SeleniumLibrary

Search

Keywords

Library version:	5.0.0b1
Library scope:	GLOBAL
Named arguments:	supported

Introduction

SeleniumLibrary is a web testing library for Robot Framework.

This document explains how to use keywords provided by SeleniumLibrary. For information about installation, support, and more, please visit the project pages. For more information about Robot Framework, see http://robotframework.org.

SeleniumLibrary uses the Selenium WebDriver modules internally to control a web browser. See http://seleniumhq.org for more information about Selenium in general and SeleniumLibrary README.rst Browser drivers chapter for more details about WebDriver binary installation.

Locating elements

All keywords in SeleniumLibrary that need to interact with an element on a web page take an argument typically named `locator` that specifies how to find the element. Most often the locator is given as a string using the locator syntax described below, but *using WebElements* is possible too.

Locator syntax

SeleniumLibrary supports finding elements based on different strategies such as the element id, XPath expressions, or CSS selectors. The strategy can either be explicitly specified with a prefix or the strategy can be implicit.

Default locator strategy

By default, locators are considered to use the keyword specific default locator strategy. All keywords support finding elements based on `id` and `name` attributes, but some keywords support additional attributes or other values that make sense in their context. For example, *Click Link* supports the `href` attribute and the link text and addition to the normal `id` and `name`.

Examples:

Click Element	example	# Match based on `id` or `name`.

SeleniumLibrary

Click Link	example	# Match also based on link text and `href`.
Click Button	example	# Match based on `id`, `name` or `value`.

If a locator accidentally starts with a prefix recognized as *explicit locator strategy* or *implicit XPath strategy*, it is possible to use the explicit `default` prefix to enable the default strategy.

Examples:

Click Element	name:foo	# Find element with name `foo`.
Click Element	default:name:foo	# Use default strategy with value `name:foo`.
Click Element	//foo	# Find element using XPath `//foo`.
Click Element	default: //foo	# Use default strategy with value `//foo`.

Explicit locator strategy

The explicit locator strategy is specified with a prefix using either syntax `strategy:value` or `strategy=value`. The former syntax is preferred because the latter is identical to Robot Framework's named argument syntax and that can cause problems. Spaces around the separator are ignored, so `id:foo`, `id: foo` and `id : foo` are all equivalent.

Locator strategies that are supported by default are listed in the table below. In addition to them, it is possible to register *custom locators*.

Strategy	Match based on	Example
id	Element `id`.	`id:example`
name	`name` attribute.	`name:example`
identifier	Either `id` or `name`.	`identifier:example`
class	Element `class`.	`class:example`
tag	Tag name.	`tag:div`
xpath	XPath expression.	`xpath://div[@id="example"]`
css	CSS selector.	`css:div#example`
dom	DOM expression.	`dom:document.images[5]`
link	Exact text a link has.	`link:The example`
partial link	Partial link text.	`partial link:he ex`
sizzle	Sizzle selector deprecated.	`sizzle:div.example`
jquery	jQuery expression.	`jquery:div.example`
default	Keyword specific default behavior.	`default:example`

See the *Default locator strategy* section below for more information about how the default strategy works. Using the explicit `default` prefix is only necessary if the locator value itself accidentally matches some of the explicit strategies.

Different locator strategies have different pros and cons. Using ids, either explicitly like `id:foo` or by using the *default locator strategy* simply like `foo`, is recommended when possible, because the syntax is simple and locating elements by id is fast for browsers. If an element does not have an id or the id is not stable, other solutions need to be used. If an element has a unique tag name or class, using `tag`, `class` or `css` strategy like `tag:h1`, `class:example` or `css:h1.example` is often an easy solution. In more complex cases using XPath expressions is typically the best approach. They are very powerful but a downside is that they can also get complex.

Examples:

Click Element	id:foo	# Element with id 'foo'.
Click Element	css:div#foo h1	# h1 element under div with id 'foo'.
Click Element	xpath: //div[@id="foo"]//h1	# Same as the above using XPath, not CSS.
Click Element	xpath: //*[contains(text(), "example")]	# Element containing text 'example'.

NOTE:

- The `strategy:value` syntax is only supported by SeleniumLibrary 3.0 and newer.
- Using the `sizzle` strategy or its alias `jquery` requires that the system under test contains the jQuery library.
- Prior to SeleniumLibrary 3.0, table related keywords only supported `xpath`, `css` and `sizzle/jquery` strategies.

SeleniumLibrary

Implicit XPath strategy

If the locator starts with `//` or `(//`, the locator is considered to be an XPath expression. In other words, using `//div` is equivalent to using explicit `xpath://div`.

Examples:

Click Element	//div[@id="foo"]//h1
Click Element	(//div)[2]

The support for the `(//` prefix is new in SeleniumLibrary 3.0.

Chaining locators

It is possible chain multiple locators together as single locator. Each chained locator must start with locator strategy. Chained locators must be separated with single space, two greater than characters and followed with space. It is also possible mix different locator strategies, example css or xpath. Also a list can also be used to specify multiple locators. This is useful, is some part of locator would match as the locator separator but it should not. Or if there is need to existing WebElement as locator.

Although all locators support chaining, some locator strategies do not abey the chaining. This is because some locator strategies use JavaScript to find elements and JavaScript is executed for the whole browser context and not for the element found be the previous locator. Chaining is supported by locator strategies which are based on Selenium API, like *xpath* or *css*, but example chaining is not supported by *sizzle* or `jquery

Examples:

Click Element	css:.bar >> xpath://a	# To find a link which is present after an element with class "bar"

List examples:

${locator_list} =	*Create List*	css:div#div_id	xpath://*[text(), " >> "]
Page Should Contain Element	${locator_list}		
${element} =	Get WebElement	xpath://*[text(), " >> "]	
${locator_list} =	*Create List*	css:div#div_id	${element}
Page Should Contain Element	${locator_list}		

Chaining locators in new in SeleniumLibrary 5.0

Using WebElements

In addition to specifying a locator as a string, it is possible to use Selenium's WebElement objects. This requires first getting a WebElement, for example, by using the *Get WebElement* keyword.

${elem} =	*Get WebElement*	id:example
Click Element	${elem}	

Custom locators

If more complex lookups are required than what is provided through the default locators, custom lookup strategies can be created. Using custom locators is a two part process. First, create a keyword that returns a WebElement that should be acted on:

Custom Locator Strategy	[Arguments]	${browser}	${locator}	${tag}	${constraints}
	${element}=	Execute Javascript	return window.document.getElementById('${locator}');		
	[Return]	${element}			

This keyword is a reimplementation of the basic functionality of the `id` locator where `${browser}` is a reference to a WebDriver instance and `${locator}` is the name of the locator strategy. To use

SeleniumLibrary

this locator, it must first be registered by using the *Add Location Strategy* keyword:

Add Location Strategy	custom	Custom Locator Strategy

The first argument of *Add Location Strategy* specifies the name of the strategy and it must be unique. After registering the strategy, the usage is the same as with other locators:

Click Element	custom:example

See the *Add Location Strategy* keyword for more details.

Browser and Window

There is different conceptual meaning when SeleniumLibrary talks about windows or browsers. This chapter explains those differences.

Browser

When *Open Browser* or *Create WebDriver* keyword is called, it will create a new Selenium WebDriver instance by using the Selenium WebDriver API. In SeleniumLibrary terms, a new browser is created. It is possible to start multiple independent browsers (Selenium Webdriver instances) at the same time, by calling *Open Browser* or *Create WebDriver* multiple times. These browsers are usually independent of each other and do not share data like cookies, sessions or profiles. Typically when the browser starts, it creates a single window which is shown to the user.

Window

Windows are the part of a browser that loads the web site and presents it to the user. All content of the site is the content of the window. Windows are children of a browser. In SeleniumLibrary browser is a synonym for WebDriver instance. One browser may have multiple windows. Windows can appear as tabs, as separate windows or pop-ups with different position and size. Windows belonging to the same browser typically share the sessions detail, like cookies. If there is a need to separate sessions detail, example login with two different users, two browsers (Selenium WebDriver instances) must be created. New windows can be opened example by the application under test or by example *Execute Javascript* keyword:

```
Execute Javascript    window.open()    # Opens a new window with loc
```

The example below opens multiple browsers and windows, to demonstrate how the different keywords can be used to interact with browsers, and windows attached to these browsers.

Structure:

```
BrowserA
           Window 1  (location=https://robotframework.org/)
           Window 2  (location=https://robocon.io/)
           Window 3  (location=https://github.com/robotframework/)

BrowserB
           Window 1  (location=https://github.com/)
```

Example:

Open Browser	https://robotframework.org	${BROWSER}	alias=BrowserA	# BrowserA with first window is opened.
Execute Javascript	window.open()			# In BrowserA second window is opened.

SeleniumLibrary

Switch Window	locator=NEW			# Switched to second window in BrowserA
Go To	https://robocon.io			# Second window navigates to robocon site.
Execute Javascript	window.open()			# In BrowserA third window is opened.
${handle}	*Switch Window*	locator=NEW		# Switched to third window in BrowserA
Go To	https://github.com/robotframework/			# Third windows goes to robot framework github site.
Open Browser	https://github.com	${BROWSER}	alias=BrowserB	# BrowserB with first windows is opened.
${location}	*Get Location*			# ${location} is: https://www.github.com
Switch Window	${handle}	browser=BrowserA		# BrowserA second windows is selected.
${location}	*Get Location*			# ${location} = https://robocon.io/
@{locations 1}	*Get Locations*			# By default, lists locations under the currectly active browser (BrowserA).
@{locations 2}	*Get Locations*	browser=ALL		# By using browser=ALL argument keyword list all locations from all browsers.

The above example, @{locations 1} contains the following items: https://robotframework.org/, https://robocon.io/ and https://github.com/robotframework/'. The @{locations 2} contains the following items: https://robotframework.org/, https://robocon.io/, https://github.com/robotframework/' and 'https://github.com/.

Timeouts, waits, and delays

This section discusses different ways how to wait for elements to appear on web pages and to slow down execution speed otherwise. It also explains the *time format* that can be used when setting various timeouts, waits, and delays.

Timeout

SeleniumLibrary contains various keywords that have an optional `timeout` argument that specifies how long these keywords should wait for certain events or actions. These keywords include, for example, `Wait ...` keywords and keywords related to alerts. Additionally *Execute Async Javascript*. Although it does not have `timeout`, argument, uses a timeout to define how long asynchronous JavaScript can run.

The default timeout these keywords use can be set globally either by using the *Set Selenium Timeout* keyword or with the `timeout` argument when *importing* the library. See *time format* below for supported timeout syntax.

Implicit wait

Implicit wait specifies the maximum time how long Selenium waits when searching for elements. It can be set by using the *Set Selenium Implicit Wait* keyword or with the `implicit_wait` argument when *importing* the library. See Selenium documentation for more information about this functionality.

See *time format* below for supported syntax.

SeleniumLibrary

Selenium speed

Selenium execution speed can be slowed down globally by using *Set Selenium speed* keyword. This functionality is designed to be used for demonstrating or debugging purposes. Using it to make sure that elements appear on a page is not a good idea. The above-explained timeouts and waits should be used instead.

See *time format* below for supported syntax.

Time format

All timeouts and waits can be given as numbers considered seconds (e.g. `0.5` or `42`) or in Robot Framework's time syntax (e.g. `1.5 seconds` or `1 min 30 s`). For more information about the time syntax see the Robot Framework User Guide.

Run-on-failure functionality

SeleniumLibrary has a handy feature that it can automatically execute a keyword if any of its own keywords fails. By default, it uses the *Capture Page Screenshot* keyword, but this can be changed either by using the *Register Keyword To Run On Failure* keyword or with the `run_on_failure` argument when *importing* the library. It is possible to use any keyword from any imported library or resource file.

The run-on-failure functionality can be disabled by using a special value `NOTHING` or anything considered false (see *Boolean arguments*) such as `NONE`.

Boolean arguments

Starting from 5.0 SeleniumLibrary relies on Robot Framework to perform the boolean conversion based on keyword arguments type hint. More details in Robot Framework user guide

Please note SeleniumLibrary 3 and 4 did have own custom methods to covert arguments to boolean values.

EventFiringWebDriver

The SeleniumLibrary offers support for EventFiringWebDriver. See the Selenium and SeleniumLibrary EventFiringWebDriver support documentation for further details.

EventFiringWebDriver is new in SeleniumLibrary 4.0

Thread support

SeleniumLibrary is not thread-safe. This is mainly due because the underlying Selenium tool is not thread-safe within one browser/driver instance. Because of the limitation in the Selenium side, the keywords or the API provided by the SeleniumLibrary is not thread-safe.

SeleniumLibrary

Plugins

SeleniumLibrary offers plugins as a way to modify and add library keywords and modify some of the internal functionality without creating a new library or hacking the source code. See plugin API documentation for further details.

Plugin API is new SeleniumLibrary 4.0

Importing

Arguments

```
timeout=0:00:05
implicit_wait=0:00:00
run_on_failure=Capture Page Screenshot
screenshot_root_directory: str = None
plugins: str = None
event_firing_webdriver: str = None
```

Documentation

SeleniumLibrary can be imported with several optional arguments.

- `timeout`: Default value for *timeouts* used with `Wait ...` keywords.
- `implicit_wait`: Default value for *implicit wait* used when locating elements.
- `run_on_failure`: Default action for the *run-on-failure functionality*.
- `screenshot_root_directory`: Path to folder where possible screenshots are created or EMBED. See *Set Screenshot Directory* keyword for further details about EMBED. If not given, the directory where the log file is written is used.
- `plugins`: Allows extending the SeleniumLibrary with external Python classes.
- `event_firing_webdriver`: Class for wrapping Selenium with EventFiringWebDriver

Keywords

Add Cookie

Arguments

```
name: str
value: str
path: str = None
domain: str = None
secure: bool = None
expiry: str = None
```

Documentation

Adds a cookie to your current session.

`name` and `value` are required, `path`, `domain`, `secure` and `expiry` are optional. Expiry supports the same formats as the DateTime library or an epoch timestamp.

Example:

Add Cookie	foo	bar		
Add Cookie	foo	bar	domain=example.com	
Add Cookie	foo	bar	expiry=2027-09-28 16:21:35	# Expiry as timestamp.
Add Cookie	foo	bar	expiry=1822137695	# Expiry as epoch seconds.

Prior to SeleniumLibrary 3.0 setting expiry did not work.

SeleniumLibrary

Add Location Strategy

Arguments

```
strategy_name: str
strategy_keyword: str
persist: bool = False
```

Documentation

Adds a custom location strategy.

See *Custom locators* for information on how to create and use custom strategies. *Remove Location Strategy* can be used to remove a registered strategy.

Location strategies are automatically removed after leaving the current scope by default. Setting `persist` to a true value (see *Boolean arguments*) will cause the location strategy to stay registered throughout the life of the test.

Alert Should Be Present

Arguments

```
text: str =
action: str = ACCEPT
timeout: timedelta = None
```

Documentation

Verifies that an alert is present and by default, accepts it.

Fails if no alert is present. If `text` is a non-empty string, then it is used to verify alert's message. The alert is accepted by default, but that behavior can be controlled by using the `action` argument same way as with *Handle Alert*.

`timeout` specifies how long to wait for the alert to appear. If it is not given, the global default *timeout* is used instead.

`action` and `timeout` arguments are new in SeleniumLibrary 3.0. In earlier versions, the alert was always accepted and a timeout was hardcoded to one second.

Alert Should Not Be Present

Arguments

```
action: str = ACCEPT
timeout: timedelta = None
```

Documentation

Verifies that no alert is present.

If the alert actually exists, the `action` argument determines how it should be handled. By default, the alert is accepted, but it can be also dismissed or left open the same way as with the *Handle Alert* keyword.

`timeout` specifies how long to wait for the alert to appear. By default, is not waited for the alert at all, but a custom time can be given if alert may be delayed. See the *time format* section for information about the syntax.

New in SeleniumLibrary 3.0.

Assign Id To Element

Arguments

```
locator: str
id: str
```

Documentation

Assigns a temporary `id` to the element specified by `locator`.

This is mainly useful if the locator is complicated and/or slow XPath expression and it is needed multiple times. Identifier expires when the page is reloaded.

See the *Locating elements* section for details about the locator syntax.

Example:

Assign ID to Element	//ul[@class='example' and ./li[contains(., 'Stuff')]]	my id
Page Should Contain Element	my id	

Capture Element Screenshot

Arguments

```
locator: str
filename: str = selenium-element-screenshot-{index}.png
```

Documentation

Captures a screenshot from the element identified by `locator` and embeds it into log file.

See *Capture Page Screenshot* for details about `filename` argument. See the *Locating elements* section for details about the locator syntax.

An absolute path to the created element screenshot is returned.

Support for capturing the screenshot from an element has limited support among browser vendors. Please check the browser vendor driver documentation does the browser support capturing a screenshot from an element.

New in SeleniumLibrary 3.3. Support for EMBED is new in SeleniumLibrary 4.2.

Examples:

Capture Element Screenshot	id:image_id	
Capture Element Screenshot	id:image_id	${OUTPUTDIR}/id_image_id-1.png
Capture Element Screenshot	id:image_id	EMBED

SeleniumLibrary

Capture Page Screenshot

Arguments

`filename: str = selenium-screenshot-{index}.png`

Documentation

Takes a screenshot of the current page and embeds it into a log file.

`filename` argument specifies the name of the file to write the screenshot into. The directory where screenshots are saved can be set when *importing* the library or by using the *Set Screenshot Directory* keyword. If the directory is not configured, screenshots are saved to the same directory where Robot Framework's log file is written.

If `filename` equals to EMBED (case insensitive), then screenshot is embedded as Base64 image to the log.html. In this case file is not created in the filesystem.

Starting from SeleniumLibrary 1.8, if `filename` contains marker `{index}`, it will be automatically replaced with an unique running index, preventing files to be overwritten. Indices start from 1, and how they are represented can be customized using Python's format string syntax.

An absolute path to the created screenshot file is returned or if `filename` equals to EMBED, word *EMBED* is returned.

Support for EMBED is new in SeleniumLibrary 4.2

Examples:

Capture Page Screenshot	
File Should Exist	${OUTPUTDIR}/selenium-screenshot-1.png
${path} =	*Capture Page Screenshot*
File Should Exist	${OUTPUTDIR}/selenium-screenshot-2.png
File Should Exist	${path}
Capture Page Screenshot	custom_name.png
File Should Exist	${OUTPUTDIR}/custom_name.png
Capture Page Screenshot	custom_with_index_{index}.png
File Should Exist	${OUTPUTDIR}/custom_with_index_1.png
Capture Page Screenshot	formatted_index_{index:03}.png
File Should Exist	${OUTPUTDIR}/formatted_index_001.png
Capture Page Screenshot	EMBED
File Should Not Exist	EMBED

Checkbox Should Be Selected

Arguments

`locator: str`

Documentation

Verifies checkbox `locator` is selected/checked.

See the *Locating elements* section for details about the locator syntax.

Checkbox Should Not Be Selected

Arguments

```
locator: str
```

Documentation

Verifies checkbox `locator` is not selected/checked.

See the *Locating elements* section for details about the locator syntax.

Choose File

Arguments

```
locator: str
file_path: str
```

Documentation

Inputs the `file_path` into the file input field `locator`.

This keyword is most often used to input files into upload forms. The keyword does not check `file_path` is the file or folder available on the machine where tests are executed. If the `file_path` points at a file and when using Selenium Grid, Selenium will magically, transfer the file from the machine where the tests are executed to the Selenium Grid node where the browser is running. Then Selenium will send the file path, from the nodes file system, to the browser.

That `file_path` is not checked, is new in SeleniumLibrary 4.0.

Example:

Choose File	my_upload_field	${CURDIR}/trades.csv

Clear Element Text

Arguments

```
locator: str
```

Documentation

Clears the value of the text-input-element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

Click Button

Arguments

```
locator: str
modifier: typing.Union[str, bool] = False
```

Documentation

Clicks the button identified by `locator`.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, buttons are searched using `id`, `name`, and `value`.

See the *Click Element* keyword for details about the `modifier` argument.

The `modifier` argument is new in SeleniumLibrary 3.3

Click Element

Arguments

```
locator: str
modifier: typing.Union[str, bool] = False
action_chain: bool = False
```

Documentation

Click the element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

The `modifier` argument can be used to pass Selenium Keys when clicking the element. The + can be used as a separator for different Selenium Keys. The *CTRL* is internally translated to the *CONTROL* key. The `modifier` is space and case insensitive, example "alt" and " aLt " are supported formats to ALT key . If `modifier` does not match to Selenium Keys, keyword fails.

If `action_chain` argument is true, see *Boolean arguments* for more details on how to set boolean argument, then keyword uses ActionChain based click instead of the <web_element>.click() function. If both `action_chain` and `modifier` are defined, the click will be performed using `modifier` and `action_chain` will be ignored.

Example:

Click Element	id:button		# Would click element without any modifiers.
Click Element	id:button	CTRL	# Would click element with CTLR key pressed down.
Click Element	id:button	CTRL+ALT	# Would click element with CTLR and ALT keys pressed down.
Click Element	id:button	action_chain=True	# Clicks the button using an Selenium ActionChains

The `modifier` argument is new in SeleniumLibrary 3.2 The `action_chain` argument is new in SeleniumLibrary 4.1

Click Element At Coordinates

Arguments

```
locator
xoffset
yoffset
```

Documentation

Click the element `locator` at `xoffset/yoffset`.

The Cursor is moved and the center of the element and x/y coordinates are calculated from that point.

See the *Locating elements* section for details about the locator syntax.

Click Image

Arguments

```
locator: str
modifier: typing.Union[str, bool] = False
```

Documentation

Clicks an image identified by `locator`.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, images are searched using `id`, `name`, `src` and `alt`.

See the *Click Element* keyword for details about the `modifier` argument.

The `modifier` argument is new in SeleniumLibrary 3.3

Click Link

Arguments

```
locator: str
modifier: typing.Union[str, bool] = False
```

Documentation

Clicks a link identified by `locator`.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, links are searched using `id`, `name`, `href` and the link text.

See the *Click Element* keyword for details about the `modifier` argument.

The `modifier` argument is new in SeleniumLibrary 3.3

Close All Browsers

Documentation

Closes all open browsers and resets the browser cache.

After this keyword, new indexes returned from *Open Browser* keyword are reset to 1.

This keyword should be used in test or suite teardown to make sure all browsers are closed.

Close Browser

SeleniumLibrary

Documentation

Closes the current browser.

Close Window

Documentation

Closes currently opened and selected browser window/tab.

Cover Element

Arguments

```
locator: str
```

Documentation

Will cover elements identified by `locator` with a blue div without breaking page layout.

See the *Locating elements* section for details about the locator syntax.

New in SeleniumLibrary 3.3.0

Example: |*Cover Element* | css:div#container |

Create Webdriver

Arguments

```
driver_name: str
alias: str = None
kwargs={}
**init_kwargs
```

Documentation

Creates an instance of Selenium WebDriver.

Like *Open Browser*, but allows passing arguments to the created WebDriver instance directly. This keyword should only be used if the functionality provided by *Open Browser* is not adequate.

`driver_name` must be a WebDriver implementation name like Firefox, Chrome, Ie, Opera, Safari, PhantomJS, or Remote.

The initialized WebDriver can be configured either with a Python dictionary `kwargs` or by using keyword arguments `**init_kwargs`. These arguments are passed directly to WebDriver without any processing. See Selenium API documentation for details about the supported arguments.

Examples:

# Use proxy with Firefox			
${proxy}=	*Evaluate*	selenium.webdriver.Proxy()	modules=selenium, selenium.webdriver

SeleniumLibrary

${proxy.http_proxy}=	*Set Variable*	localhost:8888	
Create Webdriver	Firefox	proxy=${proxy}	
# Use proxy with PhantomJS			
${service args}=	*Create List*	--proxy=192.168.132.104:8888	
Create Webdriver	PhantomJS	service_args=${service args}	

Returns the index of this browser instance which can be used later to switch back to it. Index starts from 1 and is reset back to it when *Close All Browsers* keyword is used. See *Switch Browser* for an example.

Current Frame Should Contain

Arguments

```
text: str
loglevel: str = TRACE
```

Documentation

Verifies that the current frame contains `text`.

See *Page Should Contain* for an explanation about the `loglevel` argument.

Prior to SeleniumLibrary 3.0 this keyword was named *Current Frame Contains*.

Current Frame Should Not Contain

Arguments

```
text: str
loglevel: str = TRACE
```

Documentation

Verifies that the current frame does not contain `text`.

See *Page Should Contain* for an explanation about the `loglevel` argument.

Delete All Cookies

Documentation

Deletes all cookies.

Delete Cookie

Arguments

```
name
```

Documentation

SeleniumLibrary

Deletes the cookie matching `name`.

If the cookie is not found, nothing happens.

Double Click Element

Arguments

```
locator: str
```

Documentation

Double clicks the element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

Drag And Drop

Arguments

```
locator: str
target: str
```

Documentation

Drags the element identified by `locator` into the `target` element.

The `locator` argument is the locator of the dragged element and the `target` is the locator of the target. See the *Locating elements* section for details about the locator syntax.

Example:

Drag And Drop	css:div#element	css:div.target

Drag And Drop By Offset

Arguments

```
locator: str
xoffset: int
yoffset: int
```

Documentation

Drags the element identified with `locator` by `xoffset/yoffset`.

See the *Locating elements* section for details about the locator syntax.

The element will be moved by `xoffset` and `yoffset`, each of which is a negative or positive number specifying the offset.

Example:

Drag And Drop By Offset	myElem	50	-35	# Move myElem 50px right and 35px down

SeleniumLibrary

Element Attribute Value Should Be

Arguments

```
locator: str
attribute: str
expected: str
message: str = None
```

Documentation

Verifies element identified by `locator` contains expected attribute value.

See the *Locating elements* section for details about the locator syntax.

Example: *Element Attribute Value Should Be* | css:img | href | value

New in SeleniumLibrary 3.2.

Element Should Be Disabled

Arguments

```
locator: str
```

Documentation

Verifies that element identified by `locator` is disabled.

This keyword considers also elements that are read-only to be disabled.

See the *Locating elements* section for details about the locator syntax.

Element Should Be Enabled

Arguments

```
locator: str
```

Documentation

Verifies that element identified by `locator` is enabled.

This keyword considers also elements that are read-only to be disabled.

See the *Locating elements* section for details about the locator syntax.

Element Should Be Focused

Arguments

SeleniumLibrary

```
locator: str
```

Documentation

Verifies that element identified by `locator` is focused.

See the *Locating elements* section for details about the locator syntax.

New in SeleniumLibrary 3.0.

Element Should Be Visible

Arguments

```
locator: str
message: str = None
```

Documentation

Verifies that the element identified by `locator` is visible.

Herein, visible means that the element is logically visible, not optically visible in the current browser viewport. For example, an element that carries `display:none` is not logically visible, so using this keyword on that element would fail.

See the *Locating elements* section for details about the locator syntax.

The `message` argument can be used to override the default error message.

Element Should Contain

Arguments

```
locator: str
expected: str
message: str = None
ignore_case: bool = False
```

Documentation

Verifies that element `locator` contains text `expected`.

See the *Locating elements* section for details about the locator syntax.

The `message` argument can be used to override the default error message.

The `ignore_case` argument can be set to True to compare case insensitive, default is False. New in SeleniumLibrary 3.1.

`ignore_case` argument is new in SeleniumLibrary 3.1.

Use *Element Text Should Be* if you want to match the exact text, not a substring.

Element Should Not Be Visible

Arguments

```
locator: str
message: str = None
```

Documentation

Verifies that the element identified by `locator` is NOT visible.

Passes if the element does not exists. See *Element Should Be Visible* for more information about visibility and supported arguments.

Element Should Not Contain

Arguments

```
locator: str
expected: str
message: str = None
ignore_case: bool = False
```

Documentation

Verifies that element `locator` does not contain text `expected`.

See the *Locating elements* section for details about the locator syntax.

The `message` argument can be used to override the default error message.

The `ignore_case` argument can be set to True to compare case insensitive, default is False.

`ignore_case` argument new in SeleniumLibrary 3.1.

Element Text Should Be

Arguments

```
locator: str
expected: str
message: str = None
ignore_case: bool = False
```

Documentation

Verifies that element `locator` contains exact the text `expected`.

See the *Locating elements* section for details about the locator syntax.

The `message` argument can be used to override the default error message.

The `ignore_case` argument can be set to True to compare case insensitive, default is False.

`ignore_case` argument is new in SeleniumLibrary 3.1.

Use *Element Should Contain* if a substring match is desired.

SeleniumLibrary

Element Text Should Not Be

Arguments

```
locator: str
not_expected: str


message: str = None
ignore_case: bool = False
```

Documentation

Verifies that element `locator` does not contain exact the text `not_expected`.

See the *Locating elements* section for details about the locator syntax.

The `message` argument can be used to override the default error message.

The `ignore_case` argument can be set to True to compare case insensitive, default is False.

New in SeleniumLibrary 3.1.1

Execute Async Javascript

Arguments

```
*code: str
```

Documentation

Executes asynchronous JavaScript code with possible arguments.

Similar to *Execute Javascript* except that scripts executed with this keyword must explicitly signal they are finished by invoking the provided callback. This callback is always injected into the executed function as the last argument.

Scripts must complete within the script timeout or this keyword will fail. See the *Timeout* section for more information.

Starting from SeleniumLibrary 3.2 it is possible to provide JavaScript arguments as part of `code` argument. See *Execute Javascript* for more details.

Examples:

Execute Async JavaScript	var callback = arguments[arguments.length - 1]; window.setTimeout(callback, 2000);	
Execute Async JavaScript	${CURDIR}/async_js_to_execute.js	
${result} =	*Execute Async JavaScript*	
...	var callback = arguments[arguments.length - 1];	
...	function answer(){callback("text");};	
...	window.setTimeout(answer, 2000);	
Should Be Equal	${result}	text

Execute Javascript

Arguments

SeleniumLibrary

`*code: str`

Documentation

Executes the given JavaScript code with possible arguments.

`code` may be divided into multiple cells in the test data and `code` may contain multiple lines of code and arguments. In that case, the JavaScript code parts are concatenated together without adding spaces and optional arguments are separated from `code`.

If `code` is a path to an existing file, the JavaScript to execute will be read from that file. Forward slashes work as a path separator on all operating systems.

The JavaScript executes in the context of the currently selected frame or window as the body of an anonymous function. Use `window` to refer to the window of your application and `document` to refer to the document object of the current frame or window, e.g. `document.getElementById('example')`.

This keyword returns whatever the executed JavaScript code returns. Return values are converted to the appropriate Python types.

Starting from SeleniumLibrary 3.2 it is possible to provide JavaScript arguments as part of `code` argument. The JavaScript code and arguments must be separated with *JAVASCRIPT* and *ARGUMENTS* markers and must be used exactly with this format. If the Javascript code is first, then the *JAVASCRIPT* marker is optional. The order of *JAVASCRIPT* and *ARGUMENTS* markers can be swapped, but if *ARGUMENTS* is the first marker, then *JAVASCRIPT* marker is mandatory. It is only allowed to use *JAVASCRIPT* and *ARGUMENTS* markers only one time in the `code` argument.

Examples:

Execute JavaScript	window.myFunc('arg1', 'arg2')			
Execute JavaScript	${CURDIR}/js_to_execute.js			
Execute JavaScript	alert(arguments[0]);	ARGUMENTS	123	
Execute JavaScript	ARGUMENTS	123	JAVASCRIPT	alert(arguments[0]);

Frame Should Contain

Arguments

```
locator: str
text: str
loglevel: str = TRACE
```

Documentation

Verifies that frame identified by `locator` contains `text`.

See the *Locating elements* section for details about the locator syntax.

See *Page Should Contain* for an explanation about the `loglevel` argument.

Get All Links

Documentation

Returns a list containing ids of all links found in current page.

If a link has no id, an empty string will be in the list instead.

SeleniumLibrary

Get Browser Aliases

Documentation

Returns aliases of all active browser that has an alias as NormalizedDict. The dictionary contains the aliases as keys and the index as value. This can be accessed as dictionary `${aliases.key}` or as list `@{aliases}[0]`.

Example:

Open Browser	https://example.com	alias=BrowserA	
Open Browser	https://example.com	alias=BrowserB	
&{aliases}	*Get Browser Aliases*		# &{aliases} = { BrowserA=1|BrowserB=2 }
Log	${aliases.BrowserA}		# logs `1`
FOR	${alias}	IN	@{aliases}
	Log	${alias}	# logs `BrowserA` and `BrowserB`
END			

See *Switch Browser* for more information and examples.

New in SeleniumLibrary 4.0

Get Browser Ids

Documentation

Returns index of all active browser as list.

Example:

@{browser_ids}=	Get Browser Ids		
FOR	${id}	IN	@{browser_ids}
	@{window_titles}=	Get Window Titles	browser=${id}
	Log	Browser ${id} has these windows: ${window_titles}	
END			

See *Switch Browser* for more information and examples.

New in SeleniumLibrary 4.0

Get Cookie

Arguments

`name: str`

Documentation

Returns information of cookie with `name` as an object.

If no cookie is found with `name`, keyword fails. The cookie object contains details about the cookie. Attributes available in the object are documented in the table below.

Attribute	Explanation
name	The name of a cookie.

SeleniumLibrary

value	Value of the cookie.
path	Indicates a URL path, for example `/`.
domain	The domain, the cookie is visible to.
secure	When true, the cookie is only used with HTTPS connections.
httpOnly	When true, the cookie is not accessible via JavaScript.
expiry	Python datetime object indicating when the cookie expires.
extra	Possible attributes outside of the WebDriver specification

See the WebDriver specification for details about the cookie information. Notice that `expiry` is specified as a datetime object, not as seconds since Unix Epoch like WebDriver natively does.

In some cases, example when running a browser in the cloud, it is possible that the cookie contains other attributes than is defined in the WebDriver specification. These other

attributes are available in an `extra` attribute in the cookie object and it contains a dictionary of the other attributes. The `extra` attribute is new in SeleniumLibrary 4.0.

Example:

Add Cookie	foo	bar
${cookie} =	*Get Cookie*	foo
Should Be Equal	${cookie.name}	foo
Should Be Equal	${cookie.value}	bar
Should Be True	${cookie.expiry.year} > 2017	

New in SeleniumLibrary 3.0.

Get Cookies

Arguments

```
as_dict: bool = False
```

Documentation

Returns all cookies of the current page.

If `as_dict` argument evaluates as false, see *Boolean arguments* for more details, then cookie information is returned as a single string in format `name1=value1; name2=value2; name3=value3`. When `as_dict` argument evaluates as true, cookie information is returned as Robot Framework dictionary format. The string format can be used, for example, for logging purposes or in headers when sending HTTP requests. The dictionary format is helpful when the result can be passed to requests library's Create Session keyword's optional cookies parameter.

The ` *as_dict*` argument is new in SeleniumLibrary 3.3

Get Element Attribute

Arguments

```
locator: str
attribute: str
```

Documentation

Returns the value of `attribute` from the element `locator`.

See the *Locating elements* section for details about the locator syntax.

Example:

${id}=	*Get Element Attribute*	css:h1	id

Passing attribute name as part of the `locator` was removed in SeleniumLibrary 3.2. The explicit `attribute` argument should be used instead.

Get Element Count

Arguments

`locator: str`

Documentation

Returns the number of elements matching `locator`.

If you wish to assert the number of matching elements, use *Page Should Contain Element* with `limit` argument. Keyword will always return an integer.

Example:

${count} =	*Get Element Count*	name:div_name
Should Be True	${count} > 2	

New in SeleniumLibrary 3.0.

Get Element Size

Arguments

`locator: str`

Documentation

Returns width and height of the element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

Both width and height are returned as integers.

Example:

${width}	${height} =	*Get Element Size*	css:div#container

Get Horizontal Position

Arguments

`locator: str`

Documentation

Returns the horizontal position of the element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

The position is returned in pixels off the left side of the page, as an integer.

See also *Get Vertical Position*.

Get List Items

Arguments

```
locator: str
values: bool = False
```

Documentation

Returns all labels or values of selection list `locator`.

See the *Locating elements* section for details about the locator syntax.

Returns visible labels by default, but values can be returned by setting the `values` argument to a true value (see *Boolean arguments*).

Example:

${labels} =	*Get List Items*	mylist	
${values} =	*Get List Items*	css:#example select	values=True

Support to return values is new in SeleniumLibrary 3.0.

Get Location

Documentation

Returns the current browser window URL.

Get Locations

Arguments

```
browser: str = CURRENT
```

Documentation

Returns and logs URLs of all windows of the selected browser.

Browser Scope:

The `browser` argument specifies the browser that shall return its windows information.

- `browser` can be `index_or_alias` like in *Switch Browser*.
- If `browser` is `CURRENT` (default, case-insensitive) the currently active browser is selected.
- If `browser` is `ALL` (case-insensitive) the window information of all windows of all opened browsers are returned.

Get Selected List Label

Arguments

`locator: str`

Documentation

Returns the label of selected option from selection list `locator`.

If there are multiple selected options, the label of the first option is returned.

See the *Locating elements* section for details about the locator syntax.

Get Selected List Labels

Arguments

`locator: str`

Documentation

Returns labels of selected options from selection list `locator`.

Starting from SeleniumLibrary 3.0, returns an empty list if there are no selections. In earlier versions, this caused an error.

See the *Locating elements* section for details about the locator syntax.

Get Selected List Value

Arguments

`locator: str`

Documentation

Returns the value of selected option from selection list `locator`.

If there are multiple selected options, the value of the first option is returned.

See the *Locating elements* section for details about the locator syntax.

Get Selected List Values

Arguments

`locator: str`

Documentation

Returns values of selected options from selection list `locator`.

Returns values of selected options from selection list `locator`.

Starting from SeleniumLibrary 3.0, returns an empty list if there are no selections. In earlier versions, this caused an error.

See the *Locating elements* section for details about the locator syntax.

Get Selenium Implicit Wait

Documentation

Gets the implicit wait value used by Selenium.

The value is returned as a human-readable string like `1 second`.

See the *Implicit wait* section above for more information.

Get Selenium Speed

Documentation

Gets the delay that is waited after each Selenium command.

The value is returned as a human-readable string like `1 second`.

See the *Selenium Speed* section above for more information.

Get Selenium Timeout

Documentation

Gets the timeout that is used by various keywords.

The value is returned as a human-readable string like `1 second`.

See the *Timeout* section above for more information.

Get Session Id

Documentation

Returns the currently active browser session id.

New in SeleniumLibrary 3.2

Get Source

Documentation

Returns the entire HTML source of the current page or frame.

Get Table Cell

Arguments

```
locator: str
row: int
column: int
loglevel: str = TRACE
```

Documentation

Returns contents of a table cell.

The table is located using the `locator` argument and its cell found using `row` and `column`. See the *Locating elements* section for details about the locator syntax.

Both row and column indexes start from 1, and header and footer rows are included in the count. It is possible to refer to rows and columns from the end by using negative indexes so that -1 is the last row/column, -2 is the second last, and so on.

All `<th>` and `<td>` elements anywhere in the table are considered to be cells.

See *Page Should Contain* for an explanation about the `loglevel` argument.

Get Text

Arguments

```
locator: str
```

Documentation

Returns the text value of the element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

Get Title

Documentation

Returns the title of the current page.

Get Value

Arguments

```
locator: str
```

Documentation

Returns the value attribute of the element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

Get Vertical Position

Arguments

`locator: str`

Documentation

Returns the vertical position of the element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

The position is returned in pixels off the top of the page, as an integer.

See also *Get Horizontal Position*.

Get WebElement

Arguments

`locator: str`

Documentation

Returns the first WebElement matching the given `locator`.

See the *Locating elements* section for details about the locator syntax.

Get WebElements

Arguments

`locator: str`

Documentation

Returns a list of WebElement objects matching the `locator`.

See the *Locating elements* section for details about the locator syntax.

Starting from SeleniumLibrary 3.0, the keyword returns an empty list if there are no matching elements. In previous releases, the keyword failed in this case.

Get Window Handles

Arguments

`browser: str = CURRENT`

SeleniumLibrary

Documentation

Returns all child window handles of the selected browser as a list.

Can be used as a list of windows to exclude with *Select Window*.

How to select the `browser` scope of this keyword, see *Get Locations*.

Prior to SeleniumLibrary 3.0, this keyword was named *List Windows*.

Get Window Identifiers

Arguments

`browser: str = CURRENT`

Documentation

Returns and logs id attributes of all windows of the selected browser.

How to select the `browser` scope of this keyword, see *Get Locations*.

Get Window Names

Arguments

`browser: str = CURRENT`

Documentation

Returns and logs names of all windows of the selected browser.

How to select the `browser` scope of this keyword, see *Get Locations*.

Get Window Position

Documentation

Returns current window position.

The position is relative to the top left corner of the screen. Returned values are integers. See also *Set Window Position*.

Example:

${x}	${y}=	*Get Window Position*

Get Window Size

Arguments

`inner: bool = False`

```
inner: bool = False
```

Documentation

Returns current window width and height as integers.

See also *Set Window Size*.

If `inner` parameter is set to True, keyword returns HTML DOM window.innerWidth and window.innerHeight properties. See *Boolean arguments* for more details on how to set boolean arguments. The `inner` is new in SeleniumLibrary 4.0.

Example:

${width}	${height}=	*Get Window Size*	
${width}	${height}=	*Get Window Size*	True

Get Window Titles

Arguments

```
browser: str = CURRENT
```

Documentation

Returns and logs titles of all windows of the selected browser.

How to select the `browser` scope of this keyword, see *Get Locations*.

Go Back

Documentation

Simulates the user clicking the back button on their browser.

Go To

Arguments

```
url
```

Documentation

Navigates the current browser window to the provided `url`.

Handle Alert

Arguments

```
action: str = ACCEPT
timeout: timedelta = None
```

Documentation

Handles the current alert and returns its message.

By default, the alert is accepted, but this can be controlled with the `action` argument that supports the following case-insensitive values:

- `ACCEPT`: Accept the alert i.e. press `Ok`. Default.
- `DISMISS`: Dismiss the alert i.e. press `Cancel`.
- `LEAVE`: Leave the alert open.

The `timeout` argument specifies how long to wait for the alert to appear. If it is not given, the global default *timeout* is used instead.

Examples:

Handle Alert			# Accept alert.
Handle Alert	action=DISMISS		# Dismiss alert.
Handle Alert	timeout=10 s		# Use custom timeout and accept alert.
Handle Alert	DISMISS	1 min	# Use custom timeout and dismiss alert.
${message} =	Handle Alert		# Accept alert and get its message.
${message} =	Handle Alert	LEAVE	# Leave alert open and get its message.

New in SeleniumLibrary 3.0.

Input Password

Arguments

```
locator: str
password: str
clear: bool = True
```

Documentation

Types the given password into the text field identified by `locator`.

See the *Locating elements* section for details about the locator syntax. See *Input Text* for `clear` argument details.

Difference compared to *Input Text* is that this keyword does not log the given password on the INFO level. Notice that if you use the keyword like

Input Password	password_field	password

the password is shown as a normal keyword argument. A way to avoid that is using variables like

Input Password	password_field	${PASSWORD}

Please notice that Robot Framework logs all arguments using the TRACE level and tests must not be executed using level below DEBUG if the password should not be logged in any format.

The *clear* argument is new in SeleniumLibrary 4.0. Hiding password logging from Selenium logs is new in SeleniumLibrary 4.2.

Input Text

Arguments

```
locator: str
text: str
clear: bool = True
```

SeleniumLibrary

```
clear: bool = True
```

Documentation

Types the given `text` into the text field identified by `locator`.

When `clear` is true, the input element is cleared before the text is typed into the element. When false, the previous text is not cleared from the element. Use *Input Password* if you do not want the given `text` to be logged.

If Selenium Grid is used and the `text` argument points to a file in the file system, then this keyword prevents the Selenium to transfer the file to the Selenium Grid hub. Instead, this keyword will send the `text` string as is to the element. If a file should be transferred to the hub and upload should be performed, please use *Choose File* keyword.

See the *Locating elements* section for details about the locator syntax. See the *Boolean arguments* section how Boolean values are handled.

Disabling the file upload the Selenium Grid node and the *clear* argument are new in SeleniumLibrary 4.0

Input Text Into Alert

Arguments

```
text: str
action: str = ACCEPT
timeout: timedelta = None
```

Documentation

Types the given `text` into an input field in an alert.

The alert is accepted by default, but that behavior can be controlled by using the `action` argument same way as with *Handle Alert*.

`timeout` specifies how long to wait for the alert to appear. If it is not given, the global default *timeout* is used instead.

New in SeleniumLibrary 3.0.

List Selection Should Be

Arguments

```
locator: str
*expected: str
```

Documentation

Verifies selection list `locator` has `expected` options selected.

It is possible to give expected options both as visible labels and as values. Starting from SeleniumLibrary 3.0, mixing labels and values is not possible. Order of the selected options is not validated.

If no expected options are given, validates that the list has no selections. A more explicit alternative is using *List Should Have No Selections*.

SeleniumLibrary

See the *Locating elements* section for details about the locator syntax.

Examples:

List Selection Should Be	gender	Female	
List Selection Should Be	interests	Test Automation	Python

List Should Have No Selections

Arguments

```
locator: str
```

Documentation

Verifies selection list `locator` has no options selected.

See the *Locating elements* section for details about the locator syntax.

Location Should Be

Arguments

```
url: str
message: str = None
```

Documentation

Verifies that the current URL is exactly `url`.

The `url` argument contains the exact url that should exist in browser.

The `message` argument can be used to override the default error message.

`message` argument is new in SeleniumLibrary 3.2.0.

Location Should Contain

Arguments

```
expected: str
message: str = None
```

Documentation

Verifies that the current URL contains `expected`.

The `expected` argument contains the expected value in url.

The `message` argument can be used to override the default error message.

`message` argument is new in SeleniumLibrary 3.2.0.

SeleniumLibrary

Log Location

Documentation

Logs and returns the current browser window URL.

Log Source

Arguments

`loglevel: str = INFO`

Documentation

Logs and returns the HTML source of the current page or frame.

The `loglevel` argument defines the used log level. Valid log levels are `WARN`, `INFO` (default), `DEBUG`, `TRACE` and `NONE` (no logging).

Log Title

Documentation

Logs and returns the title of the current page.

Maximize Browser Window

Documentation

Maximizes current browser window.

Mouse Down

Arguments

`locator: str`

Documentation

Simulates pressing the left mouse button on the element `locator`.

See the *Locating elements* section for details about the locator syntax.

The element is pressed without releasing the mouse button.

See also the more specific keywords *Mouse Down On Image* and *Mouse Down On Link*.

Mouse Down On Image

SeleniumLibrary

Arguments

`locator: str`

Documentation

Simulates a mouse down event on an image identified by `locator`.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, images are searched using `id`, `name`, `src` and `alt`.

Mouse Down On Link

Arguments

`locator: str`

Documentation

Simulates a mouse down event on a link identified by `locator`.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, links are searched using `id`, `name`, `href` and the link text.

Mouse Out

Arguments

`locator: str`

Documentation

Simulates moving the mouse away from the element `locator`.

See the *Locating elements* section for details about the locator syntax.

Mouse Over

Arguments

`locator: str`

Documentation

Simulates hovering the mouse over the element `locator`.

See the *Locating elements* section for details about the locator syntax.

Mouse Up

SeleniumLibrary

Arguments

```
locator: str
```

Documentation

Simulates releasing the left mouse button on the element `locator`.

See the *Locating elements* section for details about the locator syntax.

Open Browser

Arguments

```
url: str = None
browser: str = firefox
alias: str = None
remote_url: typing.Union[str, bool] = False
desired_capabilities: typing.Union[str, dict, NoneType] = None
ff_profile_dir: str = None
options: typing.Any = None
service_log_path: str = None
executable_path: str = None
```

Documentation

Opens a new browser instance to the optional `url`.

The `browser` argument specifies which browser to use. The supported browsers are listed in the table below. The browser names are case-insensitive and some browsers have multiple supported names.

Browser	Name(s)
Firefox	firefox, ff
Google Chrome	googlechrome, chrome, gc
Headless Firefox	headlessfirefox
Headless Chrome	headlesschrome
Internet Explorer	internetexplorer, ie
Edge	edge
Safari	safari
Opera	opera
Android	android
Iphone	iphone
PhantomJS	phantomjs
HTMLUnit	htmlunit
HTMLUnit with Javascript	htmlunitwithjs

To be able to actually use one of these browsers, you need to have a matching Selenium browser driver available. See the project documentation for more details. Headless Firefox and Headless Chrome are new additions in SeleniumLibrary 3.1.0 and require Selenium 3.8.0 or newer.

After opening the browser, it is possible to use optional `url` to navigate the browser to the desired address.

Optional `alias` is an alias given for this browser instance and it can be used for switching between browsers. When same `alias` is given with two *Open Browser* keywords, the first keyword will open a new browser, but the second one will switch to the already opened browser and will not open a new browser. The `alias` definition overrules `browser` definition. When same `alias` is used but a different `browser` is defined, then switch to a browser with same alias is done and new browser is not opened. An alternative approach for switching is using an index returned by this keyword. These indices start from 1, are incremented when new browsers are opened, and reset back to 1 when *Close All Browsers* is called. See *Switch Browser* for more information and examples.

Optional `remote_url` is the URL for a Selenium Grid.

Optional `desired_capabilities` can be used to configure, for example, logging preferences for a browser or a browser and operating system when using Sauce Labs. Desired capabilities can be given either as a Python dictionary or as a string in the format `key1:value1,key2:value2`. Selenium documentation lists possible capabilities that can be enabled.

Optional `ff_profile_dir` is the path to the Firefox profile directory if you wish to overwrite the default profile Selenium uses. Notice that prior to SeleniumLibrary 3.0, the library contained its own profile that was used by default. The `ff_profile_dir` can also be an instance of the selenium.webdriver.FirefoxProfile . As a third option, it is possible to use *FirefoxProfile* methods and attributes to define the profile using methods and attributes in the same way as with `options` argument. Example: It is possible to use FirefoxProfile *set_preference* to define different profile settings. See `options` argument documentation in below how to handle backslash escaping.
Optional `options` argument allows defining browser specific Selenium options. Example for Chrome, the `options` argument allows defining the following methods and attributes and for Firefox these methods and attributes are available. Please note that not all browsers, supported by the SeleniumLibrary, have Selenium options available. Therefore please consult the Selenium documentation which browsers do support the Selenium options. If `browser` argument is *android* then Chrome options is used. Selenium options are also supported, when `remote_url` argument is used.

The SeleniumLibrary `options` argument accepts Selenium options in two different formats: as a string and as Python object which is an instance of the Selenium options class.

The string format allows defining Selenium options methods or attributes and their arguments in Robot Framework test data. The method and attributes names are case and space sensitive and must match to the Selenium options methods and attributes names. When defining a method, it must be defined in a similar way as in python: method name, opening parenthesis, zero to many arguments and closing parenthesis. If there is a need to define multiple arguments for a single method, arguments must be separated with comma, just like in Python. Example: *add_argument("--headless")* or *add_experimental_option("key", "value")*. Attributes are defined in a similar way as in Python: attribute name, equal sign, and attribute value. Example, *headless=True*. Multiple methods and attributes must be separated by a semicolon. Example: *add_argument("--headless");add_argument("--start-maximized")*.

Arguments allow defining Python data types and arguments are evaluated by using Python ast.literal_eval. Strings must be quoted with single or double quotes, example "value" or 'value'. It is also possible to define other Python builtin data types, example *True* or *None*, by not using quotes around the arguments.

The string format is space friendly. Usually, spaces do not alter the defining methods or attributes. There are two exceptions. In some Robot Framework test data formats, two or more spaces are considered as cell separator and instead of defining a single argument, two or more arguments may be defined. Spaces in string arguments are not removed and are left as is. Example *add_argument ("--headless")* is same as *add_argument("--headless")*. But *add_argument(" --headless ")* is not same same as *add_argument ("--headless")*, because spaces inside of quotes are not removed. Please note that if options string contains backslash, example a Windows OS path, the backslash needs escaping both in Robot Framework data and in Python side. This means single backslash must be writen using four backslash characters. Example, Windows path: "C:\path\to\profile" must be written as "C:\\\\path\\\to\\\\profile". Another way to write backslash is use Python raw strings and example write: r"C:\\path\\to\\profile".

As last format, `options` argument also supports receiving the Selenium options as Python class instance. In this case, the instance is used as-is and the SeleniumLibrary will not convert the instance to other formats. For example, if the following code return value is saved to *${options}* variable in the Robot Framework data:

```
options = webdriver.ChromeOptions()
options.add_argument('--disable-dev-shm-usage')
```

SeleniumLibrary

```
return options
```

Then the *${options}* variable can be used as an argument to `options`.

Example the `options` argument can be used to launch Chomium-based applications which utilize the Chromium Embedded Framework . To lauch Chomium-based application, use `options` to define *binary_location* attribute and use *add_argument* method to define *remote-debugging-port* port for the application. Once the browser is opened, the test can interact with the embedded web-content of the system under test.

Optional `service_log_path` argument defines the name of the file where to write the browser driver logs. If the `service_log_path` argument contain a marker `{index}`, it will be automatically replaced with unique running index preventing files to be overwritten. Indices start's from 1, and how they are represented can be customized using Python's format string syntax.

Optional `executable_path` argument defines the path to the driver executable, example to a chromedriver or a geckodriver. If not defined it is assumed the executable is in the $PATH.

Examples:

Open Browser	http://example.com	Chrome	
Open Browser	http://example.com	Firefox	alias=Firefox
Open Browser	http://example.com	Edge	remote_url=http://127.0.0.1:4444/wd/hub
Open Browser	about:blank		
Open Browser	browser=Chrome		

Alias examples:

${1_index} =	*Open Browser*	http://example.com	Chrome	alias=Chrome	# Opens new browser because alias is new.
${2_index} =	*Open Browser*	http://example.com	Firefox		# Opens new browser because alias is not defined.
${3_index} =	*Open Browser*	http://example.com	Chrome	alias=Chrome	# Switches to the browser with Chrome alias.
${4_index} =	*Open Browser*	http://example.com	Chrome	alias=${1_index}	# Switches to the browser with Chrome alias.
Should Be Equal	${1_index}	${3_index}			
Should Be Equal	${1_index}	${4_index}			
Should Be Equal	${2_index}	${2}			

Example when using Chrome options method:

Open Browser	http://example.com	Chrome	options=add_argument("--disable-popup-blocking"); add_argument("--ignore-certificate-errors")
${options} =	Get Options		
Open Browser	http://example.com	Chrome	options=${options}
Open Browser	None	Chrome	options=binary_location="/path/to/binary";add_argument(debugging-port=port")
Open Browser	None	Chrome	options=binary_location=r"C:\\path\\to\\binary"

Example for FirefoxProfile

Open Browser	http://example.com	Firefox	ff_profile_dir=/path/to/profile	# Using profile from disk.
Open Browser	http://example.com	Firefox	ff_profile_dir=${FirefoxProfile_instance}	# Using instance of FirefoxProfile.
Open			ff_profile_dir=set_preference("key",	# Defining profile

SeleniumLibrary

Open Browser	http://example.com	Firefox	"value");set_preference("other", "setting")	using FirefoxProfile mehtods.

If the provided configuration options are not enough, it is possible to use *Create Webdriver* to customize browser initialization even more.

Applying `desired_capabilities` argument also for local browser is new in SeleniumLibrary 3.1.

Using `alias` to decide, is the new browser opened is new in SeleniumLibrary 4.0. The `options` and `service_log_path` are new in SeleniumLibrary 4.0. Support for `ff_profile_dir` accepting an instance of the *selenium.webdriver.FirefoxProfile* and support defining FirefoxProfile with methods and attributes are new in SeleniumLibrary 4.0.

Making `url` optional is new in SeleniumLibrary 4.1.

The `executable_path` argument is new in SeleniumLibrary 4.2.

Open Context Menu

Arguments

```
locator: str
```

Documentation

Opens the context menu on the element identified by `locator`.

Page Should Contain

Arguments

```
text: str
loglevel: str = TRACE
```

Documentation

Verifies that current page contains `text`.

If this keyword fails, it automatically logs the page source using the log level specified with the optional `loglevel` argument. Valid log levels are `DEBUG`, `INFO` (default), `WARN`, and `NONE`. If the log level is `NONE` or below the current active log level the source will not be logged.

Page Should Contain Button

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies button `locator` is found from current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, buttons are searched using `id`, `name`, and `value`.

Page Should Contain Checkbox

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies checkbox `locator` is found from the current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax.

Page Should Contain Element

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
limit: int = None
```

Documentation

Verifies that element `locator` is found on the current page.

See the *Locating elements* section for details about the locator syntax.

The `message` argument can be used to override the default error message.

The `limit` argument can used to define how many elements the page should contain. When `limit` is `None` (default) page can contain one or more elements. When limit is a number, page must contain same number of elements.

See *Page Should Contain* for an explanation about the `loglevel` argument.

Examples assumes that locator matches to two elements.

Page Should Contain Element	div_name	limit=1	# Keyword fails.
Page Should Contain Element	div_name	limit=2	# Keyword passes.
Page Should Contain Element	div_name	limit=none	# None is considered one or more.
Page Should Contain Element	div_name		# Same as above.

The `limit` argument is new in SeleniumLibrary 3.0.

Page Should Contain Image

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies image identified by `locator` is found from current page.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, images are searched using `id`, `name`, `src` and `alt`.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

Page Should Contain Link

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies link identified by `locator` is found from current page.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, links are searched using `id`, `name`, `href` and the link text.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

Page Should Contain List

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies selection list `locator` is found from current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax.

Page Should Contain Radio Button

SeleniumLibrary

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies radio button `locator` is found from current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, radio buttons are searched using `id`, `name` and `value`.

Page Should Contain Textfield

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies text field `locator` is found from current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax.

Page Should Not Contain

Arguments

```
text: str
loglevel: str = TRACE
```

Documentation

Verifies the current page does not contain `text`.

See *Page Should Contain* for an explanation about the `loglevel` argument.

Page Should Not Contain Button

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies button `locator` is not found from current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, buttons are searched using `id`, `name`, and `value`.

Page Should Not Contain Checkbox

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies checkbox `locator` is not found from the current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax.

Page Should Not Contain Element

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies that element `locator` is not found on the current page.

See the *Locating elements* section for details about the locator syntax.

See *Page Should Contain* for an explanation about `message` and `loglevel` arguments.

Page Should Not Contain Image

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies image identified by `locator` is not found from current page.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, images are searched using `id`, `name`, `src` and `alt`.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

Page Should Not Contain Link

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies link identified by `locator` is not found from current page.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, links are searched using `id`, `name`, `href` and the link text.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

Page Should Not Contain List

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies selection list `locator` is not found from current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax.

Page Should Not Contain Radio Button

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies radio button `locator` is not found from current page.

Verifies radio button `locator` is not found from current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax. When using the default locator strategy, radio buttons are searched using `id`, `name` and `value`.

Page Should Not Contain Textfield

Arguments

```
locator: str
message: str = None
loglevel: str = TRACE
```

Documentation

Verifies text field `locator` is not found from current page.

See *Page Should Contain Element* for an explanation about `message` and `loglevel` arguments.

See the *Locating elements* section for details about the locator syntax.

Press Key

Arguments

```
locator: str
key: str
```

Documentation

DEPRECATED in SeleniumLibrary 4.0. use *Press Keys* instead.

Press Keys

Arguments

```
locator: str = None
*keys: str
```

Documentation

Simulates the user pressing key(s) to an element or on the active browser.

If `locator` evaluates as false, see *Boolean arguments* for more details, then the `keys` are sent to the currently active browser. Otherwise element is searched and `keys` are send to the element identified by the `locator`. In later case, keyword fails if element is not found. See the *Locating elements* section for details about the locator syntax.

`keys` arguments can contain one or many strings, but it can not be empty. `keys` can also be a combination of Selenium Keys and strings or a single Selenium Key. If Selenium Key is combined with strings, Selenium key and strings must be separated by the + character, like in *CONTROL+c*. Selenium Keys are space and case sensitive and Selenium Keys are not parsed

SeleniumLibrary

inside of the string. Example AALTO, would send string *AALTO* and *ALT* not parsed inside of the string. But *A+ALT+O* would found Selenium ALT key from the `keys` argument. It also possible to press many Selenium Keys down at the same time, example 'ALT+ARROW_DOWN`.

If Selenium Keys are detected in the `keys` argument, keyword will press the Selenium Key down, send the strings and then release the Selenium Key. If keyword needs to send a Selenium Key as a string, then each character must be separated with + character, example *E+N+D*.

CTRL is alias for Selenium CONTROL and ESC is alias for Selenium ESCAPE

New in SeleniumLibrary 3.3

Examples:

Press Keys	text_field	AAAAA		# Sends string "AAAAA" to element.
Press Keys	None	BBBBB		# Sends string "BBBBB" to currently active browser.
Press Keys	text_field	E+N+D		# Sends string "END" to element.
Press Keys	text_field	XXX	YY	# Sends strings "XXX" and "YY" to element.
Press Keys	text_field	XXX+YY		# Same as above.
Press Keys	text_field	ALT+ARROW_DOWN		# Pressing "ALT" key down, then pressing ARROW_DOWN and then releasing both keys.
Press Keys	text_field	ALT	ARROW_DOWN	# Pressing "ALT" key and then pressing ARROW_DOWN.
Press Keys	text_field	CTRL+c		# Pressing CTRL key down, sends string "c" and then releases CTRL key.
Press Keys	button	RETURN		# Pressing "ENTER" key to element.

Radio Button Should Be Set To

Arguments

```
group_name: str
value: str
```

Documentation

Verifies radio button group `group_name` is set to `value`.

`group_name` is the `name` of the radio button group.

Radio Button Should Not Be Selected

Arguments

```
group_name: str
```

Documentation

Verifies radio button group `group_name` has no selection.

`group_name` is the `name` of the radio button group.

SeleniumLibrary

Register Keyword To Run On Failure

Arguments

`keyword: str`

Documentation

Sets the keyword to execute, when a SeleniumLibrary keyword fails.

`keyword` is the name of a keyword that will be executed if a SeleniumLibrary keyword fails. It is possible to use any available keyword, including user keywords or keywords from other libraries, but the keyword must not take any arguments.

The initial keyword to use is set when *importing* the library, and the keyword that is used by default is *Capture Page Screenshot*. Taking a screenshot when something failed is a very useful feature, but notice that it can slow down the execution.

It is possible to use string `NOTHING` or `NONE`, case-insensitively, as well as Python `None` to disable this feature altogether.

This keyword returns the name of the previously registered failure keyword or Python `None` if this functionality was previously disabled. The return value can be always used to restore the original value later.

Example:

Register Keyword To Run On Failure	Log Source	
${previous kw}=	*Register Keyword To Run On Failure*	NONE
Register Keyword To Run On Failure	${previous kw}	

Changes in SeleniumLibrary 3.0:

- Possible to use string `NONE` or Python `None` to disable the functionality.
- Return Python `None` when the functionality was disabled earlier. In previous versions special value `No Keyword` was returned and it could not be used to restore the original state.

Reload Page

Documentation

Simulates user reloading page.

Remove Location Strategy

Arguments

`strategy_name: str`

Documentation

Removes a previously added custom location strategy.

See *Custom locators* for information on how to create and use custom strategies.

Scroll Element Into View

SeleniumLibrary

Arguments

`locator: str`

Documentation

Scrolls the element identified by `locator` into view.

See the *Locating elements* section for details about the locator syntax.

New in SeleniumLibrary 3.2.0

Select All From List

Arguments

`locator: str`

Documentation

Selects all options from multi-selection list `locator`.

See the *Locating elements* section for details about the locator syntax.

Select Checkbox

Arguments

`locator: str`

Documentation

Selects the checkbox identified by `locator`.

Does nothing if checkbox is already selected.

See the *Locating elements* section for details about the locator syntax.

Select Frame

Arguments

`locator: str`

Documentation

Sets frame identified by `locator` as the current frame.

See the *Locating elements* section for details about the locator syntax.

Works both with frames and iframes. Use *Unselect Frame* to cancel the frame selection and return to the main frame.

Example:

Select Frame	top-frame	# Select frame with id or name 'top-frame'
Click Link	example	# Click link 'example' in the selected frame
Unselect Frame		# Back to main frame.
Select Frame	//iframe[@name='xxx']	# Select frame using xpath

Select From List By Index

Arguments

```
locator: str
*indexes: str
```

Documentation

Selects options from selection list `locator` by `indexes`.

Indexes of list options start from 0.

If more than one option is given for a single-selection list, the last value will be selected. With multi-selection lists all specified options are selected, but possible old selections are not cleared.

See the *Locating elements* section for details about the locator syntax.

Select From List By Label

Arguments

```
locator: str
*labels: str
```

Documentation

Selects options from selection list `locator` by `labels`.

If more than one option is given for a single-selection list, the last value will be selected. With multi-selection lists all specified options are selected, but possible old selections are not cleared.

See the *Locating elements* section for details about the locator syntax.

Select From List By Value

Arguments

```
locator: str
*values: str
```

Documentation

Selects options from selection list `locator` by `values`.

If more than one option is given for a single-selection list, the last value will be selected. With multi-selection lists all specified options are selected, but possible old selections are not cleared.

See the *Locating elements* section for details about the locator syntax.

Select Radio Button

Arguments

```
group_name: str
value: str
```

Documentation

Sets the radio button group `group_name` to `value`.

The radio button to be selected is located by two arguments:

- `group_name` is the name of the radio button group.
- `value` is the `id` or `value` attribute of the actual radio button.

Examples:

Select Radio Button	size	XL
Select Radio Button	contact	email

Set Browser Implicit Wait

Arguments

```
value: timedelta
```

Documentation

Sets the implicit wait value used by Selenium.

Same as *Set Selenium Implicit Wait* but only affects the current browser.

Set Focus To Element

Arguments

```
locator: str
```

Documentation

Sets the focus to the element identified by `locator`.

See the *Locating elements* section for details about the locator syntax.

Prior to SeleniumLibrary 3.0 this keyword was named *Focus*.

Set Screenshot Directory

SeleniumLibrary

Arguments

`path: str`

Documentation

Sets the directory for captured screenshots.

`path` argument specifies the absolute path to a directory where the screenshots should be written to. If the directory does not exist, it will be created. The directory can also be set when *importing* the library. If it is not configured anywhere, screenshots are saved to the same directory where Robot Framework's log file is written.

If `path` equals to EMBED (case insensitive) and *Capture Page Screenshot* or *capture Element Screenshot* keywords filename argument is not changed from the default value, then the page or element screenshot is embedded as Base64 image to the log.html.

The previous value is returned and can be used to restore the original value later if needed.

Returning the previous value is new in SeleniumLibrary 3.0. The persist argument was removed in SeleniumLibrary 3.2 and EMBED is new in SeleniumLibrary 4.2.

Set Selenium Implicit Wait

Arguments

`value: timedelta`

Documentation

Sets the implicit wait value used by Selenium.

The value can be given as a number that is considered to be seconds or as a human-readable string like `1 second`. The previous value is returned and can be used to restore the original value later if needed.

This keyword sets the implicit wait for all opened browsers. Use *Set Browser Implicit Wait* to set it only to the current browser.

See the *Implicit wait* section above for more information.

Example:

${orig wait} =	*Set Selenium Implicit Wait*	10 seconds
Perform AJAX call that is slow		
Set Selenium Implicit Wait	${orig wait}	

Set Selenium Speed

Arguments

`value: timedelta`

Documentation

Sets the delay that is waited after each Selenium command.

The value can be given as a number that is considered to be seconds or as a human-readable string like `1 second`. The previous value is returned and can be used to restore the original

SeleniumLibrary

value later if needed.

See the *Selenium Speed* section above for more information.

Example:

Set Selenium Speed	0.5 seconds

Set Selenium Timeout

Arguments

```
value: timedelta
```

Documentation

Sets the timeout that is used by various keywords.

The value can be given as a number that is considered to be seconds or as a human-readable string like `1 second`. The previous value is returned and can be used to restore the original value later if needed.

See the *Timeout* section above for more information.

Example:

${orig timeout} =	*Set Selenium Timeout*	15 seconds
Open page that loads slowly		
Set Selenium Timeout	${orig timeout}	

Set Window Position

Arguments

```
x: int
y: int
```

Documentation

Sets window position using `x` and `y` coordinates.

The position is relative to the top left corner of the screen, but some browsers exclude possible task bar set by the operating system from the calculation. The actual position may thus be different with different browsers.

Values can be given using strings containing numbers or by using actual numbers. See also *Get Window Position*.

Example:

Set Window Position	100	200

Set Window Size

Arguments

```
width: int
```

SeleniumLibrary

```
height: int
inner: bool = False
```

Documentation

Sets current windows size to given `width` and `height`.

Values can be given using strings containing numbers or by using actual numbers. See also *Get Window Size*.

Browsers have a limit on their minimum size. Trying to set them smaller will cause the actual size to be bigger than the requested size.

If `inner` parameter is set to True, keyword sets the necessary window width and height to have the desired HTML DOM *window.innerWidth* and *window.innerHeight*. See *Boolean arguments* for more details on how to set boolean arguments.

The `inner` argument is new since SeleniumLibrary 4.0.

This `inner` argument does not support Frames. If a frame is selected, switch to default before running this.

Example:

Set Window Size	800	600	
Set Window Size	800	600	True

Simulate Event

Arguments

```
locator: str
event: str
```

Documentation

Simulates `event` on the element identified by `locator`.

This keyword is useful if element has `OnEvent` handler that needs to be explicitly invoked.

See the *Locating elements* section for details about the locator syntax.

Prior to SeleniumLibrary 3.0 this keyword was named *Simulate*.

Submit Form

Arguments

```
locator: str = None
```

Documentation

Submits a form identified by `locator`.

If `locator` is not given, first form on the page is submitted.

See the *Locating elements* section for details about the locator syntax.

SeleniumLibrary

Switch Browser

Arguments

```
index_or_alias: str
```

Documentation

Switches between active browsers using `index_or_alias`.

Indices are returned by the *Open Browser* keyword and aliases can be given to it explicitly. Indices start from 1.

Example:

Open Browser	http://google.com	ff	
Location Should Be	http://google.com		
Open Browser	http://yahoo.com	ie	alias=second
Location Should Be	http://yahoo.com		
Switch Browser	1	# index	
Page Should Contain	I'm feeling lucky		
Switch Browser	second	# alias	
Page Should Contain	More Yahoo!		
Close All Browsers			

Above example expects that there was no other open browsers when opening the first one because it used index `1` when switching to it later. If you are not sure about that, you can store the index into a variable as below.

${index} =	*Open Browser*	http://google.com
# Do something ...		
Switch Browser	${index}	

Switch Window

Arguments

```
locator: str = MAIN
timeout: str = None
browser: str = CURRENT
```

Documentation

Switches to browser window matching `locator`.

If the window is found, all subsequent commands use the selected window, until this keyword is used again. If the window is not found, this keyword fails. The previous windows handle is returned and can be used to switch back to it later.

Notice that alerts should be handled with *Handle Alert* or other alert related keywords.

The `locator` can be specified using different strategies somewhat similarly as when *locating elements* on pages.

- By default, the `locator` is matched against window handle, name, title, and URL. Matching is done in that order and the first matching window is selected.
- The `locator` can specify an explicit strategy by using the format `strategy:value` (recommended) or `strategy=value`. Supported strategies are `name`, `title`, and `url`. These matches windows using their name, title, or URL, respectively. Additionally, `default` can be used to explicitly use the default strategy

SeleniumLibrary

explained above.

- If the `locator` is `NEW` (case-insensitive), the latest opened window is selected. It is an error if this is the same as the current window.
- If the `locator` is `MAIN` (default, case-insensitive), the main window is selected.
- If the `locator` is `CURRENT` (case-insensitive), nothing is done. This effectively just returns the current window handle.
- If the `locator` is not a string, it is expected to be a list of window handles *to exclude*. Such a list of excluded windows can be got from *Get Window Handles* before doing an action that opens a new window.

The `timeout` is used to specify how long keyword will poll to select the new window. The `timeout` is new in SeleniumLibrary 3.2.

Example:

Click Link	popup1		# Open new window
Switch Window	example		# Select window using default strategy
Title Should Be	Pop-up 1		
Click Button	popup2		# Open another window
${handle} =	*Switch Window*	NEW	# Select latest opened window
Title Should Be	Pop-up 2		
Switch Window	${handle}		# Select window using handle
Title Should Be	Pop-up 1		
Switch Window	MAIN		# Select the main window
Title Should Be	Main		
${excludes} =	*Get Window Handles*		# Get list of current windows
Click Link	popup3		# Open one more window
Switch Window	${excludes}		# Select window using excludes
Title Should Be	Pop-up 3		

The `browser` argument allows with `index_or_alias` to implicitly switch to a specific browser when switching to a window. See *Switch Browser*

- If the `browser` is `CURRENT` (case-insensitive), no other browser is selected.

NOTE:

- The `strategy:value` syntax is only supported by SeleniumLibrary 3.0 and newer.
- Prior to SeleniumLibrary 3.0 matching windows by name, title and URL was case-insensitive.
- Earlier versions supported aliases `None`, `null` and the empty string for selecting the main window, and alias `self` for selecting the current window. Support for these aliases was removed in SeleniumLibrary 3.2.

Table Cell Should Contain

Arguments

```
locator: str
row: int
column: int
expected: str
loglevel: str = TRACE
```

Documentation

Verifies table cell contains text `expected`.

See *Get Table Cell* that this keyword uses internally for an explanation about accepted arguments.

SeleniumLibrary

Table Column Should Contain

Arguments

```
locator: str
column: int
expected: str
loglevel: str = TRACE
```

Documentation

Verifies table column contains text `expected`.

The table is located using the `locator` argument and its column found using `column`. See the *Locating elements* section for details about the locator syntax.

Column indexes start from 1. It is possible to refer to columns from the end by using negative indexes so that -1 is the last column, -2 is the second last, and so on.

If a table contains cells that span multiple columns, those merged cells count as a single column.

See *Page Should Contain Element* for an explanation about the `loglevel` argument.

Table Footer Should Contain

Arguments

```
locator: str
expected: str
loglevel: str = TRACE
```

Documentation

Verifies table footer contains text `expected`.

Any `<td>` element inside `<tfoot>` element is considered to be part of the footer.

The table is located using the `locator` argument. See the *Locating elements* section for details about the locator syntax.

See *Page Should Contain Element* for an explanation about the `loglevel` argument.

Table Header Should Contain

Arguments

```
locator: str
expected: str
loglevel: str = TRACE
```

Documentation

Verifies table header contains text `expected`.

Any `<th>` element anywhere in the table is considered to be part of the header.

The table is located using the `locator` argument. See the *Locating elements* section for details about the locator syntax.

See *Page Should Contain Element* for an explanation about the `loglevel` argument.

Table Row Should Contain

Arguments

```
locator: str
row: int
expected: str
loglevel: str = TRACE
```

Documentation

Verifies that table row contains text `expected`.

The table is located using the `locator` argument and its column found using `column`. See the *Locating elements* section for details about the locator syntax.

Row indexes start from 1. It is possible to refer to rows from the end by using negative indexes so that -1 is the last row, -2 is the second last, and so on.

If a table contains cells that span multiple rows, a match only occurs for the uppermost row of those merged cells.

See *Page Should Contain Element* for an explanation about the `loglevel` argument.

Table Should Contain

Arguments

```
locator: str
expected: str
loglevel: str = TRACE
```

Documentation

Verifies table contains text `expected`.

The table is located using the `locator` argument. See the *Locating elements* section for details about the locator syntax.

See *Page Should Contain Element* for an explanation about the `loglevel` argument.

Textarea Should Contain

Arguments

```
locator: str
```

```
expected: str
message: str = None
```

Documentation

Verifies text area `locator` contains text `expected`.

`message` can be used to override default error message.

See the *Locating elements* section for details about the locator syntax.

Textarea Value Should Be

Arguments

```
locator: str
expected: str
message: str = None
```

Documentation

Verifies text area `locator` has exactly text `expected`.

`message` can be used to override default error message.

See the *Locating elements* section for details about the locator syntax.

Textfield Should Contain

Arguments

```
locator: str
expected: str
message: str = None
```

Documentation

Verifies text field `locator` contains text `expected`.

`message` can be used to override the default error message.

See the *Locating elements* section for details about the locator syntax.

Textfield Value Should Be

Arguments

```
locator: str
expected: str
message: str = None
```

Documentation

Verifies text field `locator` has exactly text `expected`.

`message` can be used to override default error message.

`message` can be used to override default error message.

See the *Locating elements* section for details about the locator syntax.

Title Should Be

Arguments

```
title: str
message: str = None
```

Documentation

Verifies that the current page title equals `title`.

The `message` argument can be used to override the default error message.

`message` argument is new in SeleniumLibrary 3.1.

Unselect All From List

Arguments

```
locator: str
```

Documentation

Unselects all options from multi-selection list `locator`.

See the *Locating elements* section for details about the locator syntax.

New in SeleniumLibrary 3.0.

Unselect Checkbox

Arguments

```
locator: str
```

Documentation

Removes the selection of checkbox identified by `locator`.

Does nothing if the checkbox is not selected.

See the *Locating elements* section for details about the locator syntax.

Unselect Frame

Documentation

Sets the main frame as the current frame.

In practice cancels the previous *Select Frame* call.

Unselect From List By Index

Arguments

```
locator: str
*indexes: str
```

Documentation

Unselects options from selection list `locator` by `indexes`.

Indexes of list options start from 0. This keyword works only with multi-selection lists.

See the *Locating elements* section for details about the locator syntax.

Unselect From List By Label

Arguments

```
locator: str
*labels: str
```

Documentation

Unselects options from selection list `locator` by `labels`.

This keyword works only with multi-selection lists.

See the *Locating elements* section for details about the locator syntax.

Unselect From List By Value

Arguments

```
locator: str
*values: str
```

Documentation

Unselects options from selection list `locator` by `values`.

This keyword works only with multi-selection lists.

See the *Locating elements* section for details about the locator syntax.

Wait For Condition

Arguments

```
condition: str
```

SeleniumLibrary

```
timeout: timedelta = None
error: str = None
```

Documentation

Waits until `condition` is true or `timeout` expires.

The condition can be arbitrary JavaScript expression but it must return a value to be evaluated. See *Execute JavaScript* for information about accessing content on pages.

Fails if the timeout expires before the condition becomes true. See the *Timeouts* section for more information about using timeouts and their default value.

`error` can be used to override the default error message.

Examples:

Wait For Condition	return document.title == "New Title"
Wait For Condition	return jQuery.active == 0
Wait For Condition	style = document.querySelector('h1').style; return style.background == "red" && style.color == "white"

Wait Until Element Contains

Arguments

```
locator: str
text: str
timeout: timedelta = None
error: str = None
```

Documentation

Waits until the element `locator` contains `text`.

Fails if `timeout` expires before the text appears. See the *Timeouts* section for more information about using timeouts and their default value and the *Locating elements* section for details about the locator syntax.

`error` can be used to override the default error message.

Wait Until Element Does Not Contain

Arguments

```
locator: str
text: str
timeout: timedelta = None
error: str = None
```

Documentation

Waits until the element `locator` does not contain `text`.

Fails if `timeout` expires before the text disappears. See the *Timeouts* section for more information about using timeouts and their default value and the *Locating elements* section for details about the locator syntax.

`error` can be used to override the default error message.

Wait Until Element Is Enabled

Arguments

```
locator: str
timeout: timedelta = None
error: str = None
```

Documentation

Waits until the element `locator` is enabled.

Element is considered enabled if it is not disabled nor read-only.

Fails if `timeout` expires before the element is enabled. See the *Timeouts* section for more information about using timeouts and their default value and the *Locating elements* section for details about the locator syntax.

`error` can be used to override the default error message.

Considering read-only elements to be disabled is a new feature in SeleniumLibrary 3.0.

Wait Until Element Is Not Visible

Arguments

```
locator: str
timeout: timedelta = None
error: str = None
```

Documentation

Waits until the element `locator` is not visible.

Fails if `timeout` expires before the element is not visible. See the *Timeouts* section for more information about using timeouts and their default value and the *Locating elements* section for details about the locator syntax.

`error` can be used to override the default error message.

Wait Until Element Is Visible

Arguments

```
locator: str
timeout: timedelta = None
error: str = None
```

Documentation

Waits until the element `locator` is visible.

Fails if `timeout` expires before the element is visible. See the *Timeouts* section for more information about using timeouts and their default value and the *Locating elements* section for details about the locator syntax.

`error` can be used to override the default error message.

Wait Until Location Contains

Arguments

```
expected: str
timeout: timedelta = None
message: str = None
```

Documentation

Waits until the current URL contains `expected`.

The `expected` argument contains the expected value in url.

Fails if `timeout` expires before the location contains. See the *Timeouts* section for more information about using timeouts and their default value.

The `message` argument can be used to override the default error message.

New in SeleniumLibrary 4.0

Wait Until Location Does Not Contain

Arguments

```
location: str
timeout: timedelta = None
message: str = None
```

Documentation

Waits until the current URL does not contains `location`.

The `location` argument contains value not expected in url.

Fails if `timeout` expires before the location not contains. See the *Timeouts* section for more information about using timeouts and their default value.

The `message` argument can be used to override the default error message.

New in SeleniumLibrary 4.3

Wait Until Location Is

Arguments

```
expected: str
timeout: timedelta = None
message: str = None
```

SeleniumLibrary

Documentation

Waits until the current URL is `expected`.

The `expected` argument is the expected value in url.

Fails if `timeout` expires before the location is. See the *Timeouts* section for more information about using timeouts and their default value.

The `message` argument can be used to override the default error message.

New in SeleniumLibrary 4.0

Wait Until Location Is Not

Arguments

```
location: str
timeout: timedelta = None
message: str = None
```

Documentation

Waits until the current URL is not `location`.

The `location` argument is the unexpected value in url.

Fails if `timeout` expires before the location is not. See the *Timeouts* section for more information about using timeouts and their default value.

The `message` argument can be used to override the default error message.

New in SeleniumLibrary 4.3

Wait Until Page Contains

Arguments

```
text: str
timeout: timedelta = None
error: str = None
```

Documentation

Waits until `text` appears on the current page.

Fails if `timeout` expires before the text appears. See the *Timeouts* section for more information about using timeouts and their default value.

`error` can be used to override the default error message.

Wait Until Page Contains Element

SeleniumLibrary

Arguments

```
locator: str
timeout: timedelta = None
error: str = None
limit: int = None
```

Documentation

Waits until the element `locator` appears on the current page.

Fails if `timeout` expires before the element appears. See the *Timeouts* section for more information about using timeouts and their default value and the *Locating elements* section for details about the locator syntax.

`error` can be used to override the default error message.

The `limit` argument can used to define how many elements the page should contain. When `limit` is *None* (default) page can contain one or more elements. When limit is a number, page must contain same number of elements.

`limit` is new in SeleniumLibrary 4.4

Wait Until Page Does Not Contain

Arguments

```
text: str
timeout: timedelta = None
error: str = None
```

Documentation

Waits until `text` disappears from the current page.

Fails if `timeout` expires before the text disappears. See the *Timeouts* section for more information about using timeouts and their default value.

`error` can be used to override the default error message.

Wait Until Page Does Not Contain Element

Arguments

```
locator: str
timeout: timedelta = None
error: str = None
limit: int = None
```

Documentation

Waits until the element `locator` disappears from the current page.

Fails if `timeout` expires before the element disappears. See the *Timeouts* section for more information about using timeouts and their default value and the *Locating elements* section for details about the locator syntax.

`error` can be used to override the default error message.

SeleniumLibrary

The `limit` argument can used to define how many elements the page should not contain. When `limit` is *None* (default) page can`t contain any elements. When limit is a number, page must not contain same number of elements.

`limit` is new in SeleniumLibrary 4.4

Altogether 173 keywords.
Generated by Libdoc on 2020-10-11 23:23:56.