to_frame(editor_iframe)

    return iframe_name
```

So when invoking the `GermaniumDriver` someone can:

```
GermaniumDriver(web_driver,
                iframe_selector=select_iframe)
```

screenshot_folder

The folder where to save the screenshots, whenever take_screenshot is called. It defaults to `"screenshots"`, so basically a local folder named screenshots in the current working directory.

scripts

A list of files with JavaScript to be automatically loaded into the page, whenever either `get()`, `reload_page()` or `wait_for_page_to_load()` is done.

germanium.S(locator, strategy?)

`S` stands for the super locator, and returns an object that can execute a locator in the current iframe context of germanium. The letter `S` was chosen since it is looking greatly similar with jquery's `$`.

The first parameter, the locator, can be any of the selector objects from the germanium.selectors package, or a string that will be further interpreted on what selector will be used.

For example to find a button you can either:

```
germanium.S(Button('OK'))
```

or using a CSS selector:

```
germanium.S("input[value'OK'][type='button']")
```

or using a specific locator:

```
# implicit strategy detection, will match XPath, due to // start
germanium.S("//input[@value='OK'][@type='button']")
# or explicit in-string strategy:
germanium.S("xpath://input[@value='OK'][@type='button']")
# or explicit strategy:
germanium.S("//input[@value='OK'][@type='button']", "xpath")
```

The selectors approach is recommended since a selector find will match either an html `input` element of type `button`, either a html button `element` that has the label OK.

The S locator is not itself a locator but rather a locator strategy. Thus the S locator will choose:

1. if the searched expression starts with `//` then the xpath locator will be used.

```
# will find elements by XPath
germanium.S('//*[contains(@class, "test")]');
```

1. else the css locator will be used.

```
# will find elements by CSS
germanium.S('.test')
```

```
# will find elements by the simple locator
germanium.S('*[contains(@class, "test")')
```

The S function call will return an object that is compatible with the static `wait_for` command.

germanium.js(code), germanium.execute_script(code)

Execute the given JavaScript, and return its result.

```
germanium.js('return document.title;')
```

TIP The `js` is just an alias for the `execute_script` function

germanium.take_screenshot(name)

Takes a screenshot of the browser and saves it in the configured screenshot folder.

```
# will save a screenshot as `screenshots/test.png`
germanium.take_screenshot('test')
```

germanium.load_script(filename)

Loads the JavaScript code from the file with the given name into the browser.

```
germanium.load_script('jquery.js')
```

germanium.find_element_by_simple(locator)

Finds the element in the current iframe, using the simple locator given.

```
germanium.find_element_by_simple('"Title" > button["Ok"]')
```

germanium Static Functions

These are just a bunch of utility functions, that can even be used without germanium itself.

type_keys(germanium, keys_typed, element=None)

Type the current keys into the browser, eventually specifying the element to send the events to.

```
type_keys(germanium, "send data<cr>but <!shift>not<^shift> now.")
```

Special keys such as `ENTER`, are available by just escaping them in `<` and `>` characters, e.g. `<ENTER>`. For example to send `TAB TAB ENTER` someone could type:

```
type_keys(germanium, "<tab*2><enter>")
```

TIP	Using * in special keys or combined macros, allows you to type the same key, or key combination multiple times.

Also, in the typing of the keys, combined macros such as `<ctrl-a>` are automatically understood as `CTRL+A` and translated correctly as an action chain.

Macro keys can be written such as:

- `SHIFT`: `S`, `SHIFT`
- `CONTROL`: `C`, `CTL`, `CTRL`, `CONTROL`
- `META`: `M`, `META`

Also germanium is smart enough, so the position of the macro key matters, thus `<s-s>` is equivalent to `<shift-s>` and thus interpreted as `SHIFT+s`, and not `s+s` or `SHIFT+SHIFT`.

In order to start pressing a key, and release it latter, while still typing other keys, the `!` and `^` symbols can be used. For example to type some keys with `SHIFT` pressed this can be done:

```
type_keys(germanium, "<!shift>shift is down<^shift>, and now is up.")
```

TIP	The `!` looks like a finger almost pressing the button, and the `^` is self explanatory: the finger released the given button.

click(germanium, selector)

Perform a single click mouse action.

```
click(germanium, Button("Cancel").below(Text("Delete file?")))
```

double_click(germanium, selector)

Perform a double click mouse action.

```
double_click(germanium, "a.test-label")
```

right_click(germanium, selector)

Perform a mouse right click. Also known as a context menu click.

```
right_click(germanium, webdriver_element)
```

hover(germanium, selector)

Hover the given element.

```
hover(germanium, 'a.main-menu')
```

get_attributes(germanium, selector)

Return all the attributes of the element matched by the selector as a dictionary object.

For example for this HTML:

```
<body>
    <div id='editor' class='modal superb' custom-data='simple-code'></div>
</body>
```

To get all the attributes of the editor div, someone can:

```
editor_attributes = get_attributes(germanium, '#editor')
assert editor_attributes['class'] == 'modal superb'
assert editor_attributes['id'] == 'editor'
assert editor_attributes['custom-data'] == 'simple-code'
```

wait(closure, while_not=None, timeout=10)

A function that allows waiting for a condition to happen, monitoring also that some other conditions do not happen.

```
wait(germanium.S('"document uploaded successfully"'),
    while_not = germanium.S('"an error occured"'))
```

In case the timeout expires, or one of the `while_not` conditions matches until the `closure` is not yet matching then throws an exception.

`while_not` is either a closure, either an array of closures.

Germanium Selectors

In the `germanium.selectors` package there are several selectors that are already built in order to make test writing easier.

All of the selectors extend the `germanium.selectors.AbstractSelector` class, and the selector matching in Germanium called via S will invoke them.

Selectors allow abstracting how the elements in the DOM will be found, and also allow positional filtering depending on other elements position.

Positional Filtering

Germanium provides the following methods to enable positional filtering: `left_of()`, `right_of()`, `below()`, `above()`.

These filters can be used to filter otherwise false positive matches when selecting.

Multiple filters can be chained for the same selector, for example someone can:

```
click(Link("edit").below(Text("User Edit Panel")).right_of(Text("User 11")))
```

This will find a link that contains the label `edit`, that is positioned below the text `User Edit Panel` and is to the right of the text `User 11`.

selector.left_of(other_selector)

Make a selector that will return only the items that are left of all the elements returned by the other_selector.

```
click(Input().left_of(Text("User")))
```

selector.right_of(other_selector)

Make a selector that will return only the items that are right of all the elements returned by the other_selector.

```
click(Link("edit").right_of(Text("User 11")))
```

selector.above(other_selector)

Make a selector that will return only the items that are above all the elements returned by the

other_selector.

```
click(Link("logout").above("div.toolbar"))
```

selector.below(other_selector)

Make a selector that will return only the items that are below all the elements returned by the other_selector.

```
click(Button("edit").below(Text("entry 5")))
```

Custom Selectors

You can write a new selector by extending the AbstractSelector class and implementing the `get_selectors` method, that returns an array of selectors to be searched in the document.

Utility Selectors

Utility selectors are provided so you can use the positional filtering capabilities of the selectors. For example:

```
click(Css('.tree-plus-icon').left_of(Text('Item 15')))
```

Css(locator)

A selector that finds the given CSS expression.

XPath(locator)

A selector that finds the given XPath expression.

JsSelector(code)

A selector that finds an element by evaluating the given JavaScript code.

Provided Selectors

Provided selectors are just classes that are generally useful for testing, simple things such as buttons, links or text.

The most basic of them is called `Element`. There are a lot of more specific selectors on top of that, for

`Input`s, or `Link`s.

Element(tag_name=None, index=-1, exact_text=None, contains_text=None, css_classes=[], exact_attributes={})

A selector that finds a random element.

Parameters:

- `tag_name` - the html tag name to find (e.g. `div`, `span`, `li`);
- `index` - if specified, is the 1 index based result;
- `exact_text` - if specified, the exact text the element must have;
- `contains_text` - if specified, the exact text the element should contain;
- `css_classes` - the CSS classes that the element must have;
- `exact_attributes` - attributes with their values that the element must have.

```
S(Element('div',
          contains_text='error has occured',
          css_classes=['error-message']))
```

This will find a div that contains the text `error has occured` and has also a CSS class attached to it named `error-message`.

Button(search_text = None, text = None, name = None)

Just a selector that finds a button by its label or name:

This selector will find simultaneously both `input` elements that have the `type="button"`, but also `button` elements.

- some of the text, in either the `value` attribute if it's an `input`, either the text of the `button` (`search_text`)
- the text, either the `value` attribute if it's an input, either its text if it's an actual `button` (`text`)
- its form name (`name`)

```
germanium.S(Button("Ok"))
```

Input(input_name)

Just a selector that finds an input by its name.

```
germanium.S(Input('q'))
```

InputText(input_name)

Just a selector that finds an input with the type `text` by its name.

```
germanium.S(InputText('q'))
```

Link(search_text, text, search_href, href)

Just a selector that finds a link by either:

- some of its text content (`search_text`)
- its exact text content(`text`)
- some of its link location (`search_href`)
- its link location(`href`)

To match the first link that contains the *test* string, someone can:

```
germanium.S(Link("test"))
```

Of course, the text and href search can be combined, so we can do, in order to find a link that is on the `ciplogic.com` that has in the text `testing`:

```
germanium.S(Link("testing", search_href="http://ciplogic.com"))
```

Text(text)

Just a selector that finds the element that contains the text in the page.

```
germanium.S(Text("some text"))
```

The selector can find the text even in formatted text. For example the previous selector would match the parrent div in such a DOM structure:

```
<div>
    some <b>text</b>
</div>
```