Helium Python API Documentation

2.0.3

BugFree Software

21/08/2018

Helium's API is contained in module `helium.api`. It is a simple Python API that makes specifying web automation cases as simple as describing them to someone looking over their shoulder at a screen.

The public functions and classes of Helium are listed below. If you wish to use Helium functions in your Python scripts you can import them from the `helium.api` module:

```
from helium.api import *
```

start_firefox(*url=None*)

Parameters **url** (*str*) – URL to open.

Starts an instance of Firefox, optionally opening the specified URL. For instance:

```
start_firefox()
start_firefox("google.com")
```

Helium does not close the browser window on shutdown of the Python interpreter. If you want to close the browser at the end of your script, use the following command:

```
kill_browser()
```

start_chrome(*url=None*, *headless=False*)

Parameters

- **url** (*str*) – URL to open.
- **headless** (*bool*) – Whether to start Chrome in headless mode.

Starts an instance of Google Chrome. You can optionally open a URL and/or start Chrome in headless mode. For instance:

```
start_chrome()
start_chrome("google.com")
start_chrome(headless=True)
start_chrome("google.com", headless=True)
```

On shutdown of the Python interpreter, Helium cleans up all resources used for controlling the browser (such as the ChromeDriver process), but does not close the browser itself. If you want to terminate the browser at the end of your script, use the following command:

```
kill_browser()
```

start_ie(*url=None*)

Parameters **url** (*str*) – URL to open.

(Windows only) Starts Internet Explorer, optionally opening the specified URL. For instance:

```
start_ie()
start_ie("google.com")
```

On shutdown of the Python interpreter, Helium cleans up all resources used for controlling the browser (such as the IEDriverServer process), but does not close the browser itself. If you want to terminate the browser at the end of your script, use the following command:

```
kill_browser()
```

go_to(*url*)

Parameters **url** (*str*) – URL to open.

Opens the specified URL in the current web browser window. For instance:

```
go_to("google.com")
```

set_driver(*driver*)
Sets the Selenium WebDriver used to execute Helium commands. See also *get_driver()* (page 2).

get_driver()
Returns the Selenium WebDriver currently used by Helium to execute all commands. Each Helium command such as `click("Login")` is translated to a sequence of Selenium commands that are issued to this driver.

write(*text*, *into=None*)

Parameters

- **text** (*one of str, unicode*) – The text to be written.
- **into** (one of str, unicode, `HTMLElement`, `selenium.webdriver.remote.webelement.WebEleme` *Alert* (page 13)) – The element to write into.

Types the given text into the active window. If parameter 'into' is given, writes the text into the text field or element identified by that parameter. Common examples of 'write' are:

```
write("Hello World!")
write("user12345", into="Username:")
write("Michael", into=Alert("Please enter your name"))
```

press(*key*)

Parameters **key** – Key or combination of keys to be pressed.

Presses the given key or key combination. To press a normal letter key such as 'a' simply call *press* for it:

```
press('a')
```

You can also simulate the pressing of upper case characters that way:

```
press('A')
```

The special keys you can press are those given by Selenium's class `selenium.webdriver.common.keys.Keys`. Helium makes all those keys available through its namespace, so you can just use them without having to refer to `selenium.webdriver.common.keys.Keys`. For instance, to press the Enter key:

```
press(ENTER)
```

To press multiple keys at the same time, concatenate them with +. For example, to press Control + a, call:

```
press(CONTROL + 'a')
```

click(*element*)

Parameters **element** (str, unicode, `HTMLElement`, `selenium.webdriver.remote.webelement.WebEle` or *Point* (page 14)) – The element or point to click.

Clicks on the given element or point. Common examples are:

```
click("Sign in")
click(Button("OK"))
click(Point(200, 300))
click(ComboBox("File type").top_left + (50, 0))
```

doubleclick(*element*)

Parameters **element** (str, unicode, `HTMLElement`, `selenium.webdriver.remote.webelement.WebEle` or *Point* (page 14)) – The element or point to click.

Performs a double-click on the given element or point. For example:

```
doubleclick("Double click here")
doubleclick(Image("Directories"))
doubleclick(Point(200, 300))
doubleclick(TextField("Username").top_left - (0, 20))
```

drag(*element*, *to*)

Parameters

- **element** (str, unicode, HTMLElement, selenium.webdriver.remote.webelement.WebElemen or *Point* (page 14)) – The element or point to drag.
- **to** (str, unicode, HTMLElement, selenium.webdriver.remote.webelement.WebElement or *Point* (page 14)) – The element or point to drag to.

Drags the given element or point to the given location. For example:

```
drag("Drag me!", to="Drop here.")
```

The dragging is performed by hovering the mouse cursor over element, pressing and holding the left mouse button, moving the mouse cursor over to, and then releasing the left mouse button again.

This function is exclusively used for dragging elements inside one web page. If you wish to drag a file from the hard disk onto the browser window (eg. to initiate a file upload), use function *drag_file()* (page 4).

find_all(*predicate*)

Lets you find all occurrences of the given GUI element predicate. For instance, the following statement returns a list of all buttons with label "Open":

```
find_all(Button("Open"))
```

Other examples are:

```
find_all(Window())
find_all(TextField("Address line 1"))
```

The function returns a list of elements of the same type as the passed-in parameter. For instance, find_all(Button(...)) yields a list whose elements are of type *Button* (page 8).

In a typical usage scenario, you want to pick out one of the occurrences returned by *find_all()* (page 3). In such cases, list.sort() can be very useful. For example, to find the leftmost "Open" button, you can write:

```
buttons = find_all(Button("Open"))
leftmost_button = sorted(buttons, key=lambda button: button.x)[0]
```

scroll_down(*num_pixels=100*)

Scrolls down the page the given number of pixels.

scroll_up(*num_pixels=100*)

Scrolls the the page up the given number of pixels.

scroll_right(*num_pixels=100*)

Scrolls the page to the right the given number of pixels.

scroll_left(*num_pixels=100*)

Scrolls the page to the left the given number of pixels.

hover(*element*)

Parameters **element** (str, unicode, HTMLElement, selenium.webdriver.remote.webelement.WebEle or *Point* (page 14)) – The element or point to hover.

Hovers the mouse cursor over the given element or point. For example:

```
hover("File size")
hover(Button("OK"))
hover(Link("Download"))
```

```
hover(Point(200, 300))
hover(ComboBox("File type").top_left + (50, 0))
```

rightclick(*element*)

Parameters **element** (str, unicode, HTMLElement, selenium.webdriver.remote.webelement.WebEle or *Point* (page 14)) – The element or point to click.

Performs a right click on the given element or point. For example:

```
rightclick("Something")
rightclick(Point(200, 300))
rightclick(Image("captcha"))
```

select(*combo_box*, *value*)

Parameters

- **combo_box** (str, unicode or *ComboBox* (page 10)) – The combo box whose value should be changed.
- **value** – The visible value of the combo box to be selected.

Selects a value from a combo box. For example:

```
select("Language", "English")
select(ComboBox("Language"), "English")
```

drag_file(*file_path*, *to*)

Simulates the dragging of a file from the computer over the browser window and dropping it over the given element. This allows, for example, to attach files to emails in Gmail:

```
click("COMPOSE")
write("example@gmail.com", into="To")
write("Email subject", into="Subject")
drag_file(r"C:\Documents\notes.txt", to="Drop files here")
```

attach_file(*file_path*, *to=None*)

Parameters

- **file_path** – The path of the file to be attached.
- **to** – The file input element to which the file should be attached.

Allows attaching a file to a file input element. For instance:

```
attach_file("c:/test.txt", to="Please select a file:")
```

The file input element is identified by its label. If you omit the to= parameter, then Helium attaches the file to the first file input element it finds on the page.

refresh()

Refreshes the current page. If an alert dialog is open, then Helium first closes it.

wait_until(*condition_fn*, *timeout_secs=10*, *interval_secs=0.5*)

Parameters

- **condition_fn** – A function taking no arguments that represents the condition to be waited for.
- **timeout_secs** – The timeout, in seconds, after which the condition is deemed to have failed.
- **interval_secs** – The interval, in seconds, at which the condition function is polled to determine whether the wait has succeeded.

Waits until the given condition function evaluates to true. This is most commonly used to wait for an element to exist:

```
wait_until(Text("Finished!").exists)
```

More elaborate conditions are also possible using Python lambda expressions. For instance, to wait until a text no longer exists:

```
wait_until(lambda: not Text("Uploading...").exists())
```

`wait_until` raises `selenium.common.exceptions.TimeoutException` if the condition is not satisfied within the given number of seconds. The parameter `interval_secs` specifies the number of seconds Helium waits between evaluating the condition function.

class `Config`

Bases: `object`

This class contains Helium's run-time configuration. To modify Helium's behaviour, simply assign to the properties of this class. For instance:

```
Config.implicit_wait_secs = 0
```

`implicit_wait_secs` = 10

`implicit_wait_secs` is Helium's analogue to Selenium's `.implicitly_wait(secs)`. Suppose you have a script that executes the following command:

```
>>> click("Download")
```

If the "Download" element is not immediately available, then Helium waits up to `implicit_wait_secs` for it to appear before raising a `LookupError`. This is useful in situations where the page takes slightly longer to load, or a GUI element only appears after a certain time.

To disable Helium's implicit waits, simply execute:

```
Config.implicit_wait_secs = 0
```

Helium's implicit waits do not affect commands *find_all()* (page 3) or `GUIElement.exists()`. Note also that setting `implicit_wait_secs` does not affect the underlying Selenium driver (see *get_driver()* (page 2)).

For the best results, it is recommended to not use Selenium's `.implicitly_wait(...)` in conjunction with Helium.

class `S` (*selector*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)

Bases: `helium.api.HTMLElement`

Parameters **`selector`** – The selector used to identify the HTML element(s).

A jQuery-style selector for identifying HTML elements by ID, name, CSS class, CSS selector or XPath. For example: Say you have an element with ID "myId" on a web page, such as `<div id="myId" .../>`. Then you can identify this element using `S` as follows:

```
S("#myId")
```

The parameter which you pass to `S(...)` is interpreted by Helium according to these rules:

- If it starts with an `@`, then it identifies elements by HTML `name`. Eg. `S("@btnName")` identifies an element with `name="btnName"`.
- If it starts with `//`, then Helium interprets it as an XPath.
- Otherwise, Helium interprets it as a CSS selector. This in particular lets you write `S("#myId")` to identify an element with `id="myId"`, or `S(".myClass")` to identify elements with `class="myClass"`.

`S` also makes it possible to read plain text data from a web page. For example, suppose you have a table of people's email addresses. Then you can read the list of email addresses as follows:

```
email_cells = find_all(S("table > tr > td", below="Email"))
emails = [cell.web_element.text for cell in email_cells]
```

Where `email` is the column header (`<th>Email</th>`). Similarly to `below` and `to_right_of`, the keyword parameters `above` and `to_left_of` can be used to search for elements above and to the left of other web elements.

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class Text(*text=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: `helium.api.HTMLElement`

Lets you identify any text or label on a web page. This is most useful for checking whether a particular text exists:

```
if Text("Do you want to proceed?").exists():
    click("Yes")
```

`Text` also makes it possible to read plain text data from a web page. For example, suppose you have a table of people's email addresses. Then you can read John's email addresses as follows:

```
Text(below="Email", to_right_of="John").value
```

Similarly to `below` and `to_right_of`, the keyword parameters `above` and `to_left_of` can be used to search for texts above and to the left of other web elements.

value
Returns the current value of this Text object.

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
        click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class Link(*text=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: `helium.api.HTMLElement`

Lets you identify a link on a web page. A typical usage of `Link` is:

```
click(Link("Sign in"))
```

You can also read a `Link`'s properties. This is most typically used to check for a link's existence before clicking on it:

```
if Link("Sign in").exists():
    click(Link("Sign in"))
```

When there are multiple occurrences of a link on a page, you can disambiguate between them using the keyword parameters `below`, `to_right_of`, `above` and `to_left_of`. For instance:

```
click(Link("Block User", to_right_of="John Doe"))
```

href
Returns the URL of the page the link goes to.

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
        click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class ListItem(*text=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: `helium.api.HTMLElement`

Lets you identify a list item (HTML `<li>` element) on a web page. This is often useful for interacting with elements of a navigation bar:

```
click(ListItem("News Feed"))
```

In other cases such as an automated test, you might want to query the properties of a `ListItem`. For example, the following line checks whether a list item with text "List item 1" exists, and raises an error if not:

```
assert ListItem("List item 1").exists()
```

When there are multiple occurrences of a list item on a page, you can disambiguate between them using the keyword parameters `below`, `to_right_of`, `above` and `to_left_of`. For instance:

```
click(ListItem("List item 1", below="My first list:"))
```

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class Button(*text=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: `helium.api.HTMLElement`

Lets you identify a button on a web page. A typical usage of `Button` is:

```
click(Button("Log In"))
```

`Button` also lets you read a button's properties. For example, the following snippet clicks button "OK" only if it exists:

```
if Button("OK").exists():
    click(Button("OK"))
```

When there are multiple occurrences of a button on a page, you can disambiguate between them using the keyword parameters `below`, `to_right_of`, `above` and `to_left_of`. For instance:

```
click(Button("Log In", below=TextField("Password")))
```

is_enabled()
Returns true if this UI element can currently be interacted with.

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class Image(*alt=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: helium.api.HTMLElement

Lets you identify an image (HTML <img> element) on a web page. Typically, this is done via the image's alt text. For instance:

```
click(Image(alt="Helium Logo"))
```

You can also query an image's properties. For example, the following snippet clicks on the image with alt text "Helium Logo" only if it exists:

```
if Image("Helium Logo").exists():
    click(Image("Helium Logo"))
```

When there are multiple occurrences of an image on a page, you can disambiguate between them using the keyword parameters below, to_right_of, above and to_left_of. For instance:

```
click(Image("Helium Logo", to_left_of=ListItem("Download")))
```

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class TextField(*label=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: helium.api.HTMLElement

Lets you identify a text field on a web page. This is most typically done to read the value of a text field. For example:

```
TextField("First name").value
```

This returns the value of the "First name" text field. If it is empty, the empty string "" is returned.

When there are multiple occurrences of a text field on a page, you can disambiguate between them using the keyword parameters `below`, `to_right_of`, `above` and `to_left_of`. For instance:

```
TextField("Address line 1", below="Billing Address:").value
```

value
Returns the current value of this text field. '' if there is no value.

is_enabled()
Returns true if this UI element can currently be interacted with.

The difference between a text field being 'enabled' and 'editable' is mostly visual: If a text field is not enabled, it is usually greyed out, whereas if it is not editable it looks normal. See also `is_editable`.

is_editable()
Returns true if the value of this UI element can be modified.

The difference between a text field being 'enabled' and 'editable' is mostly visual: If a text field is not enabled, it is usually greyed out, whereas if it is not editable it looks normal. See also `is_enabled`.

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class ComboBox(*label=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: `helium.api.HTMLElement`

Lets you identify a combo box on a web page. This can for instance be used to determine the current value of a combo box:

```
ComboBox("Language").value
```

A ComboBox may be *editable*, which means that it is possible to type in arbitrary values in addition to selecting from a predefined drop-down list of values. The property *ComboBox.is_editable()* (page 11) can be used to determine whether this is the case for a particular combo box instance.

When there are multiple occurrences of a combo box on a page, you can disambiguate between them using the keyword parameters `below`, `to_right_of`, `above` and `to_left_of`. For instance:

```
select(ComboBox(to_right_of="John Doe", below="Status"), "Active")
```

This sets the Status of John Doe to Active on the page.

is_editable()
Returns whether this combo box allows entering an arbitrary text in addition to selecting predefined values from a drop-down list.

value
Returns the currently selected combo box value.

options
Returns a list of all possible options available to choose from in the ComboBox.

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class CheckBox(*label=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: helium.api.HTMLElement

Lets you identify a check box on a web page. To tick a currently unselected check box, use:

```
click(CheckBox("I agree"))
```

CheckBox also lets you read the properties of a check box. For example, the method *CheckBox.is_checked()* (page 11) can be used to only click a check box if it isn't already checked:

```
if not CheckBox("I agree").is_checked():
    click(CheckBox("I agree"))
```

When there are multiple occurrences of a check box on a page, you can disambiguate between them using the keyword parameters below, to_right_of, above and to_left_of. For instance:

```
click(CheckBox("Stay signed in", below=Button("Sign in")))
```

is_enabled()
Returns True if this GUI element can currently be interacted with.

is_checked()
Returns True if this GUI element is checked (selected).

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class RadioButton(*label=None*, *below=None*, *to_right_of=None*, *above=None*, *to_left_of=None*)
Bases: helium.api.HTMLElement

Lets you identify a radio button on a web page. To select a currently unselected radio button, use:

```
click(RadioButton("Windows"))
```

RadioButton also lets you read the properties of a radio button. For example, the method *RadioButton.is_selected()* (page 12) can be used to only click a radio button if it isn't already selected:

```
if not RadioButton("Windows").is_selected():
    click(RadioButton("Windows"))
```

When there are multiple occurrences of a radio button on a page, you can disambiguate between them using the keyword parameters below, to_right_of, above and to_left_of. For instance:

```
click(RadioButton("I accept", below="License Agreement"))
```

is_selected()
Returns true if this radio button is selected.

exists()
Evaluates to true if this GUI element exists.

height
The height of this HTML element, in pixels.

top_left
The top left corner of this element, as a *helium.api.Point* (page 14). This point has exactly the coordinates given by this element's *.x* and *.y* properties. *top_left* is for instance useful for clicking at an offset of an element:

```
click(Button("OK").top_left + (30, 15))
```

web_element
The Selenium WebElement corresponding to this element.

width
The width of this HTML element, in pixels.

x
The x-coordinate on the page of the top-left point of this HTML element.

y
The y-coordinate on the page of the top-left point of this HTML element.

class Window(*title=None*)

Bases: helium.api.GUIElement

Lets you identify individual windows of the currently open browser session.

title

Returns the title of this Window.

handle

Returns the Selenium driver window handle assigned to this window. Note that this window handle is simply an abstract identifier and bears no relationship to the corresponding operating system handle (HWND on Windows).

exists()

Evaluates to true if this GUI element exists.

class Alert(*search_text=None*)

Bases: helium.api.GUIElement

Lets you identify and interact with JavaScript alert boxes.

text

The text displayed in the alert box.

accept()

Accepts this alert. This typically corresponds to clicking the "OK" button inside the alert. The typical way to use this method is:

```
>>> Alert().accept()
```

This accepts the currently open alert.

dismiss()

Dismisses this alert. This typically corresponds to clicking the "Cancel" or "Close" button of the alert. The typical way to use this method is:

```
>>> Alert().dismiss()
```

This dismisses the currently open alert.

exists()

Evaluates to true if this GUI element exists.

switch_to(*window*)

Parameters **window** – The title (string) of a browser window or a *Window* (page 12) object

Switches to the given browser window. For example:

```
switch_to("Google")
```

This searches for a browser window whose title contains "Google", and activates it.

If there are multiple windows with the same title, then you can use *find_all()* (page 3) to find all open windows, pick out the one you want and pass that to switch_to. For example, the following snippet switches to the first window in the list of open windows:

```
switch_to(find_all(Window())[0])
```

kill_browser()

Closes the current browser with all associated windows and potentially open dialogs. Dialogs opened as a response to the browser closing (eg. "Are you sure you want to leave this page?") are also ignored and closed.

This function is most commonly used to close the browser at the end of an automation run:

```
start_chrome()
...
# Close Chrome:
kill_browser()
```

highlight(*element*)

Parameters **element** – The element to highlight.

Highlights the given element on the webpage by drawing a red rectangle around it. This is useful for debugging purposes. For example:

```
highlight("Helium")
highlight(Button("Sign in"))
```

class Point(*x=0*, *y=0*)

A clickable point. To create a `Point` at an offset of an existing point, use + and −:

```
>>> point = Point(x=10, y=25)
>>> point + (10, 0)
Point(x=20, y=25)
>>> point - (0, 10)
Point(x=10, y=15)
```

x

The x coordinate of the point.

y

The y coordinate of the point.