# Germanium v1.10.3

Web Testing API that doesn't disappoint

# Table of Contents


Installation . . . . . 1
Germanium Drivers . . . . . 2
GERMANIUM_DRIVERS_FOLDER . . . . . 2
GERMANIUM_USE_PATH_DRIVER . . . . . 2
GERMANIUM_USE_IE_DRIVER_FOR_PLATFORM . . . . . 2
Germanium Static . . . . . 3
open_browser() . . . . . 3
close_browser() . . . . . 4
go_to(url) . . . . . 4
type_keys(keys, selector, delay) . . . . . 4
click(selector_or_point) . . . . . 5
hover(selector_or_point) . . . . . 5
double_click(selector_or_point) . . . . . 6
right_click(selector_or_point) . . . . . 6
drag_and_drop(from_selector_or_point, to_selector_or_point) . . . . . 7
select(selector, text?, index?, value?) . . . . . 7
deselect(selector, text?, index?, value?) . . . . . 8
select_file(selector, file_path, path_check=True) . . . . . 9
parent_node(selector) . . . . . 10
child_nodes(selector, only_elements=True) . . . . . 10
get_value(selector) . . . . . 11
get_text(selector) . . . . . 12
get_attributes(selector) . . . . . 12
get_style(selector, name) . . . . . 13
get_web_driver() . . . . . 13
get_germanium() . . . . . 14
highlight(selector, show_seconds=2) . . . . . 14
def S(*argv, **kwargs) . . . . . 15
def iframe(target, keep_new_context = False) . . . . . 15
wait(closure, while_not=None, timeout=10) . . . . . 15
waited(closure, while_not=None, timeout=10) . . . . . 17
Germanium Selectors and Locators . . . . . 18
Locators Overview . . . . . 18
String Selectors . . . . . 19
Selectors Overview . . . . . 20
Writing Custom Selectors . . . . . 20

Selectors Positional Filtering . . . 22
selector.left_of(other_selector) . . . 22
selector.right_of(other_selector) . . . 22
selector.above(other_selector) . . . 23
selector.below(other_selector) . . . 23
Selectors DOM Filtering . . . 24
selector.containing(selector..) . . . 24
selector.containing_all(selector..) . . . 24
selector.inside(selector..) . . . 25
selector.without_children() . . . 25
Germanium Selectors in Static Contexts . . . 27
selector.element() . . . 27
selector.element_list() . . . 27
selector.exists() . . . 28
selector.not_exists() . . . 29
selector.text() . . . 29
Utility Selectors . . . 30
Css(locator) . . . 30
XPath(locator) . . . 30
JsSelector(code) . . . 30
Provided Selectors . . . 30
Element(tag_name=None, ...) . . . 31
Button(search_text = None, text = None, name = None) . . . 31
Input(input_name) . . . 32
InputText(input_name) . . . 32
Link(search_text, text, search_href, href) . . . 32
Text(text) . . . 32
Point Support . . . 34
Point(x, y) . . . 34
Box(selector) . . . 34
Germanium Keys Support . . . 37
Regular Typing . . . 37
Special Keys . . . 37
Combo Presses . . . 37
Press-Release Key . . . 38
Germanium API Documentation . . . 39
@iframe - germanium iframe decorator . . . 39
germanium Instance Functions . . . 40

germanium Instance Attributes . . . . . 44
germanium Utility Functions . . . . . 44

# Installation

To install it just run:

```
pip install germanium
```

You don't need any binary drivers installed, or any other dependencies, since they are bundled (and tested) by Germanium itself.

Writing a test then becomes as easy as:

```
from germanium.static import *
from time import sleep

open_browser("ff")
go_to("http://www.google.com")
type_keys("germanium pypi<enter>", Input("q"))
wait(Link("Python Package Index"))
click(Link("Python Package Index"))
sleep(5)
close_browser()
```

Germanium supports Python 2.7, 3.4 and 3.5, and is already used in production tests.

Browsers supported are:

- IE 8+
- Chrome
- Firefox

# Germanium Drivers

Starting with version 1.8 Germanium also packages the WebDriver binary drivers inside, and will unpack them when starting a new browser.

Thus when using Germanium it's not required anymore to have the drivers downloaded.

## GERMANIUM_DRIVERS_FOLDER

Path where to unpack the drivers if they are missing, or if a wrong version is detected. If it's not set Germanium will create a folder in the temp folder named `germanium-drivers`.

```
export GERMANIUM_DRIVERS_FOLDER=/opt/germanium-drivers
```

## GERMANIUM_USE_PATH_DRIVER

If there is a driver for the current browser in the PATH, even if the version of the driver is unsupported, use that one instead the embedded binary driver that Germanium ships.

If an unsupported driver is found, Germanium will still use its internal driver.

```
export GERMANIUM_USE_PATH_DRIVER=1
```

## GERMANIUM_USE_IE_DRIVER_FOR_PLATFORM

The IE driver for 64 bit has known issues, so if Germanium needs an IE driver will implicitly use the 32 bit version.

```
export GERMANIUM_USE_IE_DRIVER_FOR_PLATFORM=1
```

# 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.

## open_browser()

**Description**

Opens the given browser instance.

**Signature**

```
def open_browser(browser="Firefox", ①
                 wd=None, ②
                 iframe_selector=DefaultIFrameSelector(), ③
                 screenshot_folder="screenshots", ④
                 scripts=list()) ⑤
```

① *browser* - The browser is case insensitive and can be one of:

1. "ff" or "firefox" - to start Mozilla Firefox
2. "chrome" - to start Google Chrome
3. "ie" - to start Microsoft Internet Explorer

② *wd* - A specific already created WebDriver instance can also be given, and then the *browser* parameter will be ignored.

③ *iframe_selector* - The strategy to use when finding the execution iframe, whenever the active iframe name changes.

④ *screenshot_folder* - Folder under browser screenshots are saved.

⑤ *scripts* - A list of JavaScript resources to be loaded whenever a page is newly loaded.

**Sample**

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

This also connecting to remote drivers, for example:

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

# close_browser()

**Description**

Close the currently running browser instance that was opened with `open_browser()`

**Signature**

```
def close_browser()
```

**Sample**

```
close_browser()
```

# go_to(url)

**Description**

Go to the given URL, and wait for the page to load. After the page will load, the scripts provided in the creation of the GermaniumDriver object will be automatically loaded.

**Signature**

```
def go_to(url) ①
```

① *url* - The URL to load in the browser.

**Sample**

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

# type_keys(keys, selector, delay)

**Description**

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

**Signature**

```
def type_keys(keys, ①
              selector=None, ②
              delay=0) ③
```

① *keys* - the keys to press. *See the **Germanium Keys Support**, to learn about having multiple keypresses, combo key presses, or repetitions.*

② *selector - optional* For what element to send the keys. In case it's missing, sends the keys to the active element. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

③ *delay - optional* Delay in seconds between each keypress.

**Sample**

```
type_keys('john.doe@example.com', Input('email')) ①
type_keys("<tab*2><enter>") ②
```

① Type in the input with the `name` attribute equal to `email`.

② Type in the currently active element in the current iframe.

# click(selector_or_point)

**Description**

Click the element with the given selector, or at the specified point position.

**Signature**

```
def click(selector_or_point) ①
```

① *selector_or_point* - What element to click. *See the **Germanium Selectors** and **Point Support**, to learn about how you can easily locate the element you want your action to be triggered against.*

**Sample**

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

# hover(selector_or_point)

**Description**

Hovers (sends a mouse over) the element with the given selector, or at the specified point position.

### Signature

```
def hover(selector_or_point) ①
```

① *selector_or_point* - What element to hover. *See the **Germanium Selectors** and **Point Support**, to learn about how you can easily locate the element you want your action to be triggered against.*

### Sample

```
hover(Element('div', id='menu1'))
```

## double_click(selector_or_point)

### Description

Double clicks the element with the given selector, or at the specified point position.

### Signature

```
def double_click(selector_or_point) ①
```

① *selector_or_point* - What element to double click. *See the **Germanium Selectors** and **Point Support**, to learn about how you can easily locate the element you want your action to be triggered against.*

### Sample

```
double_click(Element('div', css_classes='table-row'))
```

## right_click(selector_or_point)

### Description

Right clicks the element with the given selector, or at the specified point position.

### Signature

```
def right_click(selector_or_point) ①
```

① *selector_or_point* - What element to right click. *See the **Germanium Selectors** and **Point Support**, to learn about how you can easily locate the element you want your action to be triggered against.*

### Sample

```
right_click(Element('div', css_classes='table-row'))
```

# drag_and_drop(from_selector_or_point, to_selector_or_point)

**Description**

Performs a drag and drop operation from the element matching the `from_selector_or_point`, to the element matching the `to_selector_or_point`.

Both `from_selector_or_point` and `to_selector_or_point` can as the name suggest be either selectors, or point locations, and are not required to have the same type. You can start a drag from a selector, to a point, or vice-versa.

**Signature**

```
def drag_and_drop(from_selector_or_point, ①
                  to_selector_or_point) ②
```

① *from_selector_or_point* - What element to use for drag start. *See the **Germanium Selectors** and **Point Support**, to learn about how you can easily locate the element you want your action to be triggered against.*

② *to_selector_or_point* - What element to release the mouse over. *See the **Germanium Selectors** and **Point Support**, to learn about how you can easily locate the element you want your action to be triggered against.*

**Sample**

```
drag_and_drop(Element("div", css_classes="old-entry", index=2),
              "#removeContainer")
```

# select(selector, text?, index?, value?)

**Description**

Change the value of a `<select>` element by selecting items from the available options.

**Signature**

```
def select(selector, ①
           text=None, ②
           *argv,
           index=None, ③
           value=None, ④
           **kw)
```

① *selector* - What select to change its values. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

② *text* - What text(s) (if any) to use for selection.

③ *index* - What index(es) (if any) to use for selection.

④ *value* - What value(s) (if any) to use for selection.

One of `text`, `index` or `value` must be present for the selection to function, if none are present an `Exception` will be raised.

`text`, `index` and `value` can also be arrays, or single values.

**Sample**

```
select("#country", "Austria")
```

# deselect(selector, text?, index?, value?)

**Description**

Change the value of a `<select>` element by deselecting items from the available options.

**Signature**

```
def deselect(selector, ①
             text=None, ②
             *argv,
             index=None, ③
             value=None, ④
             **kw)
```

① *selector* - What select to change its values. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

② *text* - What text(s) (if any) to use for deselection.

③ *index* - What index(es) (if any) to use for deselection.

④ *value* - What value(s) (if any) to use for deselection.

Deselect will deselect all the items from the `text`, `index` and `value` parameters. *If all the parameters are unset, it will clear the selection.*

`text`, `index` and `value` can also be arrays, or single values.

**Sample**

```
deselect("#products", index=[1,3])
```

# select_file(selector, file_path, path_check=True)

**Description**

Selects the file into a file input from the disk. The file itself must exist on the system where the browser is running.

**Signature**

```
select_file(selector, ①
            file_path, ②
            path_check=True) ③
```

① *selector* - What file input to select the file for. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

② *file_path* - Path to the file that should be selected in the file input.

③ *path_check* - Check if the file exists, and convert it to an absolute path for the upload.

In case the `path_check` is unset, any path will be sent to the driver without any validation. This is useful for uploading files on a remote WebDriver browser.

WebDriver requires the path to be absolute. Germanium will convert the path to an absolute location only it `path_check` is set to `True`.

**Sample**

Selecting for upload a relative path:

```
select_file(InputFile(),
            'features/steps/test-data/upload_test_file.txt')
```

Selecting for upload a path that is available only remotely:

```
select_file(InputFile(),
            r"c:\features\steps\test-data\upload_test_file.txt",
            path_check=False)
```

# parent_node(selector)

**Description**

Gets the parent node of the given selector.

**Signature**

```
parent_node(selector) ①
```

① *selector* - What element to return the value for. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

This will return a `WebElement`.

**Sample**

```
e = parent_node('#some_element')
```

Will return the parent node for the element with the ID `some_element` that will be matched by the CSS locator.

# child_nodes(selector, only_elements=True)

**Description**

Gets the child nodes of the element that is matched by selector.

**Signature**

```
child_nodes(selector, ①
            only_elements=True) ②
```

① *selector* - What element to return the value for. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

② *only_elements* - If to return only elements, or also other node types (text, comment, etc)

This will return a `list` of the found elements, or an empty list if no element was found.

**Sample**

For example for the given HTML:

```
<div id="parent">
    <div id="child1">..</div>
    <div id="child2">..</div>
</div>
```

When calling:

```
items = child_nodes("#parent")
assert len(items) == 2
```

This will return a list of 2 elements, with the child1 and child2, since `only_elements` is set by default to true. Otherwise if setting the `only_elements` to `False`, the call will return 5 elements, since there are 3 whitespace nodes in the `#parent` div.

```
items = child_nodes('#parent', only_elements=False)
assert len(items) == 5
```

# get_value(selector)

**Description**

Gets the value of an input element. Works for: `<input>` and `<select>` elements.

**Signature**

```
get_value(selector) ①
```

① *selector* - What element to return the value for. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

`get_value` will return the current value of the element.

If the element is a multi-select, it will return an array of the values which were selected (the `value` attribute of the `<option>` elements that are selected).

**Sample**

```
assert get_value('#country') == 'at'
```

# get_text(selector)

**Description**

Gets the text from the element. This is equivalent to the `innerText`, or `textContent` element attribute of the browser.

**Signature**

```
get_text(selector) ①
```

① *selector* - What element to return the text for. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

If the selector is a `WebElement` instance, the filtering of `only_visible` will not be used, and the text from the given element will still be returned.

This is in contrast with the default Selenium approach of returning empty text for elements that are not visible.

**Sample**

```
get_text(invisible_element)
```

or

```
assert 'yay' == get_text('.success-message') ①
```

① This might throw exceptions if the `.success-message` is an element that is invisible, or doesn't exists.

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

# get_style(selector, name)

**Description**

Returns a single CSS attribute value for the element that is matched by the selector.

**Signature**

```
get_style(selector, ①
          name) ②
```

① *selector* - What element to return the CSS property for. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

② *name* - The name of the property to return, in camel case.

If the selector is a `WebElement` instance, the filtering of `only_visible` will not be used, and the style property from the given element will still be returned, even if the element is not visible.

**Sample**

```
get_style('input.red-border', 'borderTopWidth')
```

# get_web_driver()

**Description**

Return the WebDriver instance the global Germanium was built around.

**Signature**

```
def get_web_driver()
```

**Sample**

```
wd = get_web_driver()
```

# get_germanium()

**Description**

Returns the currently running Germanium instance, or `None` if no instance was opened using `open_browser()`.

**Signature**

```
def get_germanium()
```

*Please see the **Germanium API Documentation** to find out what is available on the `germanium.driver.GermaniumDriver` instance.*

**Sample**

```
g = get_germanium()
```

# highlight(selector, show_seconds=2)

**Description**

Highlights by blinking the background of the matched selector with a vivid green for debugging purposes.

**Signature**

```
def highlight_g(selector, ①
                show_seconds=2, ②
                *args,
                console=False) ③
```

① *selector* - What element to alternate the background for. *See the **Germanium Selectors**, to learn about how you can easily locate the element you want your action to be triggered against.*

② *show_seconds* - How many seconds should the element blink.

③ *console* - Should the messages be logged to the browser console.

In case the element that is found doesn't exist, or is not visible, a notification alert will pop up, with information of whether the element was not found or since it's not visible can't be highlighted.

In case `console` is set to `True` then the alert will not be displayed, but instead only the `console.log` (or `console.error`) of the browser will be used for notifying elements that are not visible, or that can not be found.

**Sample**

```
highlight('.hard-to-see-item')
```

# def S(*argv, **kwargs)

**Description**

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

**Signature**

```
def S(selector, strategy='default')
```

**Sample**

```
element = S('#editor').element()
```

# 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'
```

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

**Description**

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

In case the timeout expires, or one of the `while_not` conditions matches until the `closure` is not yet

matching then throws an exception.

Callables of both `closure` and the `while_not` are recursively resolved until a non callable *trueish* value is returned.

**Signature**

```
def wait(closure, ①
         while_not=None, ②
         timeout=10) ③
```

① *closure* is either a callable, either an array of callables. If any of them passes, the wait finished successfuly.

② *while_not* is either a callable, either an array of callables. If any of them fail, the wait throws an exception.

③ *timeout* expressed in seconds.

**Sample**

Since selectors are callables, they can be used as parameters for `wait`.

```
wait(Text("document uploaded successfully"),
    while_not = Text("an error occurred"))
```

Because callables are recursively resolved, they can be used as strategies for waiting:

```
def ButtonOrLink():
    if some_condition:
        return Link

    return Button

wait(ButtonOrLink)
```

This is roughly equivalent to:

```
def ButtonOrLink():
    if some_condition:
        return Link().exists()

    return Button().exists()

wait(ButtonOrLink)
```

# waited(closure, while_not=None, timeout=10)

**Description**

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

In case the timeout expires, or one of the `while_not` conditions matches, before the `closure` matched then it returns `None`.

Otherwise it returns the value that the closure returned.

**Signature**

```
def waited(closure, ①
           while_not=None, ②
           timeout=10) ③
```

① *closure* is either a callable, either an array of callables. If any of them passes, the wait finished successfuly.

② *while_not* is either a callable, either an array of callables. If any of them fail, the wait throws an exception.

③ *timeout* expressed in seconds.

**Sample**

```
click(waited(Button("Ok")))
```

# Germanium Selectors and Locators

`Selector` objects are similar to `String` values, that describe how an element can be found in the current page, while `Locator` objects are the implementation of actual aglorithms that find them. A parallel can be made between the string `"div.custom-text"`, and the `webdriver.find_element_by_css()` function. Selectors specify what you want to find in the page, and locators make sure you find them. It's the combination of them, `webdriver.find_element_by_css("div.custom-text")` that will return the actual DOM Element to interact with.

Selectors are in the end text strings. Locators evaluate them finding elements in the browser.

In all the API calls, where `selector` is specified, the selector is actually one of:

1. a string selector,
2. an object that inerits from `AbstractSelector` (such as `Text`, `Element`, `Image`, etc.),
3. a WebDriver WebElement,
4. a locator,
5. a list of any of the above.

Since selectors offer Positional and DOM filtering, point 1 and 2 will cover 99% of your test cases.

## Locators Overview

Locators are algorithms that are are able to find elements against the current browser. They are registered on the Germanium instance, and by default, Germanium comes with three locators registered: `"xpath"`, `"css"` and `"js"`. These are implemented in `XPathLocator`, `CssLocator` and `JsLocator` respectively, from the `germanium.locators` package. Locators use selectors to find web elements. To create a locator you need a Germanium instance, and a string specifying the selector passed to the locator itself.

These locators all extend a base class named `DeferredLocator`. This class holds the reference to the `Germanium` object, and offers utility methods to actually fetch the elements, check if such elements exist, or retrieveing their text.

Note, that the locators don't immediately find the elements. Explicit calls muse be made to:

- `element()`
- `element_list()`
- the locator itself with `()`, (since the locator is a callable and will return the element_list)

```
from germanium.util import wait

label_divs_locator = germanium.S('.label')  ①
wait(label_divs_locator) ②
```

① This will return a `CssLocator`.

② Since the locator is a callable, we can wait on it

A locator is always constructed with two things: the `Germanium` instance it will use to attept at finding the elements, and a `string` expression that will be used for finding. Note that you should never manually instantiate the locator, but you should use the super locator (the `S` function). This function will pass both the germanium instance, and the selector itself.

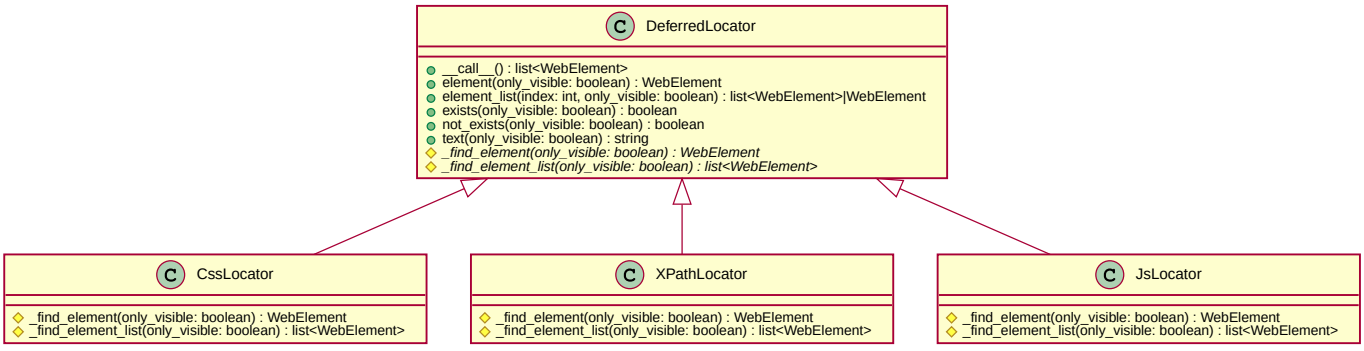
You can, and should, use the strategy parameter or the selector prefix when using the `S()` builder function:

```
germanium.S('#testDiv', strategy='css')
```

or prefixing the string itself with the strategy name:

```
germanium.S('css:#testDiv')
```

Optionally a custom locator can be defined that extends the base class `DeferredLocator`. `DeferredLocator` contains a reference to a `Germanium` object and includes utility methods to get web elements.




# String Selectors

A string selector is a selector that can specify what locators to be used. Implicitly, the selector is either an XPath if it starts with `"//"`, either a CSS selector, if there is no identifier prefix (`"name:…"`).

A string selector can also specify its locator strategy, by prefixing the selector with the locator strategy name. Currently registered into Germanium are:

**css**

```
selector = "css:div#customID"

# or without the css prefix, since the string it's
# not starting with //
selector = "div#customID"
```

**xpath**

```
selector = "xpath://div[@id='customID']"

# or without the xpath prefix, since the string it's
# starting with //
selector = "//div[@id='customID']"
```

**js**

```
selector = "js:return [ document.getElementById('customID') ];"
```

# Selectors Overview

All `Selector` objects in Germanium inherit from `germanium.selector.AbstractSelector`, which define only a single required method `get_selectors()` that returns a list of string selectors. The list item can have different locator strategies:

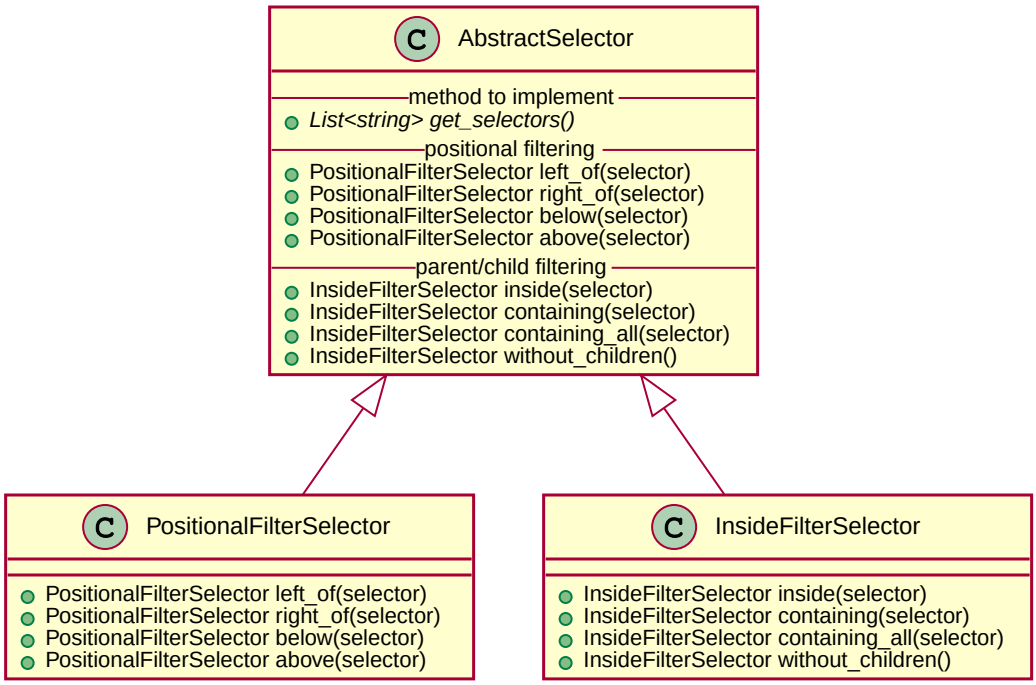
```
class AbstractSelector(object):
    # ...
    def get_selectors(self):
        raise Exception("Abstract class, not implemented.") ①

    # ... positional, and parent-child filtering methods
```

All the Selector objects return a list of strings, that define how the element, or the multiple elements will be found by the given locator.

# Writing 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.


AbstractSelector
method to implement
List<string> get_selectors()
positional filtering
PositionalFilterSelector left_of(selector)
PositionalFilterSelector right_of(selector)
PositionalFilterSelector below(selector)
PositionalFilterSelector above(selector)
parent/child filtering
InsideFilterSelector inside(selector)
InsideFilterSelector containing(selector)
InsideFilterSelector containing_all(selector)
InsideFilterSelector without_children()
PositionalFilterSelector
PositionalFilterSelector left_of(selector)
PositionalFilterSelector right_of(selector)
PositionalFilterSelector below(selector)
PositionalFilterSelector above(selector)
InsideFilterSelector
InsideFilterSelector inside(selector)
InsideFilterSelector containing(selector)
InsideFilterSelector containing_all(selector)
InsideFilterSelector without_children()

# Selectors Positional Filtering

Germanium provides the following methods directly on top of `AbstractSelector` to enable positional filtering: `left_of(selector)`, `right_of(selector)`, `below(selector)`, `above(selector)`, that are from the set of found web elements, by using reference elements, and ignoring elements `left_of`, `right_of`, `below` or `above` the references. 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)

**Description**

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

**Signature**

```
def left_of(self, other_selector)
```

**Sample**

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

## selector.right_of(other_selector)

**Description**

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

**Signature**

```
def right_of(self, other_selector)
```

**Sample**

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

# selector.above(other_selector)

**Description**

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

**Signature**

```
def above(self, other_selector)
```

**Sample**

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

# selector.below(other_selector)

**Description**

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

**Signature**

```
def below(self, other_selector)
```

**Sample**

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

# Selectors DOM Filtering

DOM Filtering selectors work by selecting only specific nodes in relations with other nodes in the DOM.

## selector.containing(selector..)

**Description**

Matches nodes that contain the other XPath/CSS selectors.

**Signature**

```
def containing(self, selector..)
```

**Sample**

```
row = Element("tr").containing(
        Element("td", contains_text="User 1"),
        Element("td", contains_text="User 2")
      ).element()
```

This will match a `<tr>` element that contains any of the `<td>` elements with the `"User 1"` or `"User 2"` text.

## selector.containing_all(selector..)

**Description**

Matches nodes that contain all the given selectors inside their tree structure.

**Signature**

```
def containing_all(selector..)
```

**Sample**

```
row = Element("tr").containing_all(
        Element("td", contains_text="user@sample.com"),
        Text("User A")
    ).element()
```

This will match a `<tr>` element that contains a `<td>` with the text `user@sample.com` and some other text, named `"User A"`

# selector.inside(selector..)

**Description**

Matches nodes that are inside any of the other selectors.

**Signature**

```
def inside(self, selector)
```

**Sample**

```
error_message = Element("div", css_classes="label") \
                    .inside(Element("div", css_classes="error-dialog"))
```

# selector.without_children()

**Description**

Matches nodes that have no children.

**Signature**

```
def without_children(self)
```

**Sample**

Given this selector:

```
Element('div', css_classes='test').without_children()
```

and HTML:

```
<div>
    <div class="test">a</div>
    <div class="test"><node/></div>
    <div class="test"></div> <!-- only this node will be matched -->
    <div class="test"><node>mix</node></div>
<div>
```

only the third `<div>` child element will be matched.

# Germanium Selectors in Static Contexts

Selectors are neat since we can reuse them, and offer a clean separation between finding the elements and inspecting them, but they also offer a few utility methods to aid you in removing that one extra call to the `S` super locator.

For example instead of writing:

```
S(Button('Ok')).element()
```

you can write:

```
Button('Ok').element()
```

*but you need to have a germanium instance already opened, or manually specify it in the element call.*

```
Button('Ok').element(germanium=my_custom_ge_instance)
```

## selector.element()

**Description**

This function allows fetching the first element from the Germanium instance, for which the current selector matches.

In case the germanium instance is not specified it will use the static instance from `germanium.static.get_germanium()`.

**Signature**

```
def element(self, *argv, germanium=None, only_visible=True)
```

**Sample**

```
Button('Ok').element()
```

## selector.element_list()

**Description**

This function allows fetching the element list from the Germanium instance, for which the current selector matches.

In case the germanium instance is not specified it will use the static instance from `germanium.static.get_germanium()`.

**Signature**

```
def element_list(self,
                 index=None, ①
                 *argv,
                 germanium=None, ②
                 only_visible=True) ③
```

① *index* - When present, the element with the given index will be returned instead of the full list of elements.

② *germanium* - What instance of germanium to use. If `None` use `germanium.static.get_germanium()`.

③ *only_visible* - If only the visible elements should be selected. Defaults to `True` across Germanium.

**Sample**

```
Element('li').element_list()
```

# selector.exists()

**Description**

This function allows checking if there is at least one element matching the current selector.

In case the germanium instance is not specified it will use the static instance from `germanium.static.get_germanium()`.

**Signature**

```
def exists(self, *argv, germanium=None, only_visible=True)
```

**Sample**

```
wait(Text('data saved successfuly').exists)
```

# selector.not_exists()

**Description**

This function allows checking if there is no element matching the current selector.

In case the germanium instance is not specified it will use the static instance from `germanium.static.get_germanium()`.

**Signature**

```
def not_exists(self, *argv, germanium=None, only_visible=True)
```

**Sample**

```
wait(Text('error occurred').not_exists)
```

# selector.text()

**Description**

This function allows returning the text of the first element that matches the current selector.

In case the germanium instance is not specified it will use the static instance from `germanium.static.get_germanium()`.

**Signature**

```
def text(self, *argv, germanium=None, only_visible=True)
```

**Sample**

```
assert Css('#messages').text() == 'data persisted'
```

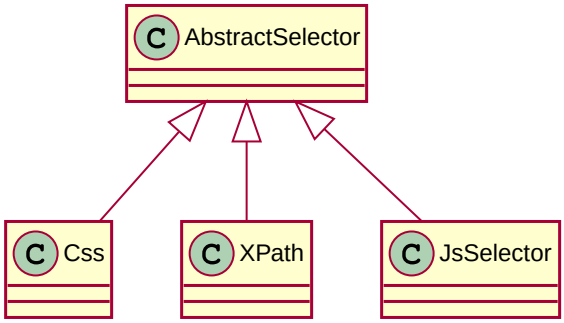
# Utility Selectors

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')))
```

The reason behind them is that you can't use positional filtering directly on the string themselves. String objects have to be recast to another object type (in this case, `AbstractSelector`) that supports the positional filtering methods.

```
click('.tree-plus-icon'.left_of(Text('Item 15'))) # throws exception
```




## 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. `arguments[0]` is the element used for subtree searches, and can be `null` if searches are made for the full document.

## 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, ...)

A selector that finds an element by looking at its XPath.

Parameters:

- `tag_name` - the html tag name to find (e.g. `div`, `span`, `li`);
- `index` - if specified, is the 1 index based result;
- `id` - If it's specified, is the id attribute of the element;
- `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 (either as a `string`, or `list` of `string`s);
- `exact_attributes` - attributes with their values that the element must have (`dict`, keys for attribute names, values for expected values);
- `contains_attributes` - attributes that contain the given values (`dict`, keys for attribute names, values for strings that the attribute values must contain);
- `extra_xpath` - extra xpath to be added to the expression, to the previously built expressions.

If the `index` is used, the whole expression is wrapped in parenthesis, and the index is applied to the whole result. In case you want multiple sub-children, use `extra_xpath` to fetch the elements.

```
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`, or the text of the `button` (`search_text`)
- the exact text, either the `value` attribute if it's an input, or 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 exact 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` website containing 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 parent div in such a DOM structure:

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

# Point Support

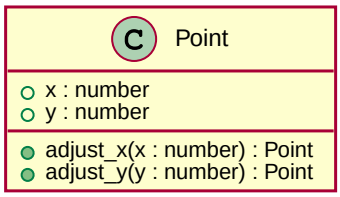
| | |
|---|---|
| **WARNING** | While Germanium *does* support IE8+, point support is supported only from IE9+. This is the only Germanium API that is not supported in IE8, and is actually caused by Selenium itself. |

In Germanium, point actions are supported for all the mouse actions, and can be used instead of selectors, namely:

1. `click()`
2. `right_click()`
3. `double_click()`
4. `hover()`
5. `drag_and_drop()`

## Point(x, y)

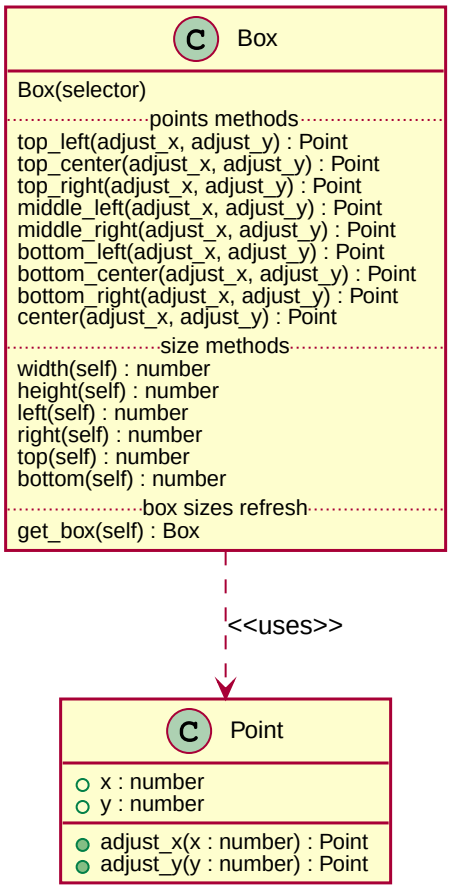
*Points are not selectors and don't specify an exact element*, but rather as the name implies a location on the screen where we want to interact. The point location is computed from the top/left of the page itself, and is specified with `x` and `y` coordinates.




Points can be adjusted, so you can have the value changed without always summing up values in a one liner.

In order to easily obtain points, a utility class is also provided that can obtain points relative to the bounding box of an element. The corners, middle of the top/left/right/bottom segments, and the center are offered as points. The class is named `Box`, and its constructor accepts a selector as an argument.

## Box(selector)




A `Box` instance will keep the sizes the first time it will be called, because we don't want to call it every time. In order to refresh it, the `get_box()` method is offered that will refresh the `Box` coordinates with the new data.

In `wait` conditions you can chain it:

```
box = Box(Css('.resizing-div'))
wait(lambda: box.get_box().width() == 100))
```

Since points are not selectors, you can click two pixels right of an element, without exactly specifying the target element like so:

```
click(Box('span.custom-text').middle_right(2, 0))
```

To click two pixels left of an element, we can just adjust with a negative value:

```
click(Box('span.custom-text').middle_left(-2, 0))
```

Assuming a canvas is more than 10x10 pixels, we could also do a drag and drop from the top left corner, to the bottom right, keeping a 5 pixel margin:

```
canvas_box = Box('canvas.drawing')
drag_and_drop(canvas_box.top_left(5, 5),
        canvas_box.bottom_right(-5, -5))
```

# Germanium Keys Support

This section details on how to type keys better, without a headache.

## Regular Typing

In general when typing keys, for example for form fields, the easiest way of doing it is to just type the actual keys to be pressed. For example to type the user name into a form field you can:

```
type_keys('John', Input('firstname'))
```

This will in turn just type the keys `["J", "o", "h", "n"]` into the input that has a `name` attribute equal to `"firstname"`. An email looks equally fascinating:

```
type_keys('john.doe@example.com', Input('email'))
```

Let's start the more interesting examples.

## Special Keys

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("<tab*2><enter>")
```

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

Now you might wonder, why is it `<enter>` and not `<ENTER>`? Or `<cr>`? Or its bigger brother `<CR>`? Or just `<Enter>`. Actually they all resolve to the same key, that is the `ENTER`. The same holds true for `<del>` vs `<delete>`, or `<bs>` vs `<backspace>`, etc. They will resolve to `DELETE`, `BACKSPACE`, etc. as expected.

## Combo Presses

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.

## Press-Release Key

In order to start pressing a key, and release it later, 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("<!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.

# Germanium API Documentation

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

1. decorator:
   - `@iframe`
2. germanium instance functions:
   - `S`, super locator
   - `js`, `execute_script`
   - `take_screenshot`
   - `load_script`
3. germanium instance attributes:
   - `iframe_selector`
4. utility functions:
   - `type_keys_g`
   - `click_g`
   - `double_click_g`
   - `right_click_g`
   - `hover_g`
   - `select_g`
   - `deselect_g`
   - `get_attributes_g`
   - `get_value_g`
   - `get_text_g`
   - `highlight_g`
   - `wait`

# @iframe - 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:

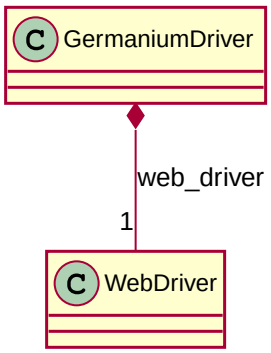
```
@iframe("default")
def close_dialog(germanium):
    germanium.S(Button("Ok").below(Text("Save dialog"))).element().click()

@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` then this property will be used.

# germanium Instance Functions

The GermaniumDriver is a simple instance that decorates an existing WebDriver:




All the attributes that are not defined on the `GermaniumDriver` instance, are searched into the `germanium.web_driver` one. For example calling:

```
print(germanium.title)
```

Will actually result in fetching the title from the web_driver instance that is used by the GermaniumDriver.

## Constructor GermaniumDriver(web_driver, ..)

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

```
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)`, or a method that has two parameters `(germanium, iframe_name)` can be provided and it will be wrapped into a decorator class by Germanium itself.

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(selector, 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 very similar to jquery's `$`.

The first parameter, the selector, 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')
```

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 Instance Attributes

Currently there is only one attribute, namely the `iframe_selector`, that allows changing the current iframe selection strategy for the given instance.

As in the constructor, it supports both the class, or the callable as values for assignment.

```
def new_iframe_selector(germanium, iframe_name):
    # ...

old_ifame_selector = get_germanium().iframe_selector
get_germanium().iframe_selector = new_iframe_selector
```

This is useful for reusing the Germanium instance across tests, without the need to recreate it just because you need another `iframe_selector` strategy.

## germanium Utility Functions

Utility functions for Germanium instances.

### type_keys_g(germanium, keys_typed, element=None, delay=0)

Type the current keys into the browser, optionally specifying the element to send the events to, and/or delay between keypresses.

```
type_keys_g(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_g(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_g(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_g(germanium, selector)

Perform a single click mouse action.

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

## double_click_g(germanium, selector)

Perform a double click mouse action.

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

### right_click_g(germanium, selector)

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

```
right_click_g(germanium, webdriver_element)
```

### hover_g(germanium, selector)

Hover the given element.

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

### select_g(germanium, selector, text=None, *argv, value=None, index=None)

Select one or more elements in a HTML `<select>` element. Can select the elements by either, text values, actual values inside the `<option>`, or by index.

```
select('select#country', value='at')
select('select#multivalueSelect', index=[1,3,7,8])
```

### deselect_g(germanium, selector, text=None, *argv, value=None, index=None)

Deselects one or more elements in a HTML `<select>` element. Can deselect the elements by either, text values, actual values inside the `<option>`, or by index.

```
deselect('select#multivalueSelect', index=[7,8])
```

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

## get_value_g(germanium, selector)

Returns the current value of the element matched by the selector. Normally for inputs it's just the string value.

In case the selector matches a multiple select, will return an array with the values that are currently selected.

```
assert get_value_g(germanium, 'select#multivalueSelect') == [1, 3]
```

## get_text_g(germanium, selector)

Returns the current text of the element matched by the selector. This will work also for `WebElement` instances that are passed as `selector` values even if they are not visible.

## highlight_g(germanium, selector)

Highlights the given selector on the germanium instance for debugging purposes. This will make the element blink in the actual browser for easy visual identification.