7. WebDriver API

> **Note:**
>
> This is not an official documentation. Official API documentation is available here.

This chapter covers all the interfaces of Selenium WebDriver.

Recommended Import Style

The API definitions in this chapter show the absolute location of classes. However, the recommended import style is as given below:

```
from selenium import webdriver
```

Then, you can access the classes like this:

```
webdriver.Firefox
webdriver.FirefoxProfile
webdriver.Chrome
webdriver.ChromeOptions
webdriver.Ie
webdriver.Opera
webdriver.PhantomJS
webdriver.Remote
webdriver.DesiredCapabilities
webdriver.ActionChains
webdriver.TouchActions
webdriver.Proxy
```

The special keys class (`Keys`) can be imported like this:

```
from selenium.webdriver.common.keys import Keys
```

The exception classes can be imported like this (Replace the `TheNameOfTheExceptionClass` with the actual class name given below):

```
from selenium.common.exceptions import [TheNameOfTheExceptionClass]
```

Conventions used in the API

Some attributes are callable (or methods) and others are non-callable (properties). All the callable attributes are ending with round brackets.

Here is an example for property:

- current_url

  URL of the currently loaded page.

  Usage:

  ```
  driver.current_url
  ```


Here is an example of a method:

- close()

  Closes the current window.

  Usage:

  ```
  driver.close()
  ```

7.1. Exceptions

Exceptions that may happen in all the webdriver code.

exception selenium.common.exceptions.**ElementClickInterceptedException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

The Element Click command could not be completed because the element receiving the events is obscuring the element that was requested clicked.

exception selenium.common.exceptions.**ElementNotInteractableException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.InvalidElementStateException**

Thrown when an element is present in the DOM but interactions with that element will hit another element do to paint order

exception selenium.common.exceptions.**ElementNotSelectableException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.InvalidElementStateException**

Thrown when trying to select an unselectable element.

For example, selecting a 'script' element.

exception selenium.common.exceptions.**ElementNotVisibleException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.InvalidElementStateException**

Thrown when an element is present on the DOM, but it is not visible, and so is not able to be interacted with.

Most commonly encountered when trying to click or read text of an element that is hidden from view.

exception selenium.common.exceptions.**ErrorInResponseException**(*response*, *msg*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when an error has occurred on the server side.

This may happen when communicating with the firefox extension or the remote driver server.

__init__(*response*, *msg*)

Initialize self. See help(type(self)) for accurate signature.

exception selenium.common.exceptions.**ImeActivationFailedException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when activating an IME engine has failed.

exception selenium.common.exceptions.**ImeNotAvailableException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when IME support is not available. This exception is thrown for every IME-related method call if IME support is not available on the machine.

exception selenium.common.exceptions.**InsecureCertificateException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Navigation caused the user agent to hit a certificate warning, which is usually the result of an expired or invalid TLS certificate.

exception selenium.common.exceptions.**InvalidArgumentException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

The arguments passed to a command are either invalid or malformed.

exception selenium.common.exceptions.**InvalidCookieDomainException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when attempting to add a cookie under a different domain than the current URL.

exception selenium.common.exceptions.**InvalidCoordinatesException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

The coordinates provided to an interactions operation are invalid.

exception selenium.common.exceptions.**InvalidElementStateException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when a command could not be completed because the element is in an invalid state.

This can be caused by attempting to clear an element that isn't both editable and resettable.

exception selenium.common.exceptions.**InvalidSelectorException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.NoSuchElementException**

Thrown when the selector which is used to find an element does not return a WebElement. Currently this only happens when the selector is an xpath expression and it is either syntactically invalid (i.e. it is not a xpath expression) or the expression does not select WebElements (e.g. "count(//input)").

exception selenium.common.exceptions.**InvalidSessionIdException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Occurs if the given session id is not in the list of active sessions, meaning the session either does not exist or that it's not active.

exception selenium.common.exceptions.**InvalidSwitchToTargetException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when frame or window target to be switched doesn't exist.

exception selenium.common.exceptions.**JavascriptException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

An error occurred while executing JavaScript supplied by the user.

exception selenium.common.exceptions.**MoveTargetOutOfBoundsException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when the target provided to the *ActionsChains* move() method is invalid, i.e. out of document.

exception selenium.common.exceptions.**NoAlertPresentException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when switching to no presented alert.

This can be caused by calling an operation on the Alert() class when an alert is not yet on the screen.

exception selenium.common.exceptions.**NoSuchAttributeException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when the attribute of element could not be found.

You may want to check if the attribute exists in the particular browser you are testing against. Some browsers may have different property names for the same property. (IE8's .innerText vs. Firefox .textContent)

exception selenium.common.exceptions.**NoSuchCookieException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

No cookie matching the given path name was found amongst the associated cookies of the current browsing context's active document.

exception selenium.common.exceptions.**NoSuchElementException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when element could not be found.

If you encounter this exception, you may want to check the following:

- Check your selector used in your find_by...
- Element may not yet be on the screen at the time of the find operation, (webpage is still loading) see selenium.webdriver.support.wait.WebDriverWait() for how to write a wait wrapper to

wait for an element to appear.

exception selenium.common.exceptions.**NoSuchFrameException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.InvalidSwitchToTargetException**

Thrown when frame target to be switched doesn't exist.

exception selenium.common.exceptions.**NoSuchWindowException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.InvalidSwitchToTargetException**

Thrown when window target to be switched doesn't exist.

To find the current set of active window handles, you can get a list of the active window handles in the following way:

```
print driver.window_handles
```

exception selenium.common.exceptions.**RemoteDriverServerException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

exception selenium.common.exceptions.**ScreenshotException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

A screen capture was made impossible.

exception selenium.common.exceptions.**SessionNotCreatedException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

A new session could not be created.

exception selenium.common.exceptions.**StaleElementReferenceException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when a reference to an element is now "stale".

Stale means the element no longer appears on the DOM of the page.

Possible causes of StaleElementReferenceException include, but not limited to:

- You are no longer on the same page, or the page may have refreshed since the element was located.
- The element may have been removed and re-added to the screen, since it was located. Such as an element being relocated. This can happen typically with a javascript framework when values are updated and the node is rebuilt.
- Element may have been inside an iframe or another context which was refreshed.

exception selenium.common.exceptions.**TimeoutException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when a command does not complete in enough time.

exception selenium.common.exceptions.**UnableToSetCookieException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when a driver fails to set a cookie.

exception selenium.common.exceptions.**UnexpectedAlertPresentException**(*msg=None*, *screen=None*, *stacktrace=None*, *alert_text=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when an unexpected alert is appeared.

Usually raised when when an expected modal is blocking webdriver form executing any more commands.

__init__(*msg=None*, *screen=None*, *stacktrace=None*, *alert_text=None*)

Initialize self. See help(type(self)) for accurate signature.

exception selenium.common.exceptions.**UnexpectedTagNameException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

Thrown when a support class did not get an expected web element.

exception selenium.common.exceptions.**UnknownMethodException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **selenium.common.exceptions.WebDriverException**

The requested command matched a known URL but did not match an method for that URL.

exception selenium.common.exceptions.**WebDriverException**(*msg=None*, *screen=None*, *stacktrace=None*)

Bases: **Exception**

Base webdriver exception.

__init__(*msg=None*, *screen=None*, *stacktrace=None*)

Initialize self. See help(type(self)) for accurate signature.

7.2. Action Chains

The ActionChains implementation,

class selenium.webdriver.common.action_chains.**ActionChains**(*driver*)

Bases: **object**

ActionChains are a way to automate low level interactions such as mouse movements, mouse button actions, key press, and context menu interactions. This is useful for doing more complex actions like hover over and drag and drop.

Generate user actions.

When you call methods for actions on the ActionChains object, the actions are stored in a queue in the ActionChains object. When you call perform(), the events are fired in the order they are queued up.

ActionChains can be used in a chain pattern:

```
menu = driver.find_element_by_css_selector(".nav")
hidden_submenu = driver.find_element_by_css_selector(".nav #submenu1")

ActionChains(driver).move_to_element(menu).click(hidden_submenu).perform()
```

Or actions can be queued up one by one, then performed.:

```
menu = driver.find_element_by_css_selector(".nav")
hidden_submenu = driver.find_element_by_css_selector(".nav #submenu1")

actions = ActionChains(driver)
actions.move_to_element(menu)
actions.click(hidden_submenu)
actions.perform()
```

Either way, the actions are performed in the order they are called, one after another.

__init__(*driver*)

Creates a new ActionChains.

Args:
- driver: The WebDriver instance which performs user actions.

click(*on_element=None*)

Clicks an element.

Args:
- on_element: The element to click. If None, clicks on current mouse position.

click_and_hold(*on_element=None*)

Holds down the left mouse button on an element.

Args:
- on_element: The element to mouse down. If None, clicks on current mouse position.

context_click(*on_element=None*)

Performs a context-click (right click) on an element.

Args:
- on_element: The element to context-click. If None, clicks on current mouse position.

double_click(*on_element=None*)

Double-clicks an element.

Args:
- on_element: The element to double-click. If None, clicks on current mouse position.

drag_and_drop(*source*, *target*)

Holds down the left mouse button on the source element,
then moves to the target element and releases the mouse button.

Args:
- source: The element to mouse down.
- target: The element to mouse up.

drag_and_drop_by_offset(*source*, *xoffset*, *yoffset*)

Holds down the left mouse button on the source element,
then moves to the target offset and releases the mouse button.

Args:
- source: The element to mouse down.
- xoffset: X offset to move to.
- yoffset: Y offset to move to.

`key_down`(*value*, *element=None*)

Sends a key press only, without releasing it.

Should only be used with modifier keys (Control, Alt and Shift).

Args:
- value: The modifier key to send. Values are defined in *Keys* class.
- element: The element to send keys. If None, sends a key to current focused element.

Example, pressing ctrl+c:

```
ActionChains(driver).key_down(Keys.CONTROL).send_keys('c').key_up(Keys.CONTROL).perfc
```

`key_up`(*value*, *element=None*)

Releases a modifier key.

Args:
- value: The modifier key to send. Values are defined in Keys class.
- element: The element to send keys. If None, sends a key to current focused element.

Example, pressing ctrl+c:

```
ActionChains(driver).key_down(Keys.CONTROL).send_keys('c').key_up(Keys.CONTROL).perfc
```

`move_by_offset`(*xoffset*, *yoffset*)

Moving the mouse to an offset from current mouse position.

Args:
- xoffset: X offset to move to, as a positive or negative integer.
- yoffset: Y offset to move to, as a positive or negative integer.

`move_to_element`(*to_element*)

Moving the mouse to the middle of an element.

Args:
- to_element: The WebElement to move to.

`move_to_element_with_offset`(*to_element*, *xoffset*, *yoffset*)

Move the mouse by an offset of the specified element.

Offsets are relative to the top-left corner of the element.

Args:
- to_element: The WebElement to move to.
- xoffset: X offset to move to.
- yoffset: Y offset to move to.

`pause`(*seconds*)

Pause all inputs for the specified duration in seconds

`perform`()

Performs all stored actions.

`release`(*on_element=None*)

Releasing a held mouse button on an element.

Args:
- on_element: The element to mouse up. If None, releases on current mouse position.

`reset_actions`()

Clears actions that are already stored locally and on the remote end

`send_keys`(**keys_to_send*)

Sends keys to current focused element.

Args:
- keys_to_send: The keys to send. Modifier keys constants can be found in the 'Keys' class.

send_keys_to_element(*element*, **keys_to_send*)

Sends keys to an element.

Args:
- element: The element to send keys.
- keys_to_send: The keys to send. Modifier keys constants can be found in the 'Keys' class.

7.3. Alerts

The Alert implementation.

class selenium.webdriver.common.alert.**Alert**(*driver*)

Bases: **object**

Allows to work with alerts.

Use this class to interact with alert prompts. It contains methods for dismissing, accepting, inputting, and getting text from alert prompts.

Accepting / Dismissing alert prompts:

```
Alert(driver).accept()
Alert(driver).dismiss()
```

Inputting a value into an alert prompt:

name_prompt = Alert(driver) name_prompt.send_keys("Willian Shakesphere") name_prompt.accept()

Reading a the text of a prompt for verification:

alert_text = Alert(driver).text self.assertEqual("Do you wish to quit?", alert_text)

__init__(*driver*)

Creates a new Alert.

Args:
- driver: The WebDriver instance which performs user actions.

accept()

Accepts the alert available.

Usage:: Alert(driver).accept() # Confirm a alert dialog.

dismiss()

Dismisses the alert available.

send_keys(*keysToSend*)

Send Keys to the Alert.

Args:
- keysToSend: The text to be sent to Alert.

text

Gets the text of the Alert.

7.4. Special Keys

The Keys implementation.

class `selenium.webdriver.common.keys.`**`Keys`**

Bases: **`object`**

Set of special keys codes.

`ADD` = *'\ue025'*

`ALT` = *'\ue00a'*

`ARROW_DOWN` = *'\ue015'*

`ARROW_LEFT` = *'\ue012'*

`ARROW_RIGHT` = *'\ue014'*

`ARROW_UP` = *'\ue013'*

`BACKSPACE` = *'\ue003'*

`BACK_SPACE` = *'\ue003'*

`CANCEL` = *'\ue001'*

`CLEAR` = *'\ue005'*

`COMMAND` = *'\ue03d'*

`CONTROL` = *'\ue009'*

`DECIMAL` = *'\ue028'*

`DELETE` = *'\ue017'*

`DIVIDE` = *'\ue029'*

`DOWN` = *'\ue015'*

`END` = *'\ue010'*

`ENTER` = *'\ue007'*

`EQUALS` = *'\ue019'*

`ESCAPE` = *'\ue00c'*

`F1` = *'\ue031'*

`F10` = *'\ue03a'*

`F11` = *'\ue03b'*

`F12` = *'\ue03c'*

`F2` = *'\ue032'*

`F3` = *'\ue033'*

`F4` = *'\ue034'*

`F5` = *'\ue035'*

`F6` = *'\ue036'*

`F7` = *'\ue037'*

`F8` = *'\ue038'*

F9 = '\ue039'

HELP = '\ue002'

HOME = '\ue011'

INSERT = '\ue016'

LEFT = '\ue012'

LEFT_ALT = '\ue00a'

LEFT_CONTROL = '\ue009'

LEFT_SHIFT = '\ue008'

META = '\ue03d'

MULTIPLY = '\ue024'

NULL = '\ue000'

NUMPAD0 = '\ue01a'

NUMPAD1 = '\ue01b'

NUMPAD2 = '\ue01c'

NUMPAD3 = '\ue01d'

NUMPAD4 = '\ue01e'

NUMPAD5 = '\ue01f'

NUMPAD6 = '\ue020'

NUMPAD7 = '\ue021'

NUMPAD8 = '\ue022'

NUMPAD9 = '\ue023'

PAGE_DOWN = '\ue00f'

PAGE_UP = '\ue00e'

PAUSE = '\ue00b'

RETURN = '\ue006'

RIGHT = '\ue014'

SEMICOLON = '\ue018'

SEPARATOR = '\ue026'

SHIFT = '\ue008'

SPACE = '\ue00d'

SUBTRACT = '\ue027'

TAB = '\ue004'

UP = '\ue013'

7.5. Locate elements By

These are the attributes which can be used to locate elements. See the Locating Elements chapter for example usages.

The By implementation.

class `selenium.webdriver.common.by.`**`By`**

Bases: **`object`**

Set of supported locator strategies.

`CLASS_NAME` = *'class name'*

`CSS_SELECTOR` = *'css selector'*

`ID` = *'id'*

`LINK_TEXT` = *'link text'*

`NAME` = *'name'*

`PARTIAL_LINK_TEXT` = *'partial link text'*

`TAG_NAME` = *'tag name'*

`XPATH` = *'xpath'*

7.6. Desired Capabilities

See the Using Selenium with remote WebDriver section for example usages of desired capabilities.

The Desired Capabilities implementation.

class `selenium.webdriver.common.desired_capabilities.`**`DesiredCapabilities`**

Bases: **`object`**

Set of default supported desired capabilities.

Use this as a starting point for creating a desired capabilities object for requesting remote webdrivers for connecting to selenium server or selenium grid.

Usage Example:

```
from selenium import webdriver

selenium_grid_url = "http://198.0.0.1:4444/wd/hub"

# Create a desired capabilities object as a starting point.
capabilities = DesiredCapabilities.FIREFOX.copy()
capabilities['platform'] = "WINDOWS"
capabilities['version'] = "10"

# Instantiate an instance of Remote WebDriver with the desired capabilities.
driver = webdriver.Remote(desired_capabilities=capabilities,
                          command_executor=selenium_grid_url)
```

Note: Always use '.copy()' on the DesiredCapabilities object to avoid the side effects of altering the Global class instance.

`ANDROID` = *{'browserName': 'android', 'platform': 'ANDROID', 'version': ''}*

`CHROME` = *{'browserName': 'chrome', 'platform': 'ANY', 'version': ''}*

`EDGE` = *{'browserName': 'MicrosoftEdge', 'platform': 'WINDOWS', 'version': ''}*

`FIREFOX` = *{'acceptInsecureCerts': True, 'browserName': 'firefox', 'marionette': True}*

HTMLUNIT = *{'browserName': 'htmlunit', 'platform': 'ANY', 'version': ''}*

HTMLUNITWITHJS = *{'browserName': 'htmlunit', 'javascriptEnabled': True, 'platform': 'ANY', 'version': 'firefox'}*

INTERNETEXPLORER = *{'browserName': 'internet explorer', 'platform': 'WINDOWS', 'version': ''}*

IPAD = *{'browserName': 'iPad', 'platform': 'MAC', 'version': ''}*

IPHONE = *{'browserName': 'iPhone', 'platform': 'MAC', 'version': ''}*

OPERA = *{'browserName': 'opera', 'platform': 'ANY', 'version': ''}*

PHANTOMJS = *{'browserName': 'phantomjs', 'javascriptEnabled': True, 'platform': 'ANY', 'version': ''}*

SAFARI = *{'browserName': 'safari', 'platform': 'MAC', 'version': ''}*

WEBKITGTK = *{'browserName': 'MiniBrowser', 'platform': 'ANY', 'version': ''}*

7.7. Touch Actions

The Touch Actions implementation

class `selenium.webdriver.common.touch_actions.`**`TouchActions`**(*driver*)

Bases: **`object`**

Generate touch actions. Works like ActionChains; actions are stored in the TouchActions object and are fired with perform().

`__init__`(*driver*)

Creates a new TouchActions object.

Args:
- driver: The WebDriver instance which performs user actions. It should be with touch-screen enabled.

`double_tap`(*on_element*)

Double taps on a given element.

Args:
- on_element: The element to tap.

`flick`(*xspeed*, *yspeed*)

Flicks, starting anywhere on the screen.

Args:
- xspeed: The X speed in pixels per second.
- yspeed: The Y speed in pixels per second.

`flick_element`(*on_element*, *xoffset*, *yoffset*, *speed*)

Flick starting at on_element, and moving by the xoffset and yoffset with specified speed.

Args:
- on_element: Flick will start at center of element.
- xoffset: X offset to flick to.
- yoffset: Y offset to flick to.
- speed: Pixels per second to flick.

`long_press`(*on_element*)

Long press on an element.

Args:
- on_element: The element to long press.

move(*xcoord*, *ycoord*)
Move held tap to specified location.

Args:
- xcoord: X Coordinate to move.
- ycoord: Y Coordinate to move.

perform()
Performs all stored actions.

release(*xcoord*, *ycoord*)
Release previously issued tap 'and hold' command at specified location.

Args:
- xcoord: X Coordinate to release.
- ycoord: Y Coordinate to release.

scroll(*xoffset*, *yoffset*)
Touch and scroll, moving by xoffset and yoffset.

Args:
- xoffset: X offset to scroll to.
- yoffset: Y offset to scroll to.

scroll_from_element(*on_element*, *xoffset*, *yoffset*)
Touch and scroll starting at on_element, moving by xoffset and yoffset.

Args:
- on_element: The element where scroll starts.
- xoffset: X offset to scroll to.
- yoffset: Y offset to scroll to.

tap(*on_element*)
Taps on a given element.

Args:
- on_element: The element to tap.

tap_and_hold(*xcoord*, *ycoord*)
Touch down at given coordinates.

Args:
- xcoord: X Coordinate to touch down.
- ycoord: Y Coordinate to touch down.

7.8. Proxy

The Proxy implementation.

class selenium.webdriver.common.proxy.**Proxy**(*raw=None*)
Bases: **object**

Proxy contains information about proxy type and necessary proxy settings.

__init__(*raw=None*)
Creates a new Proxy.

Args:
- raw: raw proxy data. If None, default class values are used.

add_to_capabilities(*capabilities*)
Adds proxy information as capability in specified capabilities.

Args: • capabilities: The capabilities to which proxy will be added.

auto_detect
Returns autodetect setting.

autodetect = *False*

ftpProxy = ''

ftp_proxy
Returns ftp proxy setting.

httpProxy = ''

http_proxy
Returns http proxy setting.

noProxy = ''

no_proxy
Returns noproxy setting.

proxyAutoconfigUrl = ''

proxyType = *{'ff_value': 6, 'string': 'UNSPECIFIED'}*

proxy_autoconfig_url
Returns proxy autoconfig url setting.

proxy_type
Returns proxy type as *ProxyType*.

socksPassword = ''

socksProxy = ''

socksUsername = ''

socks_password
Returns socks proxy password setting.

socks_proxy
Returns socks proxy setting.

socks_username
Returns socks proxy username setting.

sslProxy = ''

ssl_proxy
Returns https proxy setting.

class selenium.webdriver.common.proxy.**ProxyType**
Bases: **object**

Set of possible types of proxy.

Each proxy type has 2 properties:
'ff_value' is value of Firefox profile preference, 'string' is id of proxy type.

classmethod **load**(*value*)

AUTODETECT = *{'ff_value': 4, 'string': 'AUTODETECT'}*

DIRECT = *{'ff_value': 0, 'string': 'DIRECT'}*

MANUAL = *{'ff_value': 1, 'string': 'MANUAL'}*

PAC = *{'ff_value': 2, 'string': 'PAC'}*

RESERVED_1 = *{'ff_value': 3, 'string': 'RESERVED1'}*

SYSTEM = *{'ff_value': 5, 'string': 'SYSTEM'}*

UNSPECIFIED = *{'ff_value': 6, 'string': 'UNSPECIFIED'}*

class selenium.webdriver.common.proxy.**ProxyTypeFactory**

Bases: **object**

Factory for proxy types.

static **make**(*ff_value*, *string*)

7.9. Utilities

The Utils methods.

selenium.webdriver.common.utils.**find_connectable_ip**(*host*, *port=None*)

Resolve a hostname to an IP, preferring IPv4 addresses.

We prefer IPv4 so that we don't change behavior from previous IPv4-only implementations, and because some drivers (e.g., FirefoxDriver) do not support IPv6 connections.

If the optional port number is provided, only IPs that listen on the given port are considered.

Args:
- host - A hostname.
- port - Optional port number.

Returns: A single IP address, as a string. If any IPv4 address is found, one is returned. Otherwise, if any IPv6 address is found, one is returned. If neither, then None is returned.

selenium.webdriver.common.utils.**free_port**()

Determines a free port using sockets.

selenium.webdriver.common.utils.**is_connectable**(*port*, *host='localhost'*)

Tries to connect to the server at port to see if it is running.

Args:
- port - The port to connect.

selenium.webdriver.common.utils.**is_url_connectable**(*port*)

Tries to connect to the HTTP server at /status path and specified port to see if it responds successfully.

Args:
- port - The port to connect.

selenium.webdriver.common.utils.**join_host_port**(*host*, *port*)

Joins a hostname and port together.

This is a minimal implementation intended to cope with IPv6 literals. For example, _join_host_port('::1', 80) == '[::1]:80'.

Args:
- host - A hostname.
- port - An integer port.

selenium.webdriver.common.utils.**keys_to_typing**(*value*)

Processes the values that will be typed in the element.

7.10. Service

class selenium.webdriver.common.service.**Service**(*executable*, *port=0*, *log_file=-3*, *env=None*, *start_error_message=''*)

Bases: **object**

__init__(*executable*, *port=0*, *log_file=-3*, *env=None*, *start_error_message=''*)

Initialize self. See help(type(self)) for accurate signature.

assert_process_still_running()

command_line_args()

is_connectable()

send_remote_shutdown_command()

start()

Starts the Service.

Exceptions:
- WebDriverException : Raised either when it can't start the service or when it can't connect to the service

stop()

Stops the service.

service_url

Gets the url of the Service

7.11. Application Cache

The ApplicationCache implementaion.

class selenium.webdriver.common.html5.application_cache.**ApplicationCache**(*driver*)

Bases: **object**

__init__(*driver*)

Creates a new Aplication Cache.

Args:
- driver: The WebDriver instance which performs user actions.

CHECKING = *2*

DOWNLOADING = *3*

IDLE = *1*

OBSOLETE = *5*

UNCACHED = *0*

UPDATE_READY = *4*

status

Returns a current status of application cache.

7.12. Firefox WebDriver

class `selenium.webdriver.firefox.webdriver.`**`WebDriver`**(*firefox_profile=None*, *firefox_binary=None*, *timeout=30*, *capabilities=None*, *proxy=None*, *executable_path='geckodriver'*, *options=None*, *service_log_path='geckodriver.log'*, *firefox_options=None*, *service_args=None*, *desired_capabilities=None*, *log_path=None*, *keep_alive=True*)

Bases: **`selenium.webdriver.remote.webdriver.WebDriver`**

`__init__`(*firefox_profile=None*, *firefox_binary=None*, *timeout=30*, *capabilities=None*, *proxy=None*, *executable_path='geckodriver'*, *options=None*, *service_log_path='geckodriver.log'*, *firefox_options=None*, *service_args=None*, *desired_capabilities=None*, *log_path=None*, *keep_alive=True*)

Starts a new local session of Firefox.

Based on the combination and specificity of the various keyword arguments, a capabilities dictionary will be constructed that is passed to the remote end.

The keyword arguments given to this constructor are helpers to more easily allow Firefox WebDriver sessions to be customised with different options. They are mapped on to a capabilities dictionary that is passed on to the remote end.

As some of the options, such as *firefox_profile* and *options.profile* are mutually exclusive, precedence is given from how specific the setting is. *capabilities* is the least specific keyword argument, followed by *options*, followed by *firefox_binary* and *firefox_profile*.

In practice this means that if *firefox_profile* and *options.profile* are both set, the selected profile instance will always come from the most specific variable. In this case that would be *firefox_profile*. This will result in *options.profile* to be ignored because it is considered a less specific setting than the top-level *firefox_profile* keyword argument. Similarily, if you had specified a *capabilities["moz:firefoxOptions"]["profile"]* Base64 string, this would rank below *options.profile*.

Parameters:

- **firefox_profile** – Instance of `FirefoxProfile` object or a string. If undefined, a fresh profile will be created in a temporary location on the system.
- **firefox_binary** – Instance of `FirefoxBinary` or full path to the Firefox binary. If undefined, the system default Firefox installation will be used.
- **timeout** – Time to wait for Firefox to launch when using the extension connection.
- **capabilities** – Dictionary of desired capabilities.
- **proxy** – The proxy settings to us when communicating with Firefox via the extension connection.
- **executable_path** – Full path to override which geckodriver binary to use for Firefox 47.0.1 and greater, which defaults to picking up the binary from the system path.
- **options** – Instance of `options.Options`.
- **service_log_path** – Where to log information from the driver.
- **firefox_options** – Deprecated argument for options
- **service_args** – List of args to pass to the driver service
- **desired_capabilities** – alias of capabilities. In future versions of this library, this will replace 'capabilities'. This will make the signature consistent with RemoteWebDriver.
- **log_path** – Deprecated argument for service_log_path
- **keep_alive** – Whether to configure remote_connection.RemoteConnection to use HTTP keep-alive.

context(*context*)

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

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

Usage example:

```
with selenium.context(selenium.CONTEXT_CHROME):
    # chrome scope
    ... do stuff ...
```

install_addon(*path*, *temporary=None*)

Installs Firefox addon.

Returns identifier of installed addon. This identifier can later be used to uninstall addon.

Parameters: **path** – Absolute path to the addon that will be installed.
Usage: driver.install_addon('/path/to/firebug.xpi')

quit()

Quits the driver and close every associated window.

set_context(*context*)

uninstall_addon(*identifier*)

Uninstalls Firefox addon using its identifier.

Usage: driver.uninstall_addon('addon@foo.com')

CONTEXT_CHROME = *'chrome'*

CONTEXT_CONTENT = *'content'*

NATIVE_EVENTS_ALLOWED = *True*

firefox_profile

7.13. Firefox WebDriver Options

class selenium.webdriver.firefox.options.**Log**

Bases: **object**

__init__()

Initialize self. See help(type(self)) for accurate signature.

to_capabilities()

class selenium.webdriver.firefox.options.**Options**

Bases: **object**

__init__()

Initialize self. See help(type(self)) for accurate signature.

add_argument(*argument*)

Add argument to be used for the browser process.

set_capability(*name*, *value*)
Sets a capability.

set_headless(*headless=True*)
Deprecated, options.headless = True

set_preference(*name*, *value*)
Sets a preference.

to_capabilities()
Marshals the Firefox options to a *moz:firefoxOptions* object.

KEY = *'moz:firefoxOptions'*

accept_insecure_certs

arguments
Returns a list of browser process arguments.

binary
Returns the FirefoxBinary instance

binary_location
Returns the location of the binary.

capabilities

headless
Returns whether or not the headless argument is set

preferences
Returns a dict of preferences.

profile
Returns the Firefox profile to use.

proxy
returns Proxy if set otherwise None.

7.14. Firefox WebDriver Profile

exception selenium.webdriver.firefox.firefox_profile.**AddonFormatError**
Bases: **Exception**

Exception for not well-formed add-on manifest files

class selenium.webdriver.firefox.firefox_profile.**FirefoxProfile**(*profile_directory=None*)
Bases: **object**

__init__(*profile_directory=None*)
Initialises a new instance of a Firefox Profile

Args:
- profile_directory: Directory of profile that you want to use. If a directory is passed in it will be cloned and the cloned directory will be used by the driver when instantiated. Thi_ defaults to None and will create a new directory when object is created.

add_extension(*extension='webdriver.xpi'*)

set_preference(*key*, *value*)

sets the preference that we want in the profile.

set_proxy(*proxy*)

update_preferences()

ANONYMOUS_PROFILE_NAME = *'WEBDRIVER_ANONYMOUS_PROFILE'*

DEFAULT_PREFERENCES = *None*

accept_untrusted_certs

assume_untrusted_cert_issuer

encoded

A zipped, base64 encoded string of profile directory for use with remote WebDriver JSON wire protocol

native_events_enabled

path

Gets the profile directory that is currently being used

port

Gets the port that WebDriver is working on

7.15. Firefox WebDriver Binary

class selenium.webdriver.firefox.firefox_binary.**FirefoxBinary**(*firefox_path=None*, *log_file=None*)

Bases: **object**

__init__(*firefox_path=None*, *log_file=None*)

Creates a new instance of Firefox binary.

Args:
- firefox_path - Path to the Firefox executable. By default, it will be detected from the standard locations.
- log_file - A file object to redirect the firefox process output to. It can be sys.stdout. Please note that with parallel run the output won't be synchronous. By default, it will be redirected to /dev/null.

add_command_line_options(**args*)

kill()

Kill the browser.

This is useful when the browser is stuck.

launch_browser(*profile*, *timeout=30*)

Launches the browser for the given profile name. It is assumed the profile already exists.

which(*fname*)

Returns the fully qualified path by searching Path of the given name

NO_FOCUS_LIBRARY_NAME = *'x_ignore_nofocus.so'*

7.16. Firefox WebDriver Extension Connection

exception selenium.webdriver.firefox.extension_connection.**ExtensionConnectionError**

Bases: **Exception**

An internal error occurred int the extension.

Might be caused by bad input or bugs in webdriver

class selenium.webdriver.firefox.extension_connection.**ExtensionConnection**(*host*, *firefox_profile*, *firefox_binary=None*, *timeout=30*)

Bases: **selenium.webdriver.remote.remote_connection.RemoteConnection**

__init__(*host*, *firefox_profile*, *firefox_binary=None*, *timeout=30*)

Initialize self. See help(type(self)) for accurate signature.

connect()

Connects to the extension and retrieves the session id.

classmethod **connect_and_quit**()

Connects to an running browser and quit immediately.

classmethod **is_connectable**()

Trys to connect to the extension but do not retrieve context.

quit(*sessionId=None*)

7.17. Chrome WebDriver

class selenium.webdriver.chrome.webdriver.**WebDriver**(*executable_path='chromedriver'*, *port=0*, *options=None*, *service_args=None*, *desired_capabilities=None*, *service_log_path=None*, *chrome_options=None*, *keep_alive=True*)

Bases: **selenium.webdriver.remote.webdriver.WebDriver**

Controls the ChromeDriver and allows you to drive the browser.

You will need to download the ChromeDriver executable from
http://chromedriver.storage.googleapis.com/index.html

__init__(*executable_path='chromedriver'*, *port=0*, *options=None*, *service_args=None*, *desired_capabilities=None*, *service_log_path=None*, *chrome_options=None*, *keep_alive=True*)

Creates a new instance of the chrome driver.

Starts the service and then creates new instance of chrome driver.

Args:

- executable_path - path to the executable. If the default is used it assumes the executable is in the $PATH
- port - port you would like the service to run, if left as 0, a free port will be found.
- options - this takes an instance of ChromeOptions
- service_args - List of args to pass to the driver service
- desired_capabilities - Dictionary object with non-browser specific capabilities only, such as “proxy” or “loggingPref”.
- service_log_path - Where to log information from the driver.
- chrome_options - Deprecated argument for options

- keep_alive - Whether to configure ChromeRemoteConnection to use HTTP keep-alive.

`create_options`()

`execute_cdp_cmd`(*cmd*, *cmd_args*)

Execute Chrome Devtools Protocol command and get returned result

The command and command args should follow chrome devtools protocol domains/commands, refer to link https://chromedevtools.github.io/devtools-protocol/

Args:
- cmd: A str, command name
- cmd_args: A dict, command args. empty dict {} if there is no command args

Usage: driver.execute_cdp_cmd('Network.getResponseBody', {'requestId': requestId})

Returns: A dict, empty dict {} if there is no result to return. For example to getResponseBody:

{'base64Encoded': False, 'body': 'response body string'}

`get_network_conditions`()

Gets Chrome network emulation settings.

Returns: A dict. For example:

{'latency': 4, 'download_throughput': 2, 'upload_throughput': 2, 'offline': False}

`launch_app`(*id*)

Launches Chrome app specified by id.

`quit`()

Closes the browser and shuts down the ChromeDriver executable that is started when starting the ChromeDriver

`set_network_conditions`(****network_conditions*)

Sets Chrome network emulation settings.

Args:
- network_conditions: A dict with conditions specification.

Usage: driver.set_network_conditions(
offline=False, latency=5, # additional latency (ms) download_throughput=500 * 1024, # maximal throughput upload_throughput=500 * 1024) # maximal throughput

Note: 'throughput' can be used to set both (for download and upload).

7.18. Chrome WebDriver Options

class `selenium.webdriver.chrome.options.`**`Options`**

Bases: `object`

`__init__`()

Initialize self. See help(type(self)) for accurate signature.

`add_argument`(*argument*)

Adds an argument to the list

Args:
- Sets the arguments

add_encoded_extension(*extension*)

Adds Base64 encoded string with extension data to a list that will be used to extract it to the ChromeDriver

Args:
- extension: Base64 encoded string with extension data

add_experimental_option(*name*, *value*)

Adds an experimental option which is passed to chrome.

Args:

name: The experimental option name. value: The option value.

add_extension(*extension*)

Adds the path to the extension to a list that will be used to extract it to the ChromeDriver

Args:
- extension: path to the *.crx file

set_capability(*name*, *value*)

Sets a capability.

set_headless(*headless=True*)

Deprecated, options.headless = True

to_capabilities()

Creates a capabilities with all the options that have been set and

returns a dictionary with everything

KEY = *'goog:chromeOptions'*

arguments

Returns a list of arguments needed for the browser

binary_location

Returns the location of the binary otherwise an empty string

capabilities

debugger_address

Returns the address of the remote devtools instance

experimental_options

Returns a dictionary of experimental options for chrome.

extensions

Returns a list of encoded extensions that will be loaded into chrome

headless

Returns whether or not the headless argument is set

7.19. Chrome WebDriver Service

class `selenium.webdriver.chrome.service.`**Service**(*executable_path*, *port=0*, *service_args=None*, *log_path=None*, *env=None*)

Bases: **selenium.webdriver.common.service.Service**

Object that manages the starting and stopping of the ChromeDriver

__init__(*executable_path*, *port=0*, *service_args=None*, *log_path=None*, *env=None*)

Creates a new instance of the Service

Args:
- executable_path : Path to the ChromeDriver
- port : Port the service is running on
- service_args : List of args to pass to the chromedriver service
- log_path : Path for the chromedriver service to log to

command_line_args()

7.20. Remote WebDriver

The WebDriver implementation.

class selenium.webdriver.remote.webdriver.**WebDriver**(*command_executor='http://127.0.0.1:4444/wd/hub'*, *desired_capabilities=None*, *browser_profile=None*, *proxy=None*, *keep_alive=False*, *file_detector=None*, *options=None*)

Bases: **object**

Controls a browser by sending commands to a remote server. This server is expected to be running the WebDriver wire protocol as defined at https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

Attributes:
- session_id - String ID of the browser session started and controlled by this WebDriver.
- capabilities - Dictionaty of effective capabilities of this browser session as returned by the remote server. See https://github.com/SeleniumHQ/selenium/wiki/DesiredCapabilities
- command_executor - remote_connection.RemoteConnection object used to execute commands.
- error_handler - errorhandler.ErrorHandler object used to handle errors.

__init__(*command_executor='http://127.0.0.1:4444/wd/hub'*, *desired_capabilities=None*, *browser_profile=None*, *proxy=None*, *keep_alive=False*, *file_detector=None*, *options=None*)

Create a new driver that will issue commands using the wire protocol.

Args:
- command_executor - Either a string representing URL of the remote server or a custom remote_connection.RemoteConnection object. Defaults to 'http://127.0.0.1:4444/wd/hub'.
- desired_capabilities - A dictionary of capabilities to request when starting the browser session. Required parameter.
- browser_profile - A selenium.webdriver.firefox.firefox_profile.FirefoxProfile object. Only used if Firefox is requested. Optional.
- proxy - A selenium.webdriver.common.proxy.Proxy object. The browser session will be started with given proxy settings, if possible. Optional.
- keep_alive - Whether to configure remote_connection.RemoteConnection to use HTTP keep-alive. Defaults to False.
- file_detector - Pass custom file detector object during instantiation. If None,

then default LocalFileDetector() will be used.

- options - instance of a driver options.Options class

`add_cookie`(*cookie_dict*)

Adds a cookie to your current session.

Args:
- cookie_dict: A dictionary object, with required keys - "name" and "value"; optional keys - "path", "domain", "secure", "expiry"

Usage:

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

`back`()

Goes one step backward in the browser history.

Usage: driver.back()

`close`()

Closes the current window.

Usage: driver.close()

`create_web_element`(*element_id*)

Creates a web element with the specified *element_id*.

`delete_all_cookies`()

Delete all cookies in the scope of the session.

Usage: driver.delete_all_cookies()

`delete_cookie`(*name*)

Deletes a single cookie with the given name.

Usage: driver.delete_cookie('my_cookie')

`execute`(*driver_command*, *params=None*)

Sends a command to be executed by a command.CommandExecutor.

Args:
- driver_command: The name of the command to execute as a string.
- params: A dictionary of named parameters to send with the command.

Returns: The command's JSON response loaded into a dictionary object.

`execute_async_script`(*script*, **args*)

Asynchronously Executes JavaScript in the current window/frame.

Args:
- script: The JavaScript to execute.
- *args: Any applicable arguments for your JavaScript.

Usage: script = "var callback = arguments[arguments.length - 1]; " "window.setTimeout(function(){ callback('timeout') }, 3000);" driver.execute_async_script(script)

`execute_script`(*script*, **args*)

Synchronously Executes JavaScript in the current window/frame.

Args:
- script: The JavaScript to execute.
- *args: Any applicable arguments for your JavaScript.

Usage: driver.execute_script('return document.title;')

`file_detector_context`(*file_detector_class*, **args*, ***kwargs*)

Overrides the current file detector (if necessary) in limited context. Ensures the original file detector is set afterwards.

Example:

with webdriver.file_detector_context(UselessFileDetector):
    someinput.send_keys('/etc/hosts')

Args:
- file_detector_class - Class of the desired file detector. If the class is different from the current file_detector, then the class is instantiated with args and kwargs and used as a file detector during the duration of the context manager.
- args - Optional arguments that get passed to the file detector class during instantiation.
- kwargs - Keyword arguments, passed the same way as args.

`find_element`(*by='id'*, *value=None*)

Find an element given a By strategy and locator. Prefer the find_element_by_* methods when possible.

Usage: element = driver.find_element(By.ID, 'foo')
Return type: WebElement

`find_element_by_class_name`(*name*)

Finds an element by class name.

Args:
- name: The class name of the element to find.

Returns:
- WebElement - the element if it was found

Raises:
- NoSuchElementException - if the element wasn't found

Usage: element = driver.find_element_by_class_name('foo')

`find_element_by_css_selector`(*css_selector*)

Finds an element by css selector.

Args:
- css_selector - CSS selector string, ex: 'a.nav#home'

Returns:
- WebElement - the element if it was found

Raises:
- NoSuchElementException - if the element wasn't found

Usage: element = driver.find_element_by_css_selector('#foo')

`find_element_by_id`(*id_*)

Finds an element by id.

Args:
- id_ - The id of the element to be found.

Returns:
- WebElement - the element if it was found

Raises:
- NoSuchElementException - if the element wasn't found

Usage: element = driver.find_element_by_id('foo')

`find_element_by_link_text`(*link_text*)

Finds an element by link text.

Args: • link_text: The text of the element to be found.
Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = driver.find_element_by_link_text('Sign In')

`find_element_by_name`(*name*)

Finds an element by name.

Args: • name: The name of the element to find.
Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = driver.find_element_by_name('foo')

`find_element_by_partial_link_text`(*link_text*)

Finds an element by a partial match of its link text.

Args: • link_text: The text of the element to partially match on.
Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = driver.find_element_by_partial_link_text('Sign')

`find_element_by_tag_name`(*name*)

Finds an element by tag name.

Args: • name - name of html tag (eg: h1, a, span)
Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = driver.find_element_by_tag_name('h1')

`find_element_by_xpath`(*xpath*)

Finds an element by xpath.

Args: • xpath - The xpath locator of the element to find.
Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = driver.find_element_by_xpath('//div/td[1]')

`find_elements`(*by='id'*, *value=None*)

Find elements given a By strategy and locator. Prefer the find_elements_by_* methods when possible.

Usage: elements = driver.find_elements(By.CLASS_NAME, 'foo')
Return type: list of WebElement

`find_elements_by_class_name`(*name*)

Finds elements by class name.

Args: • name: The class name of the elements to find.
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = driver.find_elements_by_class_name('foo')

`find_elements_by_css_selector`(*css_selector*)

Finds elements by css selector.

Args: • css_selector - CSS selector string, ex: 'a.nav#home'
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = driver.find_elements_by_css_selector('.foo')

`find_elements_by_id`(*id_*)

Finds multiple elements by id.

Args: • id_ - The id of the elements to be found.
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = driver.find_elements_by_id('foo')

`find_elements_by_link_text`(*text*)

Finds elements by link text.

Args: • link_text: The text of the elements to be found.
Returns: • list of webelement - a list with elements if any was found. an empty list if not
Usage: elements = driver.find_elements_by_link_text('Sign In')

`find_elements_by_name`(*name*)

Finds elements by name.

Args: • name: The name of the elements to find.
Returns: • list of webelement - a list with elements if any was found. an empty list if not
Usage: elements = driver.find_elements_by_name('foo')

`find_elements_by_partial_link_text`(*link_text*)

Finds elements by a partial match of their link text.

Args: • link_text: The text of the element to partial match on.
Returns: • list of webelement - a list with elements if any was found. an empty list if not
Usage: elements = driver.find_elements_by_partial_link_text('Sign')

`find_elements_by_tag_name`(*name*)

Finds elements by tag name.

Args: • name - name of html tag (eg: h1, a, span)
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = driver.find_elements_by_tag_name('h1')

`find_elements_by_xpath`(*xpath*)

Finds multiple elements by xpath.

Args: • xpath - The xpath locator of the elements to be found.
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = driver.find_elements_by_xpath("//div[contains(@class, 'foo')]")

`forward`()

Goes one step forward in the browser history.

Usage: driver.forward()

`fullscreen_window`()

Invokes the window manager-specific 'full screen' operation

`get`(*url*)

Loads a web page in the current browser session.

`get_cookie`(*name*)

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

Usage: driver.get_cookie('my_cookie')

`get_cookies`()

Returns a set of dictionaries, corresponding to cookies visible in the current session.

Usage: driver.get_cookies()

`get_log`(*log_type*)

Gets the log for a given log type

Args:
- log_type: type of log that which will be returned

Usage: driver.get_log('browser') driver.get_log('driver') driver.get_log('client') driver.get_log('server')

`get_screenshot_as_base64`()

Gets the screenshot of the current window as a base64 encoded string
which is useful in embedded images in HTML.

Usage: driver.get_screenshot_as_base64()

`get_screenshot_as_file`(*filename*)

Saves a screenshot of the current window to a PNG image file. Returns
False if there is any IOError, else returns True. Use full paths in your filename.

Args:
- filename: The full path you wish to save your screenshot to. This should end with a *.png* extension.

Usage: driver.get_screenshot_as_file('/Screenshots/foo.png')

`get_screenshot_as_png`()

Gets the screenshot of the current window as a binary data.

Usage: driver.get_screenshot_as_png()

`get_window_position`(*windowHandle='current'*)

Gets the x,y position of the current window.

Usage: driver.get_window_position()

`get_window_rect`()

Gets the x, y coordinates of the window as well as height and width of the current window.

Usage: driver.get_window_rect()

`get_window_size`(*windowHandle='current'*)

Gets the width and height of the current window.

Usage: driver.get_window_size()

`implicitly_wait`(*time_to_wait*)

Sets a sticky timeout to implicitly wait for an element to be found,

or a command to complete. This method only needs to be called one time per session. To set the timeout for calls to execute_async_script, see set_script_timeout.

Args:
- time_to_wait: Amount of time to wait (in seconds)

Usage: driver.implicitly_wait(30)

`maximize_window`()

Maximizes the current window that webdriver is using

`minimize_window`()

Invokes the window manager-specific 'minimize' operation

`quit`()

Quits the driver and closes every associated window.

Usage: driver.quit()

`refresh`()

Refreshes the current page.

Usage: driver.refresh()

`save_screenshot`(*filename*)

Saves a screenshot of the current window to a PNG image file. Returns False if there is any IOError, else returns True. Use full paths in your filename.

Args:
- filename: The full path you wish to save your screenshot to. This should end with a *.png* extension.

Usage: driver.save_screenshot('/Screenshots/foo.png')

`set_page_load_timeout`(*time_to_wait*)

Set the amount of time to wait for a page load to complete before throwing an error.

Args:
- time_to_wait: The amount of time to wait

Usage: driver.set_page_load_timeout(30)

`set_script_timeout`(*time_to_wait*)

Set the amount of time that the script should wait during an execute_async_script call before throwing an error.

Args:
- time_to_wait: The amount of time to wait (in seconds)

Usage: driver.set_script_timeout(30)

`set_window_position`(*x*, *y*, *windowHandle='current'*)

Sets the x,y position of the current window. (window.moveTo)

Args:
- x: the x-coordinate in pixels to set the window position
- y: the y-coordinate in pixels to set the window position

Usage: driver.set_window_position(0,0)

`set_window_rect`(*x=None*, *y=None*, *width=None*, *height=None*)

Sets the x, y coordinates of the window as well as height and width of the current window.

Usage: driver.set_window_rect(x=10, y=10) driver.set_window_rect(width=100, height=200)

driver.set_window_rect(x=10, y=10, width=100, height=200)

set_window_size(*width*, *height*, *windowHandle='current'*)

Sets the width and height of the current window. (window.resizeTo)

Args:
- width: the width in pixels to set the window to
- height: the height in pixels to set the window to

Usage: driver.set_window_size(800,600)

start_client()

Called before starting a new session. This method may be overridden to define custom startup behavior.

start_session(*capabilities*, *browser_profile=None*)

Creates a new session with the desired capabilities.

Args:
- browser_name - The name of the browser to request.
- version - Which browser version to request.
- platform - Which platform to request the browser on.
- javascript_enabled - Whether the new session should support JavaScript.
- browser_profile - A selenium.webdriver.firefox.firefox_profile.FirefoxProfile object. Only used if Firefox is requested.

stop_client()

Called after executing a quit command. This method may be overridden to define custom shut-down behavior.

switch_to_active_element()

Deprecated use driver.switch_to.active_element

switch_to_alert()

Deprecated use driver.switch_to.alert

switch_to_default_content()

Deprecated use driver.switch_to.default_content

switch_to_frame(*frame_reference*)

Deprecated use driver.switch_to.frame

switch_to_window(*window_name*)

Deprecated use driver.switch_to.window

application_cache

Returns a ApplicationCache Object to interact with the browser app cache

current_url

Gets the URL of the current page.

Usage: driver.current_url

current_window_handle

Returns the handle of the current window.

Usage: driver.current_window_handle

desired_capabilities

returns the drivers current desired capabilities being used

file_detector

log_types

Gets a list of the available log types

Usage: driver.log_types

mobile

name

Returns the name of the underlying browser for this instance.

Usage: name = driver.name

orientation

Gets the current orientation of the device

Usage: orientation = driver.orientation

page_source

Gets the source of the current page.

Usage: driver.page_source

switch_to

Returns: • SwitchTo: an object containing all options to switch focus into

Usage: element = driver.switch_to.active_element alert = driver.switch_to.alert driver.switch_to.default_content() driver.switch_to.frame('frame_name') driver.switch_to.frame(1) driver.switch_to.frame(driver.find_elements_by_tag_name("iframe")[0]) driver.switch_to.parent_frame() driver.switch_to.window('main')

title

Returns the title of the current page.

Usage: title = driver.title

window_handles

Returns the handles of all windows within the current session.

Usage: driver.window_handles

7.21. Remote WebDriver WebElement

class selenium.webdriver.remote.webelement.**WebElement**(*parent*, *id_*, *w3c=False*)

Bases: **object**

Represents a DOM element.

Generally, all interesting operations that interact with a document will be performed through this interface.

All method calls will do a freshness check to ensure that the element reference is still valid. This essen tially determines whether or not the element is still attached to the DOM. If this test fails, then an

`StaleElementReferenceException` is thrown, and all future calls to this instance will fail.

__init__(*parent*, *id_*, *w3c=False*)

Initialize self. See help(type(self)) for accurate signature.

clear()

Clears the text if it's a text entry element.

click()

Clicks the element.

find_element(*by='id'*, *value=None*)

Find an element given a By strategy and locator. Prefer the find_element_by_* methods when possible.

Usage: element = element.find_element(By.ID, 'foo')
Return type: WebElement

find_element_by_class_name(*name*)

Finds element within this element's children by class name.

Args:
- name: The class name of the element to find.

Returns:
- WebElement - the element if it was found

Raises:
- NoSuchElementException - if the element wasn't found

Usage: element = element.find_element_by_class_name('foo')

find_element_by_css_selector(*css_selector*)

Finds element within this element's children by CSS selector.

Args:
- css_selector - CSS selector string, ex: 'a.nav#home'

Returns:
- WebElement - the element if it was found

Raises:
- NoSuchElementException - if the element wasn't found

Usage: element = element.find_element_by_css_selector('#foo')

find_element_by_id(*id_*)

Finds element within this element's children by ID.

Args:
- id_ - ID of child element to locate.

Returns:
- WebElement - the element if it was found

Raises:
- NoSuchElementException - if the element wasn't found

Usage: foo_element = element.find_element_by_id('foo')

find_element_by_link_text(*link_text*)

Finds element within this element's children by visible link text.

Args:
- link_text - Link text string to search for.

Returns:
- WebElement - the element if it was found

Raises:
- NoSuchElementException - if the element wasn't found

Usage: element = element.find_element_by_link_text('Sign In')

find_element_by_name(*name*)

Finds element within this element's children by name.

Args: • name - name property of the element to find.
Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = element.find_element_by_name('foo')

`find_element_by_partial_link_text`(*link_text*)

Finds element within this element's children by partially visible link text.

Args: • link_text: The text of the element to partially match on.
Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = element.find_element_by_partial_link_text('Sign')

`find_element_by_tag_name`(*name*)

Finds element within this element's children by tag name.

Args: • name - name of html tag (eg: h1, a, span)
Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = element.find_element_by_tag_name('h1')

`find_element_by_xpath`(*xpath*)

Finds element by xpath.

Args: • xpath - xpath of element to locate. "//input[@class='myelement']"

Note: The base path will be relative to this element's location.

This will select the first link under this element.

```
myelement.find_element_by_xpath(".//a")
```

However, this will select the first link on the page.

```
myelement.find_element_by_xpath("//a")
```

Returns: • WebElement - the element if it was found
Raises: • NoSuchElementException - if the element wasn't found
Usage: element = element.find_element_by_xpath('//div/td[1]')

`find_elements`(*by='id'*, *value=None*)

Find elements given a By strategy and locator. Prefer the find_elements_by_* methods when possible.

Usage: element = element.find_elements(By.CLASS_NAME, 'foo')
Return type: list of WebElement

`find_elements_by_class_name`(*name*)

Finds a list of elements within this element's children by class name.

Args: • name: The class name of the elements to find.
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = element.find_elements_by_class_name('foo')

`find_elements_by_css_selector`(*css_selector*)

Finds a list of elements within this element's children by CSS selector.

Args: • css_selector - CSS selector string, ex: 'a.nav#home'
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = element.find_elements_by_css_selector('.foo')

`find_elements_by_id`(*id_*)

Finds a list of elements within this element's children by ID. Will return a list of webelements if found, or an empty list if not.

Args: • id_ - Id of child element to find.
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = element.find_elements_by_id('foo')

`find_elements_by_link_text`(*link_text*)

Finds a list of elements within this element's children by visible link text.

Args: • link_text - Link text string to search for.
Returns: • list of webelement - a list with elements if any was found. an empty list if not
Usage: elements = element.find_elements_by_link_text('Sign In')

`find_elements_by_name`(*name*)

Finds a list of elements within this element's children by name.

Args: • name - name property to search for.
Returns: • list of webelement - a list with elements if any was found. an empty list if not
Usage: elements = element.find_elements_by_name('foo')

`find_elements_by_partial_link_text`(*link_text*)

Finds a list of elements within this element's children by link text.

Args: • link_text: The text of the element to partial match on.
Returns: • list of webelement - a list with elements if any was found. an empty list if not
Usage: elements = element.find_elements_by_partial_link_text('Sign')

`find_elements_by_tag_name`(*name*)

Finds a list of elements within this element's children by tag name.

Args: • name - name of html tag (eg: h1, a, span)
Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = element.find_elements_by_tag_name('h1')

`find_elements_by_xpath`(*xpath*)

Finds elements within the element by xpath.

Args: • xpath - xpath locator string.

Note: The base path will be relative to this element's location.

This will select all links under this element.


```
myelement.find_elements_by_xpath(".//a")
```

However, this will select all links in the page itself.

```
myelement.find_elements_by_xpath("//a")
```

Returns: • list of WebElement - a list with elements if any was found. An empty list if not
Usage: elements = element.find_elements_by_xpath("//div[contains(@class, 'foo')]")

get_attribute(*name*)

Gets the given attribute or property of the element.

This method will first try to return the value of a property with the given name. If a property with that name doesn't exist, it returns the value of the attribute with the same name. If there's no attribute with that name, `None` is returned.

Values which are considered truthy, that is equals "true" or "false", are returned as booleans. All other non-`None` values are returned as strings. For attributes or properties which do not exist, `None` is returned.

Args: • name - Name of the attribute/property to retrieve.

Example:

```
# Check if the "active" CSS class is applied to an element.
is_active = "active" in target_element.get_attribute("class")
```

get_property(*name*)

Gets the given property of the element.

Args: • name - Name of the property to retrieve.

Example:

```
text_length = target_element.get_property("text_length")
```

is_displayed()

Whether the element is visible to a user.

is_enabled()

Returns whether the element is enabled.

is_selected()

Returns whether the element is selected.

Can be used to check if a checkbox or radio button is selected.

screenshot(*filename*)

Saves a screenshot of the current element to a PNG image file. Returns
False if there is any IOError, else returns True. Use full paths in your filename.

Args: • filename: The full path you wish to save your screenshot to. This should end with a *.png* extension.
Usage: element.screenshot('/Screenshots/foo.png')

send_keys(**value*)

Simulates typing into the element.

Args:
- value - A string for typing, or setting form fields. For setting file inputs, this could be a local file path.

Use this to send simple key events or to fill out form fields:

```
form_textfield = driver.find_element_by_name('username')
form_textfield.send_keys("admin")
```

This can also be used to set file inputs.

```
file_input = driver.find_element_by_name('profilePic')
file_input.send_keys("path/to/profilepic.gif")
# Generally it's better to wrap the file path in one of the methods
# in os.path to return the actual path to support cross OS testing.
# file_input.send_keys(os.path.abspath("path/to/profilepic.gif"))
```

submit()

Submits a form.

value_of_css_property(*property_name*)

The value of a CSS property.

id

Internal ID used by selenium.

This is mainly for internal use. Simple use cases such as checking if 2 webelements refer to the same element, can be done using `==`:

```
if element1 == element2:
    print("These 2 are equal")
```

location

The location of the element in the renderable canvas.

location_once_scrolled_into_view

THIS PROPERTY MAY CHANGE WITHOUT WARNING. Use this to discover where on the screen an element is so that we can click it. This method should cause the element to be scrolled into view.

Returns the top lefthand corner location on the screen, or `None` if the element is not visible.

parent

Internal reference to the WebDriver instance this element was found from.

rect

A dictionary with the size and location of the element.

screenshot_as_base64

Gets the screenshot of the current element as a base64 encoded string.

Usage: img_b64 = element.screenshot_as_base64

screenshot_as_png

Gets the screenshot of the current element as a binary data.

Usage: element_png = element.screenshot_as_png

size
The size of the element.

tag_name
This element's `tagName` property.

text
The text of the element.

7.22. Remote WebDriver Command

class `selenium.webdriver.remote.command.`**Command**
Bases: **object**

Defines constants for the standard WebDriver commands.

While these constants have no meaning in and of themselves, they are used to marshal commands through a service that implements WebDriver's remote wire protocol:

https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

ACCEPT_ALERT = *'acceptAlert'*

ADD_COOKIE = *'addCookie'*

CLEAR_APP_CACHE = *'clearAppCache'*

CLEAR_ELEMENT = *'clearElement'*

CLEAR_LOCAL_STORAGE = *'clearLocalStorage'*

CLEAR_SESSION_STORAGE = *'clearSessionStorage'*

CLICK = *'mouseClick'*

CLICK_ELEMENT = *'clickElement'*

CLOSE = *'close'*

CONTEXT_HANDLES = *'getContextHandles'*

CURRENT_CONTEXT_HANDLE = *'getCurrentContextHandle'*

DELETE_ALL_COOKIES = *'deleteAllCookies'*

DELETE_COOKIE = *'deleteCookie'*

DELETE_SESSION = *'deleteSession'*

DISMISS_ALERT = *'dismissAlert'*

DOUBLE_CLICK = *'mouseDoubleClick'*

DOUBLE_TAP = *'touchDoubleTap'*

ELEMENT_SCREENSHOT = *'elementScreenshot'*

EXECUTE_ASYNC_SCRIPT = *'executeAsyncScript'*

EXECUTE_SCRIPT = *'executeScript'*

EXECUTE_SQL = *'executeSql'*

FIND_CHILD_ELEMENT = *'findChildElement'*

FIND_CHILD_ELEMENTS = *'findChildElements'*

FIND_ELEMENT = *'findElement'*

FIND_ELEMENTS = *'findElements'*

FLICK = *'touchFlick'*

FULLSCREEN_WINDOW = *'fullscreenWindow'*

GET = *'get'*

GET_ACTIVE_ELEMENT = *'getActiveElement'*

GET_ALERT_TEXT = *'getAlertText'*

GET_ALL_COOKIES = *'getCookies'*

GET_ALL_SESSIONS = *'getAllSessions'*

GET_APP_CACHE = *'getAppCache'*

GET_APP_CACHE_STATUS = *'getAppCacheStatus'*

GET_AVAILABLE_LOG_TYPES = *'getAvailableLogTypes'*

GET_COOKIE = *'getCookie'*

GET_CURRENT_URL = *'getCurrentUrl'*

GET_CURRENT_WINDOW_HANDLE = *'getCurrentWindowHandle'*

GET_ELEMENT_ATTRIBUTE = *'getElementAttribute'*

GET_ELEMENT_LOCATION = *'getElementLocation'*

GET_ELEMENT_LOCATION_ONCE_SCROLLED_INTO_VIEW = *'getElementLocationOnceScrolledIntoView'*

GET_ELEMENT_PROPERTY = *'getElementProperty'*

GET_ELEMENT_RECT = *'getElementRect'*

GET_ELEMENT_SIZE = *'getElementSize'*

GET_ELEMENT_TAG_NAME = *'getElementTagName'*

GET_ELEMENT_TEXT = *'getElementText'*

GET_ELEMENT_VALUE = *'getElementValue'*

GET_ELEMENT_VALUE_OF_CSS_PROPERTY = *'getElementValueOfCssProperty'*

GET_LOCAL_STORAGE_ITEM = *'getLocalStorageItem'*

GET_LOCAL_STORAGE_KEYS = *'getLocalStorageKeys'*

GET_LOCAL_STORAGE_SIZE = *'getLocalStorageSize'*

GET_LOCATION = *'getLocation'*

GET_LOG = *'getLog'*

GET_NETWORK_CONNECTION = *'getNetworkConnection'*

GET_PAGE_SOURCE = *'getPageSource'*

GET_SCREEN_ORIENTATION = *'getScreenOrientation'*

GET_SESSION_STORAGE_ITEM = *'getSessionStorageItem'*

GET_SESSION_STORAGE_KEYS = *'getSessionStorageKeys'*

GET_SESSION_STORAGE_SIZE = *'getSessionStorageSize'*

GET_TITLE = *'getTitle'*

GET_WINDOW_HANDLES = *'getWindowHandles'*

GET_WINDOW_POSITION = *'getWindowPosition'*

GET_WINDOW_RECT = *'getWindowRect'*

GET_WINDOW_SIZE = *'getWindowSize'*

GO_BACK = *'goBack'*

GO_FORWARD = *'goForward'*

IMPLICIT_WAIT = *'implicitlyWait'*

IS_ELEMENT_DISPLAYED = *'isElementDisplayed'*

IS_ELEMENT_ENABLED = *'isElementEnabled'*

IS_ELEMENT_SELECTED = *'isElementSelected'*

LONG_PRESS = *'touchLongPress'*

MAXIMIZE_WINDOW = *'windowMaximize'*

MINIMIZE_WINDOW = *'minimizeWindow'*

MOUSE_DOWN = *'mouseButtonDown'*

MOUSE_UP = *'mouseButtonUp'*

MOVE_TO = *'mouseMoveTo'*

NEW_SESSION = *'newSession'*

QUIT = *'quit'*

REFRESH = *'refresh'*

REMOVE_LOCAL_STORAGE_ITEM = *'removeLocalStorageItem'*

REMOVE_SESSION_STORAGE_ITEM = *'removeSessionStorageItem'*

SCREENSHOT = *'screenshot'*

SEND_KEYS_TO_ACTIVE_ELEMENT = *'sendKeysToActiveElement'*

SEND_KEYS_TO_ELEMENT = *'sendKeysToElement'*

SET_ALERT_CREDENTIALS = *'setAlertCredentials'*

SET_ALERT_VALUE = *'setAlertValue'*

SET_ELEMENT_SELECTED = *'setElementSelected'*

SET_LOCAL_STORAGE_ITEM = *'setLocalStorageItem'*

SET_LOCATION = *'setLocation'*

SET_NETWORK_CONNECTION = *'setNetworkConnection'*

SET_SCREEN_ORIENTATION = *'setScreenOrientation'*

SET_SCRIPT_TIMEOUT = *'setScriptTimeout'*

SET_SESSION_STORAGE_ITEM = *'setSessionStorageItem'*

SET_TIMEOUTS = *'setTimeouts'*

SET_WINDOW_POSITION = *'setWindowPosition'*

SET_WINDOW_RECT = *'setWindowRect'*

SET_WINDOW_SIZE = *'setWindowSize'*

SINGLE_TAP = *'touchSingleTap'*

STATUS = *'status'*

SUBMIT_ELEMENT = *'submitElement'*

SWITCH_TO_CONTEXT = *'switchToContext'*

SWITCH_TO_FRAME = *'switchToFrame'*

SWITCH_TO_PARENT_FRAME = *'switchToParentFrame'*

SWITCH_TO_WINDOW = *'switchToWindow'*

TOUCH_DOWN = *'touchDown'*

TOUCH_MOVE = *'touchMove'*

TOUCH_SCROLL = *'touchScroll'*

TOUCH_UP = *'touchUp'*

UPLOAD_FILE = *'uploadFile'*

W3C_ACCEPT_ALERT = *'w3cAcceptAlert'*

W3C_ACTIONS = *'actions'*

W3C_CLEAR_ACTIONS = *'clearActionState'*

W3C_DISMISS_ALERT = *'w3cDismissAlert'*

W3C_EXECUTE_SCRIPT = *'w3cExecuteScript'*

W3C_EXECUTE_SCRIPT_ASYNC = *'w3cExecuteScriptAsync'*

W3C_GET_ACTIVE_ELEMENT = *'w3cGetActiveElement'*

W3C_GET_ALERT_TEXT = *'w3cGetAlertText'*

W3C_GET_CURRENT_WINDOW_HANDLE = *'w3cGetCurrentWindowHandle'*

W3C_GET_WINDOW_HANDLES = *'w3cGetWindowHandles'*

W3C_GET_WINDOW_POSITION = *'w3cGetWindowPosition'*

W3C_GET_WINDOW_SIZE = *'w3cGetWindowSize'*

W3C_MAXIMIZE_WINDOW = *'w3cMaximizeWindow'*

W3C_SET_ALERT_VALUE = *'w3cSetAlertValue'*

W3C_SET_WINDOW_POSITION = *'w3cSetWindowPosition'*

W3C_SET_WINDOW_SIZE = *'w3cSetWindowSize'*

7.23. Remote WebDriver Error Handler

class `selenium.webdriver.remote.errorhandler.`**`ErrorCode`**

Bases: **`object`**

Error codes defined in the WebDriver wire protocol.

ELEMENT_CLICK_INTERCEPTED = *[64, 'element click intercepted']*

ELEMENT_IS_NOT_SELECTABLE = *[15, 'element not selectable']*

ELEMENT_NOT_INTERACTABLE = *[60, 'element not interactable']*

ELEMENT_NOT_VISIBLE = *[11, 'element not visible']*

IME_ENGINE_ACTIVATION_FAILED = *[31, 'ime engine activation failed']*

IME_NOT_AVAILABLE = *[30, 'ime not available']*

INSECURE_CERTIFICATE = *['insecure certificate']*

INVALID_ARGUMENT = *[61, 'invalid argument']*

INVALID_COOKIE_DOMAIN = *[24, 'invalid cookie domain']*

INVALID_COORDINATES = *['invalid coordinates']*

INVALID_ELEMENT_COORDINATES = *[29, 'invalid element coordinates']*

INVALID_ELEMENT_STATE = *[12, 'invalid element state']*

INVALID_SELECTOR = *[32, 'invalid selector']*

INVALID_SESSION_ID = *['invalid session id']*

INVALID_XPATH_SELECTOR = *[51, 'invalid selector']*

INVALID_XPATH_SELECTOR_RETURN_TYPER = *[52, 'invalid selector']*

JAVASCRIPT_ERROR = *[17, 'javascript error']*

METHOD_NOT_ALLOWED = *[405, 'unsupported operation']*

MOVE_TARGET_OUT_OF_BOUNDS = *[34, 'move target out of bounds']*

NO_ALERT_OPEN = *[27, 'no such alert']*

NO_SUCH_COOKIE = *[62, 'no such cookie']*

NO_SUCH_ELEMENT = *[7, 'no such element']*

NO_SUCH_FRAME = *[8, 'no such frame']*

NO_SUCH_WINDOW = *[23, 'no such window']*

SCRIPT_TIMEOUT = *[28, 'script timeout']*

SESSION_NOT_CREATED = *[33, 'session not created']*

STALE_ELEMENT_REFERENCE = *[10, 'stale element reference']*

SUCCESS = *0*

TIMEOUT = *[21, 'timeout']*

UNABLE_TO_CAPTURE_SCREEN = *[63, 'unable to capture screen']*

UNABLE_TO_SET_COOKIE = *[25, 'unable to set cookie']*

UNEXPECTED_ALERT_OPEN = *[26, 'unexpected alert open']*

UNKNOWN_COMMAND = *[9, 'unknown command']*

UNKNOWN_ERROR = *[13, 'unknown error']*

UNKNOWN_METHOD = *['unknown method exception']*

XPATH_LOOKUP_ERROR = *[19, 'invalid selector']*

class `selenium.webdriver.remote.errorhandler.`**`ErrorHandler`**

Bases: **`object`**

Handles errors returned by the WebDriver server.

v: latest

`check_response`(*response*)

Checks that a JSON response from the WebDriver does not have an error.

Args: • response - The JSON response from the WebDriver server as a dictionary object.
Raises: If the response contains an error message.

7.24. Remote WebDriver Mobile

class `selenium.webdriver.remote.mobile.`**`Mobile`**(*driver*)

Bases: **`object`**

class **`ConnectionType`**(*mask*)

Bases: **`object`**

`__init__`(*mask*)

Initialize self. See help(type(self)) for accurate signature.

`airplane_mode`

`data`

`wifi`

`__init__`(*driver*)

Initialize self. See help(type(self)) for accurate signature.

`set_network_connection`(*network*)

Set the network connection for the remote device.

Example of setting airplane mode:

```
driver.mobile.set_network_connection(driver.mobile.AIRPLANE_MODE)
```

`AIRPLANE_MODE` = <*selenium.webdriver.remote.mobile.Mobile.ConnectionType object*>

`ALL_NETWORK` = <*selenium.webdriver.remote.mobile.Mobile.ConnectionType object*>

`DATA_NETWORK` = <*selenium.webdriver.remote.mobile.Mobile.ConnectionType object*>

`WIFI_NETWORK` = <*selenium.webdriver.remote.mobile.Mobile.ConnectionType object*>

`context`

returns the current context (Native or WebView).

`contexts`

returns a list of available contexts

`network_connection`

7.25. Remote WebDriver Remote Connection

class `selenium.webdriver.remote.remote_connection.`**`RemoteConnection`**(*remote_server_addr*, *keep_alive=False*, *resolve_ip=True*)

Bases: **`object`**

A connection with the Remote WebDriver server.

Communicates with the server using the WebDriver wire protocol:
https://github.com/SeleniumHQ/selenium/wiki/JsonWireProtocol

__init__(*remote_server_addr*, *keep_alive=False*, *resolve_ip=True*)

Initialize self. See help(type(self)) for accurate signature.

execute(*command*, *params*)

Send a command to the remote server.

Any path subtitutions required for the URL mapped to the command should be included in the command parameters.

Args:
- command - A string specifying the command to execute.
- params - A dictionary of named parameters to send with the command as its JSON payload.

classmethod **get_remote_connection_headers**(*parsed_url*, *keep_alive=False*)

Get headers for remote request.

Args:
- parsed_url - The parsed url
- keep_alive (Boolean) - Is this a keep-alive connection (default: False)

classmethod **get_timeout**()

Returns: Timeout value in seconds for all http requests made to the Remote Connection

classmethod **reset_timeout**()

Reset the http request timeout to socket._GLOBAL_DEFAULT_TIMEOUT

classmethod **set_timeout**(*timeout*)

Override the default timeout

Args:
- timeout - timeout value for http requests in seconds

7.26. Remote WebDriver Utils

selenium.webdriver.remote.utils.**dump_json**(*json_struct*)

selenium.webdriver.remote.utils.**format_json**(*json_struct*)

selenium.webdriver.remote.utils.**load_json**(*s*)

selenium.webdriver.remote.utils.**unzip_to_temp_dir**(*zip_file_name*)

Unzip zipfile to a temporary directory.

The directory of the unzipped files is returned if success, otherwise None is returned.

7.27. Internet Explorer WebDriver

class selenium.webdriver.ie.webdriver.**WebDriver**(*executable_path='IEDriverServer.exe'*, *capabilities=None*, *port=0*, *timeout=30*, *host=None*, *log_level=None*, *service_log_path=None*, *options=None*, *ie_options=None*, *desired_capabilities=None*, *log_file=None*, *keep_alive=False*)

Bases: **selenium.webdriver.remote.webdriver.WebDriver**

Controls the IEServerDriver and allows you to drive Internet Explorer

__init__(*executable_path='IEDriverServer.exe'*, *capabilities=None*, *port=0*, *timeout=30*, *host=None*, *log_level=None*, *service_log_path=None*, *options=None*, *ie_options=None*, *desired_capabilities=None*, *log_file=None*, *keep_alive=False*)

Creates a new instance of the chrome driver.

Starts the service and then creates new instance of chrome driver.

Args:
- executable_path - path to the executable. If the default is used it assumes the executable is in the $PATH
- capabilities: capabilities Dictionary object
- port - port you would like the service to run, if left as 0, a free port will be found.
- timeout - no longer used, kept for backward compatibility
- host - IP address for the service
- log_level - log level you would like the service to run.
- service_log_path - target of logging of service, may be “stdout”, “stderr” or file path.
- options - IE Options instance, providing additional IE options
- ie_options - Deprecated argument for options
- desired_capabilities - alias of capabilities; this will make the signature consistent with RemoteWebDriver.
- log_file - Deprecated argument for service_log_path
- keep_alive - Whether to configure RemoteConnection to use HTTP keep-alive.

create_options()

quit()

Quits the driver and closes every associated window.

Usage: driver.quit()

7.28. Android WebDriver

class selenium.webdriver.android.webdriver.**WebDriver**(*host='localhost', port=4444, desired_capabilities={'browserName': 'android', 'platform': 'ANDROID', 'version': ''}*)

Bases: **selenium.webdriver.remote.webdriver.WebDriver**

Simple RemoteWebDriver wrapper to start connect to Selendroid’s WebView app

For more info on getting started with Selendroid http://selendroid.io/mobileWeb.html

__init__(*host='localhost', port=4444, desired_capabilities={'browserName': 'android', 'platform': 'ANDROID', 'version': ''}*)

Creates a new instance of Selendroid using the WebView app

Args:
- host - location of where selendroid is running
- port - port that selendroid is running on
- desired_capabilities: Dictionary object with capabilities

7.29. Opera WebDriver

class selenium.webdriver.opera.webdriver.**OperaDriver**(*executable_path=None, port=0, options=None, service_args=None, desired_capabilities=None, service_log_path=None, opera_options=None, keep_alive=True*)

Bases: **selenium.webdriver.chrome.webdriver.WebDriver**

Controls the new OperaDriver and allows you to drive the Opera browser based on Chromium.

__init__(*executable_path=None, port=0, options=None, service_args=None, desired_capabilities=None, service_log_path=None, opera_options=None, keep_alive=True*)

Creates a new instance of the operadriver.

Starts the service and then creates new instance of operadriver.

Args:
- executable_path - path to the executable. If the default is used it assumes the executable is in the $PATH
- port - port you would like the service to run, if left as 0, a free port will be found.
- options: this takes an instance of OperaOptions
- service_args - List of args to pass to the driver service
- desired_capabilities: Dictionary object with non-browser specific
- service_log_path - Where to log information from the driver.
- opera_options - Deprecated argument for options capabilities only, such as “proxy” or “loggingPref”.

create_options()

class selenium.webdriver.opera.webdriver.**WebDriver**(*desired_capabilities=None*, *executable_path=None*, *port=0*, *service_log_path=None*, *service_args=None*, *options=None*)

Bases: **selenium.webdriver.opera.webdriver.OperaDriver**

class **ServiceType**

Bases: **object**

CHROMIUM = *2*

__init__(*desired_capabilities=None*, *executable_path=None*, *port=0*, *service_log_path=None*, *service_args=None*, *options=None*)

Creates a new instance of the operadriver.

Starts the service and then creates new instance of operadriver.

Args:
- executable_path - path to the executable. If the default is used it assumes the executable is in the $PATH
- port - port you would like the service to run, if left as 0, a free port will be found.
- options: this takes an instance of OperaOptions
- service_args - List of args to pass to the driver service
- desired_capabilities: Dictionary object with non-browser specific
- service_log_path - Where to log information from the driver.
- opera_options - Deprecated argument for options capabilities only, such as “proxy” or “loggingPref”.

7.30. PhantomJS WebDriver

class selenium.webdriver.phantomjs.webdriver.**WebDriver**(*executable_path='phantomjs'*, *port=0*, *desired_capabilities={'browserName': 'phantomjs', 'javascriptEnabled': True, 'platform': 'ANY', 'version': ''}*, *service_args=None*, *service_log_path=None*)

Bases: **selenium.webdriver.remote.webdriver.WebDriver**

Wrapper to communicate with PhantomJS through Ghostdriver.

You will need to follow all the directions here: https://github.com/detro/ghostdriver

__init__(*executable_path='phantomjs', port=0, desired_capabilities={'browserName': 'phantomjs', 'javascriptEnabled': True, 'platform': 'ANY', 'version': ''}, service_args=None, service_log_path=None*)

Creates a new instance of the PhantomJS / Ghostdriver.

Starts the service and then creates new instance of the driver.

Args:
- executable_path - path to the executable. If the default is used it assumes the executable is in the $PATH
- port - port you would like the service to run, if left as 0, a free port will be found.
- desired_capabilities: Dictionary object with non-browser specific capabilities only, such as "proxy" or "loggingPref".
- service_args : A List of command line arguments to pass to PhantomJS
- service_log_path: Path for phantomjs service to log to.

quit()

Closes the browser and shuts down the PhantomJS executable that is started when starting the PhantomJS

7.31. PhantomJS WebDriver Service

class selenium.webdriver.phantomjs.service.**Service**(*executable_path, port=0, service_args=None, log_path=None*)

Bases: **selenium.webdriver.common.service.Service**

Object that manages the starting and stopping of PhantomJS / Ghostdriver

__init__(*executable_path, port=0, service_args=None, log_path=None*)

Creates a new instance of the Service

Args:
- executable_path : Path to PhantomJS binary
- port : Port the service is running on
- service_args : A List of other command line options to pass to PhantomJS
- log_path: Path for PhantomJS service to log to

command_line_args()

send_remote_shutdown_command()

service_url

Gets the url of the GhostDriver Service

7.32. Safari WebDriver

class selenium.webdriver.safari.webdriver.**WebDriver**(*port=0, executable_path='/usr/bin/safaridriver', reuse_service=False, desired_capabilities={'browserName': 'safari', 'platform': 'MAC', 'version': ''}, quiet=False, keep_alive=True, service_args=None*)

Bases: **selenium.webdriver.remote.webdriver.WebDriver**

Controls the SafariDriver and allows you to drive the browser.

__init__(*port=0, executable_path='/usr/bin/safaridriver', reuse_service=False, desired_capabilities={'browserName': 'safari', 'platform': 'MAC', 'version': ''}, quiet=False, keep_alive=True, service_args=None*)

Creates a new Safari driver instance and launches or finds a running safaridriver service.

Args:
- port - The port on which the safaridriver service should listen for new connections. If zero, a free port will be found.
- executable_path - Path to a custom safaridriver executable to be used. If absent, /usr/bin/safaridriver is used.
- reuse_service - If True, do not spawn a safaridriver instance; instead, connect to an already-running service that was launched externally.
- desired_capabilities: Dictionary object with desired capabilities (Can be used to provide various Safari switches).
- quiet - If True, the driver's stdout and stderr is suppressed.
- keep_alive - Whether to configure SafariRemoteConnection to use HTTP keep-alive. Defaults to False.
- service_args : List of args to pass to the safaridriver service

debug()

get_permission(*permission*)

quit()

Closes the browser and shuts down the SafariDriver executable that is started when starting the SafariDriver

set_permission(*permission*, *value*)

7.33. Safari WebDriver Service

class selenium.webdriver.safari.service.**Service**(*executable_path*, *port=0*, *quiet=False*, *service_args=None*)

Bases: **selenium.webdriver.common.service.Service**

Object that manages the starting and stopping of the SafariDriver

__init__(*executable_path*, *port=0*, *quiet=False*, *service_args=None*)

Creates a new instance of the Service

Args:
- executable_path : Path to the SafariDriver
- port : Port the service is running on
- quiet : Suppress driver stdout and stderr
- service_args : List of args to pass to the safaridriver service

command_line_args()

service_url

Gets the url of the SafariDriver Service

7.34. Select Support

class selenium.webdriver.support.select.**Select**(*webelement*)

Bases: **object**

__init__(*webelement*)

Constructor. A check is made that the given element is, indeed, a SELECT tag. If it is not, then an UnexpectedTagNameException is thrown.

Args: • webelement - element SELECT element to wrap

Example:

```
from selenium.webdriver.support.ui import Select

Select(driver.find_element_by_tag_name("select")).select_by_index(2)
```

deselect_all()

Clear all selected entries. This is only valid when the SELECT supports multiple selections. throws NotImplementedError If the SELECT does not support multiple selections

deselect_by_index(*index*)

Deselect the option at the given index. This is done by examing the "index" attribute of an element, and not merely by counting.

Args: • index - The option at this index will be deselected
throws NoSuchElementException If there is no option with specisied index in SELECT

deselect_by_value(*value*)

Deselect all options that have a value matching the argument. That is, when given "foo" this would deselect an option like:

```
<option value="foo">Bar</option>
```

Args: • value - The value to match against
throws NoSuchElementException If there is no option with specisied value in SELECT

deselect_by_visible_text(*text*)

Deselect all options that display text matching the argument. That is, when given "Bar" this would deselect an option like:

<option value="foo">Bar</option>

Args: • text - The visible text to match against

select_by_index(*index*)

Select the option at the given index. This is done by examing the "index" attribute of an element, and not merely by counting.

Args: • index - The option at this index will be selected

throws NoSuchElementException If there is no option with specisied index in SELECT

select_by_value(*value*)

Select all options that have a value matching the argument. That is, when given "foo" this would select an option like:

<option value="foo">Bar</option>

Args: • value - The value to match against

throws NoSuchElementException If there is no option with specisied value in SELECT

select_by_visible_text(*text*)

Select all options that display text matching the argument. That is, when given "Bar" this would select an option like:

<option value="foo">Bar</option>

Args:
- text - The visible text to match against

throws NoSuchElementException If there is no option with specisied text in SELECT

all_selected_options

Returns a list of all selected options belonging to this select tag

first_selected_option

The first selected option in this select tag (or the currently selected option in a normal select)

options

Returns a list of all options belonging to this select tag

7.35. Wait Support

class selenium.webdriver.support.wait.**WebDriverWait**(*driver*, *timeout*, *poll_frequency=0.5*, *ignored_exceptions=None*)

Bases: **object**

__init__(*driver*, *timeout*, *poll_frequency=0.5*, *ignored_exceptions=None*)

Constructor, takes a WebDriver instance and timeout in seconds.

Args:
- driver - Instance of WebDriver (Ie, Firefox, Chrome or Remote)
- timeout - Number of seconds before timing out
- poll_frequency - sleep interval between calls By default, it is 0.5 second.
- ignored_exceptions - iterable structure of exception classes ignored during calls. By default, it contains NoSuchElementException only.

Example:

from selenium.webdriver.support.ui import WebDriverWait

element = WebDriverWait(driver, 10).until(lambda x: x.find_element_by_id("someId"))

is_disappeared = WebDriverWait(driver, 30, 1, (ElementNotVisibleException)).

until_not(lambda x: x.find_element_by_id("someId").is_displayed())

until(*method*, *message=''*)

Calls the method provided with the driver as an argument until the return value is not False.

until_not(*method*, *message=''*)

Calls the method provided with the driver as an argument until the return value is False.

7.36. Color Support

class selenium.webdriver.support.color.**Color**(*red*, *green*, *blue*, *alpha=1*)

Bases: **object**

Color conversion support class

Example:

```
from selenium.webdriver.support.color import Color

print(Color.from_string('#00ff33').rgba)
```

```
print(Color.from_string('rgb(1, 255, 3)').hex)
print(Color.from_string('blue').rgba)
```

__init__(*red*, *green*, *blue*, *alpha=1*)

Initialize self. See help(type(self)) for accurate signature.

static **from_string**(*str_*)

hex

rgb

rgba

7.37. Event Firing WebDriver Support

class selenium.webdriver.support.event_firing_webdriver.**EventFiringWebDriver**(*driver*, *event_listener*)

Bases: **object**

A wrapper around an arbitrary WebDriver instance which supports firing events

__init__(*driver*, *event_listener*)

Creates a new instance of the EventFiringWebDriver

Args:
- driver : A WebDriver instance
- event_listener : Instance of a class that subclasses AbstractEventListener and implements it fully or partially

Example:

```
from selenium.webdriver import Firefox
from selenium.webdriver.support.events import EventFiringWebDriver, AbstractEventLis

class MyListener(AbstractEventListener):
    def before_navigate_to(self, url, driver):
        print("Before navigate to %s" % url)
    def after_navigate_to(self, url, driver):
        print("After navigate to %s" % url)

driver = Firefox()
ef_driver = EventFiringWebDriver(driver, MyListener())
ef_driver.get("http://www.google.co.in/")
```

back()

close()

execute_async_script(*script*, **args*)

execute_script(*script*, **args*)

find_element(*by='id'*, *value=None*)

find_element_by_class_name(*name*)

find_element_by_css_selector(*css_selector*)

find_element_by_id(*id_*)

find_element_by_link_text(*link_text*)

find_element_by_name(*name*)

find_element_by_partial_link_text(*link_text*)

find_element_by_tag_name(*name*)

find_element_by_xpath(*xpath*)

find_elements(*by='id'*, *value=None*)

find_elements_by_class_name(*name*)

find_elements_by_css_selector(*css_selector*)

find_elements_by_id(*id_*)

find_elements_by_link_text(*text*)

find_elements_by_name(*name*)

find_elements_by_partial_link_text(*link_text*)

find_elements_by_tag_name(*name*)

find_elements_by_xpath(*xpath*)

forward()

get(*url*)

quit()

wrapped_driver
Returns the WebDriver instance wrapped by this EventsFiringWebDriver

class selenium.webdriver.support.event_firing_webdriver.**EventFiringWebElement**(*webelement*, *ef_driver*)
Bases: **object**

” A wrapper around WebElement instance which supports firing events

__init__(*webelement*, *ef_driver*)
Creates a new instance of the EventFiringWebElement

clear()

click()

find_element(*by='id'*, *value=None*)

find_element_by_class_name(*name*)

find_element_by_css_selector(*css_selector*)

find_element_by_id(*id_*)

find_element_by_link_text(*link_text*)

find_element_by_name(*name*)

find_element_by_partial_link_text(*link_text*)

find_element_by_tag_name(*name*)

find_element_by_xpath(*xpath*)

find_elements(*by='id'*, *value=None*)

find_elements_by_class_name(*name*)

find_elements_by_css_selector(*css_selector*)

find_elements_by_id(*id_*)

find_elements_by_link_text(*link_text*)

find_elements_by_name(*name*)

find_elements_by_partial_link_text(*link_text*)

find_elements_by_tag_name(*name*)

find_elements_by_xpath(*xpath*)

send_keys(**value*)

wrapped_element

Returns the WebElement wrapped by this EventFiringWebElement instance

7.38. Abstract Event Listener Support

class selenium.webdriver.support.abstract_event_listener.**AbstractEventListener**

Bases: **object**

Event listener must subclass and implement this fully or partially

after_change_value_of(*element*, *driver*)

after_click(*element*, *driver*)

after_close(*driver*)

after_execute_script(*script*, *driver*)

after_find(*by*, *value*, *driver*)

after_navigate_back(*driver*)

after_navigate_forward(*driver*)

after_navigate_to(*url*, *driver*)

after_quit(*driver*)

before_change_value_of(*element*, *driver*)

before_click(*element*, *driver*)

before_close(*driver*)

before_execute_script(*script*, *driver*)

before_find(*by*, *value*, *driver*)

before_navigate_back(*driver*)

before_navigate_forward(*driver*)

before_navigate_to(*url*, *driver*)

before_quit(*driver*)

on_exception(*exception*, *driver*)

7.39. Expected conditions Support

class selenium.webdriver.support.expected_conditions.**alert_is_present**

Bases: **object**

Expect an alert to be present.

__init__()

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**element_located_selection_state_to_be**(*locator*, *is_selected*)

Bases: **object**

An expectation to locate an element and check if the selection state specified is in that state. locator is a tuple of (by, path) is_selected is a boolean

__init__(*locator*, *is_selected*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**element_located_to_be_selected**(*locator*)

Bases: **object**

An expectation for the element to be located is selected. locator is a tuple of (by, path)

__init__(*locator*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**element_selection_state_to_be**(*element*, *is_selected*)

Bases: **object**

An expectation for checking if the given element is selected. element is WebElement object is_selected is a Boolean.”

__init__(*element*, *is_selected*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**element_to_be_clickable**(*locator*)

Bases: **object**

An Expectation for checking an element is visible and enabled such that you can click it.

__init__(*locator*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**element_to_be_selected**(*element*)

Bases: **object**

An expectation for checking the selection is selected. element is WebElement object

__init__(*element*)
Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**frame_to_be_available_and_switch_to_it**(*locator*)
Bases: **object**

An expectation for checking whether the given frame is available to switch to. If the frame is available it switches the given driver to the specified frame.

__init__(*locator*)
Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**invisibility_of_element**(*locator*)
Bases: **selenium.webdriver.support.expected_conditions.invisibility_of_element_located**

An Expectation for checking that an element is either invisible or not present on the DOM.

element is either a locator (text) or an WebElement

class selenium.webdriver.support.expected_conditions.**invisibility_of_element_located**(*locator*)
Bases: **object**

An Expectation for checking that an element is either invisible or not present on the DOM.

locator used to find the element

__init__(*locator*)
Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**new_window_is_opened**(*current_handles*)
Bases: **object**

An expectation that a new window will be opened and have the number of windows handles increase

__init__(*current_handles*)
Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**number_of_windows_to_be**(*num_windows*)
Bases: **object**

An expectation for the number of windows to be a certain value.

__init__(*num_windows*)
Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**presence_of_all_elements_located**(*locator*)
Bases: **object**

An expectation for checking that there is at least one element present on a web page. locator is used to find the element returns the list of WebElements once they are located

__init__(*locator*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**presence_of_element_located**(*locator*)

Bases: **object**

An expectation for checking that an element is present on the DOM of a page. This does not necessarily mean that the element is visible. locator - used to find the element returns the WebElement once it is located

__init__(*locator*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**staleness_of**(*element*)

Bases: **object**

Wait until an element is no longer attached to the DOM. element is the element to wait for. returns False if the element is still attached to the DOM, true otherwise.

__init__(*element*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**text_to_be_present_in_element**(*locator*, *text_*)

Bases: **object**

An expectation for checking if the given text is present in the specified element. locator, text

__init__(*locator*, *text_*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**text_to_be_present_in_element_value**(*locator*, *text_*)

Bases: **object**

An expectation for checking if the given text is present in the element's locator, text

__init__(*locator*, *text_*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**title_contains**(*title*)

Bases: **object**

An expectation for checking that the title contains a case-sensitive substring. title is the fragment of title expected returns True when the title matches, False otherwise

__init__(*title*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**title_is**(*title*)

Bases: **object**

An expectation for checking the title of a page. title is the expected title, which must be an exact match returns True if the title matches, false otherwise.

__init__(*title*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**url_changes**(*url*)

Bases: **object**

An expectation for checking the current url. url is the expected url, which must not be an exact match returns True if the url is different, false otherwise.

__init__(*url*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**url_contains**(*url*)

Bases: **object**

An expectation for checking that the current url contains a case-sensitive substring. url is the fragment of url expected, returns True when the url matches, False otherwise

__init__(*url*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**url_matches**(*pattern*)

Bases: **object**

An expectation for checking the current url. pattern is the expected pattern, which must be an exact match returns True if the url matches, false otherwise.

__init__(*pattern*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**url_to_be**(*url*)

Bases: **object**

An expectation for checking the current url. url is the expected url, which must be an exact match returns True if the url matches, false otherwise.

__init__(*url*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**visibility_of**(*element*)

Bases: **object**

An expectation for checking that an element, known to be present on the DOM of a page, is visible. Visibility means that the element is not only displayed but also has a height and width that is greater than 0. element is the WebElement returns the (same) WebElement once it is visible

__init__(*element*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**visibility_of_all_elements_located**(*locator*)

Bases: **object**

An expectation for checking that all elements are present on the DOM of a page and visible. Visibility means that the elements are not only displayed but also has a height and width that is greater than 0.

locator - used to find the elements returns the list of WebElements once they are located and visible

__init__(*locator*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**visibility_of_any_elements_located**(*locator*)

Bases: **object**

An expectation for checking that there is at least one element visible on a web page. locator is used to find the element returns the list of WebElements once they are located

__init__(*locator*)

Initialize self. See help(type(self)) for accurate signature.

class selenium.webdriver.support.expected_conditions.**visibility_of_element_located**(*locator*)

Bases: **object**

An expectation for checking that an element is present on the DOM of a page and visible. Visibility means that the element is not only displayed but also has a height and width that is greater than 0. locator - used to find the element returns the WebElement once it is located and visible

__init__(*locator*)

Initialize self. See help(type(self)) for accurate signature.