Package ‘selenider’

October 1, 2023

Title  Concise, Lazy and Reliable Wrapper for ‘chromote’ and ‘RSelenium’
Version  0.1.2
Description  A user-friendly wrapper for web automation, using either ‘chromote’ or ‘RSelenium’. Provides a simple and consistent API to make web scraping and testing scripts easy to write and understand. Elements are lazy, and automatically wait for the website to be valid, resulting in reliable and reproducible code, with no visible impact on the experience of the programmer.
License  MIT + file LICENSE
Encoding  UTF-8
RoxygenNote  7.2.3
Imports  cli, coro, curl, lifecycle, prettyunits, processx, rlang (>= 1.1.0), utils, vctrs, withr
URL  https://github.com/ashbythorpe/selenider,
     https://ashbythorpe.github.io/selenider/
BugReports  https://github.com/ashbythorpe/selenider/issues
Suggests  chromote, jsonlite, knitr, purrr, rmarkdown, RSelenium, rvest, shiny, shinytest2, showimage, testthat (>= 3.0.0), wdman, xml2
Config/testthat/edition  3
Depends  R (>= 2.10)
Config/Needs/website  rmarkdown
LazyData  true
VignetteBuilder  knitr
Language  en-GB
NeedsCompilation  no
Author  Ashby Thorpe [aut, cre, cph] (<https://orcid.org/0000-0003-3106-099X>)
Maintainer  Ashby Thorpe <ashbythorpe@gmail.com>
Repository  CRAN
Date/Publication  2023-10-01 16:30:02 UTC

1
R topics documented:

- as.list.selenider_elements ........................................... 3
- back ........................................................................... 5
- close_session ................................................................. 6
- elem_ancestors ............................................................... 7
- elem_attr ..................................................................... 9
- elem_cache ................................................................... 10
- elem_click ................................................................... 12
- elem_css_property ............................................................. 13
- elem_equal ................................................................... 14
- elem_expect ................................................................. 15
- elem_expect_all .............................................................. 19
- elem_filter ................................................................. 21
- elem_flatten ................................................................ 22
- elem_hover .................................................................. 24
- elem_name .................................................................. 25
- elem_scroll_to ............................................................... 26
- elem_set_value .............................................................. 27
- elem_size .................................................................. 29
- elem_submit ................................................................. 30
- elem_text ................................................................... 31
- execute_js_fn ................................................................. 32
- find_element .................................................................. 33
- find_elements ................................................................ 35
- get_actual_element ........................................................... 37
- get_page_source ............................................................... 39
- get_session .................................................................. 40
- has_attr ..................................................................... 41
- has_css_property .............................................................. 43
- has_length .................................................................. 44
- has_name ..................................................................... 45
- has_text ..................................................................... 46
- is_enabled ................................................................... 47
- is_present ................................................................... 48
- is_visible .................................................................. 49
- keys .......................................................................... 50
- minimal_selenider_session .................................................. 51
- open_url .................................................................... 51
- read_html.selenider_session ................................................... 53
- reload ...................................................................... 54
- s ............................................................................ 55
- selenider-config ............................................................... 56
- selenider_available ............................................................ 57
- selenider_session .............................................................. 58
- take_screenshot ................................................................. 61

Index 63
as.list.selenider_elements

Iterate over an element collection

Description

[Experimental]

as.list() transforms a selenider_elements object into a list of selenider_element objects. The result can then be used in for loops and higher order functions like `lapply()`/`purrr::map()` (whereas a selenider_element object cannot). This function is stable.

`element_list()` is the underlying function called by `element_list()`.

Use `elem_flatmap()` when you want to select further sub-elements *for each* element of a collection.

`elem_flatmap()` allows you to apply a function to each element of a selenider_elements object, provided that the function returns a selenider_element/selenider_elements object itself. The result will then be flattened into a single selenider_elements object. The benefit of this over traditional iteration techniques is that the laziness of the elements will be maintained, and nothing will be fetched from the DOM. This function is experimental, and won’t work if `.f` uses `elem_flatten()` (or nested `elem_flatmap()`).

`elem_flatmap()` works by executing `.f` on a mock element, then recording the results in `x`. This means that no matter the length of `x`, `.f` is only evaluated once, and during the `elem_flatmap()` call. For this reason, `.f` should not invoke any side effects or do anything other than selecting sub-elements.

`elem_flatmap()` can essentially be viewed as a map operation (e.g. `lapply()`, `purrr::map()`) followed by a flattening operation (`elem_flatmap()`). This means that:

\[
\text{x |> elem_flatmap(.f)}
\]

is essentially equivalent to:

\[
\text{x |> as.list() |> lapply(.f) |> elem_flatten()}
\]

However, the second approach is not done lazily.

`as.list()`/`element_list()` essentially turns `x` into: \(\text{list(x[[1]], x[[2]], \ldots)\)}\) However, to do this, the length of `x` must be computed. This means that while each element inside the list is still lazy, the list itself cannot be considered lazy, since the number of elements in the DOM may change. To avoid problems, it is recommended to use an element list just after it is created, to make sure the list is an accurate representation of the DOM when it is being used.
Usage

```r
## S3 method for class 'selenider_elements'
as.list(x, timeout = NULL, ...)

element_list(x, timeout = NULL)

elem_flatmap(x, .f, ...)
```

Arguments

- **x**: A `selenider_elements` object.
- **timeout**: How long to wait for `x` to exist while computing its length.
- **...**: Passed into `.f`.
- **.f**: A function to apply to each element of `x`.

Value

`elem_flatmap()` returns a `selenider_element` object. `as.list()` returns a `selenider_element` object. `element_list()` returns a list of `selenider_element` objects.

See Also

- `elem_flatten()` to combine multiple `selenider_element/selenider_elements` objects into a single object.
- `elem_filter()` and `elem_find()` to filter element collections using a condition.

Examples

```r
html <- "
<div id='div1'>
  <p>Text 1</p>
</div>
<div id='div2'>
  <p>Text 2</p>
</div>
<div id='div3'>
  <p>Text 3</p>
</div>
<div id='div4'>
  <p>Text 4</p>
</div>
"

session <- minimal_selenider_session(html)

divs <- ss("div")

# Get the <p> tag inside each div.
```
back

```r
divs |> 
  elem_flatmap(~ x |> find_element("p"))

# Or:
p_tags <- divs |> 
  elem_flatmap(find_element, "p")

# To get the text in each tag, we can't use elem_flatmap()
for (elem in as.list(p_tags)) {
  print(elem_text(elem))
}

# Or:
lapply(as.list(p_tags), elem_text)
```

---

**Description**

`back()` navigates to the previously opened URL, or the previously opened page in your browsing history. `forward()` reverses the action of `back()`, going to the next page in your browsing history.

**Usage**

```r
back(session = NULL)
forward(session = NULL)
```

**Arguments**

- `session` A `selenider_session` object. If not specified, the global session object (the result of `get_session()`) is used.

**Value**

The session object, invisibly.

**See Also**

Other global actions: `execute_js_fn()`, `get_page_source()`, `open_url()`, `reload()`, `take_screenshot()`
Examples

```r
session <- selenider_session()
open_url("https://r-project.org")
open_url("https://www.tidyverse.org/")
back()
forward()
```

close_session

Close a session object

Description

Shut down a session object, closing the browser and stopping the server. This will be done automatically if the session is set as the local session (which happens by default).

Usage

```r
close_session(x = NULL)
```

Arguments

- `x` A `selenider_session` object. If omitted, the local session object will be closed.

Value

Nothing.

See Also

- `selenider_session()`

Examples

```r
session <- selenider_session(local = FALSE)
close_session(session)
```
**elem_ancestors**

*Get the DOM family of an element*

---

**Description**

Find all elements with a certain relative position to an HTML element.

- `elem_ancestors()` selects every element which contains the current element (children, grandchildren, etc.).
- `elem_parent()` selects the element that contains the current element.
- `elem_siblings()` selects every element which has the same parent as the current element.
- `elem_children()` selects every element which is connected to and directly below the current element.
- `elem_descendants()` selects every element that is contained by the current element. The current element does not have to be a direct parent, but must be some type of ancestor.

**Usage**

- `elem_ancestors(x)`
- `elem_parent(x)`
- `elem_siblings(x)`
- `elem_children(x)`
- `elem_descendants(x)`

**Arguments**

- `x` A `selenider_element` object.

**Details**

All functions except `elem_children()` and `elem_descendants()` use XPath selectors, so may be slow, especially when using chromote as a backend.

**Value**

All functions return a `selenider_elements` object, except `elem_parent()`, which returns a `selenider_element` object (since an element can only have one parent).
See Also

- `find_element()` and `find_elements()` for other ways of selecting elements. These functions allow you to select ancestors using one or more conditions (e.g. CSS selectors).
- `elem_filter()` and `elem_find()` for filtering element collections.

Examples

```r
html <- "
<html>
<body>
<div>
 <div id='current'>
  <p></p>
  <div>
   <p></p>
   <br>
  </div>
 </div>
 <div></div>
<p></p>
</div>
</body>
</html>
"

session <- minimal_selenider_session(html)

current <- s("#current")

# Get all the names of an element collection
elem_names <- function(x) {
  x |> as.list() |> vapply(elem_name, FUN.VALUE = character(1))
}

current |> elem_ancestors() |> elem_expect(has_length(3)) |> elem_names() # html, div, body

current |> elem_parent() |> elem_name() # div

current |> elem_siblings() |> elem_expect(has_length(2)) |
```
elem_attr

Get attributes of an element

Description

Get an attribute of a selenider_element object.

elem_attr() returns a single attribute value as a string.
elem_attrs() returns a named list containing every attribute.
elem_value() returns the 'value' attribute.

Usage

elem_attr(x, name, default = NA_character_, timeout = NULL)

elem_attrs(x, timeout = NULL)

elem_value(x, ptype = character(), timeout = NULL)

Arguments

x A selenider_element object.
name The name of the attribute to get; a string.
default The default value to use if the attribute does not exist in the element.
timeout The time to wait for x to exist.
ptype The type to cast the value to. Useful when the value is an integer or decimal number. By default, the value is returned as a string.

Value

elem_attr() returns a character vector of length 1. elem_attrs() returns a named list of strings. The return value of elem_value() has the same type as ptype and length 1.
See Also

Other properties: elem_css_property(), elem_name(), elem_size(), elem_text()

Examples

```r
html <- "
<a class='link' href='https://r-project.org'>R</a>
<input type='number' value='0'>
"

session <- minimal_selenider_session(html)

s("a") |>
  elem_attr("href")

s("a") |>
  elem_attrs()

s("input[type='number']") |>
  elem_value(ptype = integer())
```

---

**elem_cache**

*Force an element to be collected and stored*

Description

selenider_element/selenider_elements objects are generally lazy, meaning they only collect the actual element in the DOM when absolutely necessary, and forget it immediately after. This is to avoid situations where the DOM changes after an element has been collected, resulting in errors and unreliable behaviour.

elem_cache() forces an element or collection of elements to be collected and stored, making it eager rather than lazy. This is useful when you are operating on the same element multiple times, since only collecting the element once will improve performance. However, you must be sure that the element will not change on the page while you are using it.

Usage

```r
elem_cache(x, timeout = NULL)
```

Arguments

- `x` A selenider_element/selenider_elements object.
- `timeout` How long to wait for the element(s) to exist while collecting them.
Details
These functions do not make selenider elements *permanently* eager. Further sub-elements will not be cached unless specified.
For example, consider the following code:

```r
s(".class1") |>  
  elem_parent() |>  
  elem_cache() |>  
  find_element(".class2")
```

In this example, the parent of the element with class ".class1" will be cached, but the child element with class ".class2" will not.

Value
A modified version of `x`. The result of `elem_cache()` can be used as a normal `selenider_element/selenider_elements` object.

See Also
- `find_element()` and `find_elements()` to select elements.
- `element_list()` and `elem_flatmap()` if you want to iterate over an element collection.

Examples

```r
html <- "
<div>
  <p id='specifictext'></p>
  <button></button>
</div>
"

session <- minimal_selenider_session(html)

# Selecting this button may be slow, since we are using relative XPath selectors.
button <- s("#specifictext") |>  
  elem_siblings() |>  
  elem_find(has_name("button"))

# But we need to click the button 10 times!
# Normally, this would involve fetching the button from the DOM 10 times
click_button_10_times <- function(x) {
  lapply(1:10, \(unnused) elem_click(x))
  invisible(NULL)
}

# But with elem_cache(), the button will only be fetched once
cached_button <- elem_cache(button)
```
elem_click

# But the cached button is less reliable if the DOM is changing
execute_js_fn("x => { x.outerHTML = '<button></button>'; }", button)

try(elem_click(cached_button, timeout = 0.1))

# But the non-cached version works
elem_click(button)

elem_click

Click an element

Description

Clicks on an HTML element, either by simulating a mouse click or by triggering the element’s "click" event.

elem_click() left clicks on the element, elem_double_click() left clicks on the element two times in a short period of time, while elem_right_click() right clicks on an element, opening its context menu.

Usage

elem_click(x, js = FALSE, timeout = NULL)

elem_double_click(x, js = FALSE, timeout = NULL)

elem_right_click(x, js = FALSE, timeout = NULL)

Arguments

x A selenider_element object.

js Whether to click the element using JavaScript. For elem_right_click(), this is ignored if Selenium is being used, since right clicking using RSelenium does not seem to work (so JavaScript is used instead).

timeout How long to wait for the element to exist.

Value

x, invisibly.

See Also

Other actions: elem_hover(), elem_scroll_to(), elem_set_value(), elem_submit()
**elem_css_property**

Get a CSS property of an element

---

**Usage**

```r
elem_css_property(x, name, timeout = NULL)
```

---

**Description**

Get a CSS property of an element (e.g. "background-color"). Specifically, the computed style is returned, meaning that, for example, widths and heights will be returned in pixels, and colours will be returned as an RGB value.

---

**Examples**

```r
html <- "
<button onclick = hidetext() oncontextmenu = showtext()></button>
<p id = 'texttohide'>Hello!</p>
"

js <- "
function hidetext() {
    document.getElementById('texttohide').style.display = 'none'
}

function showtext() {
    document.getElementById('texttohide').style.display = 'block'
}
"

session <- minimal_selenider_session(html, js = js)
elem_expect(s("p"), is_visible)
s("button") |> 
elem_click()
elem_expect(s("p"), is_invisible)
s("button") |> 
elem_right_click()
elem_expect(s("p"), is_visible)
```
**elem_equal**

Are two elements equivalent?

**Description**

Checks if two selenider_element objects point to the same element on the page. elem_equal() is equivalent to using ==, but allows you to specify a timeout value if needed.

**Usage**

```
elem_equal(x, y, timeout = NULL)
```

## S3 method for class 'selenider_element'
el1 == e2

---

### Arguments

- **x**
  - A selenider_element object.
- **name**
  - The name of the CSS property to get.
- **timeout**
  - The time to wait for x to exist.

### Value

A string, or NA if the property does not exist.

### See Also

Other properties: elem_attr(), elem_name(), elem_size(), elem_text()

### Examples

```
html <- "<p style='visibility:hidden; color:red;'>Text</p>"

session <- minimal_selenider_session(html)

s("p") |> elem_css_property("visibility")

s("p") |> elem_css_property("color")
```
elem_expect

Arguments

- **x, y, e1, e2**: `selenider_element` objects to compare.
- **timeout**: How long to wait for the elements to be present.

Value

TRUE or FALSE.

See Also

- `elem_filter()` and `elem_find()` for filtering collection of elements.

Examples

```r
html <- "
<div></div>
<div class='second'>
<p></p>
</div>
"

session <- minimal_selenider_session(html)

s("div") == ss("div")[[1]]

has_p_child <- function(x) {
  x |> # Direct children
  elem_children() |> # Direct children
  elem_filter(has_name("p")) |> # Direct children
  has_at_least(1)
}

ss("div") |> # elem_find has_p_child
  elem_find(has_p_child) |> # TRUE
  elem_equal(s(".second"))
```

elem_expect

Test one or more conditions on HTML elements

Description

`elem_expect()` waits for a set of conditions to return TRUE. If, after a certain period of time (by default 4 seconds), this does not happen, an informative error is thrown. Otherwise, the original element is returned.

`elem_wait_until()` does the same, but returns a logical value (whether or not the test passed), allowing you to handle the failure case explicitly.
Usage

elem_expect(x, ..., testthat = NULL, timeout = NULL)

elem_wait_until(x, ..., timeout = NULL)

Arguments

x  A selenider_element/selenider_elements object, or a condition.

... <dynamic-dots> Function calls or functions that must return a logical value. If multiple conditions are given, they must all be TRUE for the test to pass.

testthat Whether to treat the expectation as a testthat test. You do not need to explicitly provide this most of the time, since by default, we can use testthat::is_testing() to figure out whether elem_expect() is being called from within a testthat test.

timeout The number of seconds to wait for a condition to pass. If not specified, the timeout used for x will be used, or the timeout of the local session if an element is not given.

Value

elem_expect() invisibly returns the element(s) x, or NULL if an element or collection of elements was not given in x.

elem_wait_for() returns a boolean flag: TRUE if the test passes, FALSE otherwise.

Conditions

Conditions can be supplied as functions or calls.

Functions allow you to use unary conditions without formatting them as a call (e.g. is_present rather than is_present()). It also allows you to make use of R’s anonymous function syntax to quickly create custom conditions. x will be supplied as the first argument of this function.

Function calls allow you to use conditions that take multiple arguments (e.g. has_text()) without the use of an intermediate function. The call will be modified so that x is the first argument to the function call. For example, has_text("a") will be modified to become: has_text(x, "a").

The and (&&), or (||) and not (!) functions can be used on both types of conditions. If more than one condition are given in . . . , they are combined using &&.

Custom conditions

Any function which takes a selenider element or element collection as its first argument, and returns a logical value, can be used as a condition.

Additionally, these functions provide a few features that make creating custom conditions easy:

• Errors with class expect_error_continue are handled, and the function is prevented from terminating early. This means that if an element is not found, the function will retry instead of immediately throwing an error.
• selenider functions used inside conditions have their timeout, by default, set to 0, ignoring the local timeout. This is important, since `elem_expect()` and `elem_wait_until()` implement a retry mechanic manually. To override this default, manually specify a timeout.

These two features allow you to use functions like `elem_text()` to access properties of an element, without needing to worry about the errors that they throw or the timeouts that they use. See Examples for a few examples of a custom condition.

These custom conditions can also be used with `elem_filter()` and `elem_find()`.

See Also

• `is_present()` and other conditions for predicates for HTML elements. (If you scroll down to the See also section, you will find the rest).

• `elem_expect_all()` and `elem_wait_until_all()` for an easy way to test a single condition on multiple elements.

• `elem_filter()` and `elem_find()` to use conditions to filter elements.

Examples

```r
elem <- "
<div class='class1'>
  <button id='disabled-button' disabled>Disabled</button>
  <p style='visibility:hidden;'>Example text</p>
  <button id='enabled-button'>Enabled</button>
</div>
<div class='class3'>
</div>
"

session <- minimal_selenider_session(elem)

s(".class1") |> elem_expect(is_present)

s("#enabled-button") |> elem_expect(is_visible, is_enabled)

s("#disabled-button") |> elem_expect(is_disabled)

# Error: element is visible but not enabled
s("#disabled-button") |> elem_expect(is_visible, is_enabled, timeout = 0.5) |> try() # Since this condition will fail

s(".class2") |> elem_expect(!is_present, !is_in_dom, is_absent) # All 3 are equivalent

# All other conditions will error if the element does not exist
s(".class2") >
```
elem_expect(is_invisible, timeout = 0.1) |> try()

# elem_expect() returns the element, so can be used in chains
s("#enabled-button") |> elem_expect(is_visible && is_enabled) |> elem_click()
# Note that elem_click() will do this automatically
s("p") |> elem_expect(is_hidden, has_exact_text("Example text"))

# Or use an anonymous function
s("p") |> elem_expect(function(elem) identical(elem_text(elem), "Example text"))

# If your conditions are not specific to an element, you can omit the 'x' argument
elem_1 <- s(".class1")
elem_2 <- s(".class2")
elem_expect(is_present(elem_1) || is_present(elem_2))

# We can now use the conditions on their own to figure out which element exists
if (is_present(elem_1)) {
  print("Element 1 is visible")
} else {
  print("Element 2 is visible")
}

# Use elem_wait_until() to handle failures manually
elem <- s(".class2")
if (elem_wait_until(elem, is_present)) {
  elem_click(elem)
} else {
  reload()
}

# Creating a custom condition is easiest with an anonymous function
s("p") |> elem_expect(
  \(x\) if (elem) elem |
    elem_text() |
      grepl(pattern = "Example .*")
)

# Or create a function, to reuse the condition multiple times
text_contains <- function(x, pattern) {
  text <- elem_text(x)
  grepl(pattern, text)
}

s("p") |>
elem_expect_all

```r
elem_expect(text_contains("Example *
"))

# If we want to continue on error, we need to use the "expect_error_continue" class
# This involves making a custom error object
error_condition <- function()
  
  my_condition <- list(message = "Custom error!")
  class(my_condition) <- c("expect_error_continue", "error", "condition")
  stop(my_condition)

# This is much easier with rlang::abort() / cli::cli_abort():
error_condition_2 <- function()
  
  rlang::abort("Custom error!", class = "expect_error_continue")

# This error will not be caught
try(elem_expect(stop("Uncaught error!")))

# These will eventually throw an error, but will wait 0.5 seconds to do so.
try(elem_expect(error_condition(), timeout = 0.5))
try(elem_expect(error_condition_2(), timeout = 0.5))
```

description

**elem_expect_all** and **elem_wait_until_all** are complements to **elem_expect()** and **elem_wait_until()** that test conditions on multiple elements in an element collection.

**Usage**

```r
elem_expect_all(x, ..., testthat = NULL, timeout = NULL)
elem_wait_until_all(x, ..., timeout = NULL)
```

**Arguments**

- `x` A selenider_elements() object.
- `...` <dynamic-dots> Function calls or functions that must return a logical value. If multiple conditions are given, they must all be TRUE for the test to pass. See **elem_expect()** for more details.
- `testthat` Whether to treat the expectation as a testthat test. You do not need to explicitly provide this most of the time, since by default, we can use **testthat::is_testing()** to figure out whether **elem_expect()** is being called from within a testthat test.
timeout The number of seconds to wait for a condition to pass. If not specified, the timeout used for \texttt{x} will be used, or the timeout of the local session if an element is not given.

Details

If \texttt{x} does not contain any elements, \texttt{elem_expect_all()} and \texttt{elem_wait_until_all()} will succeed. You may want to first verify that at least one element exists with \texttt{has_at_least()}.

\texttt{elem_expect_all()} and \texttt{elem_wait_until_all()} can be thought of as alternatives to the use of \texttt{all(vapply(FUN.VALUE = logical(1)))(or purrr::every())} within \texttt{elem_expect()} and \texttt{elem_wait_until()}.

For example, the following two expressions are equivalent (where \texttt{x} is an element collection).

\begin{verbatim}
  elem_expect_all(x, \(\text{all(vapply(as.list(element), is_present, logical(1))})\))
  elem_expect_all(x, is_present)
\end{verbatim}

However, the second example will give a more detailed error message on failure.

Value

\texttt{elem_expect_all()} returns \texttt{x}, invisibly.

\texttt{elem_wait_until_all()} returns a boolean flag: \texttt{TRUE} if the test passes, \texttt{FALSE} otherwise.

See Also

- \texttt{elem_expect()} and \texttt{elem_wait_until()}.
- \texttt{is_present()} and other conditions for predicates for HTML elements. (If you scroll down to the \textit{See also} section, you will find the rest).

Examples

\begin{verbatim}
html <- "
  <div id='div1'>Content 1</div>
  <div id='div2'>Content 2</div>
  <div id='div3' style='display:none;'>Content 3</div>
  <div id='div4'>Content 4</div>
"

session <- minimal_selenider_session(html)

ss("div") |> elem_expect_all(is_visible, timeout = 0.1) |> try()

ss("div")[-3] |> elem_expect_all(is_visible)
\end{verbatim}
**elem_filter**

*Extract a subset of HTML elements*

---

**Description**

Operators to extract a subset of elements, or a single element, from a selenider element collection. `elem_filter()` and `elem_find()` allow you to use conditions to filter HTML elements (see `is_present()` and other conditions). `elem_find()` returns the first element that satisfies one or more conditions, while `elem_filter()` returns every element that satisfies these conditions. 

`[` and `[[` with a numeric subscript can be used on an element collection to filter the elements by position. `[` returns a single element at a specified location, while `[[` returns a collection of the elements at more than one position.

**Usage**

```r
elem_filter(x, ...)

elem_find(x, ...)

## S3 method for class 'selenider_elements'
x[i]

## S3 method for class 'selenider_elements'
x[[i]]
```

**Arguments**

- **x**  
  A `selenider_elements` object.

- **...**  
  `<dynamic-dots>` Conditions (functions or function calls) that are used to filter the elements of `x`.

- **i**  
  A number (or for `[`, a vector of one or more numbers) used to select elements by position.

**Details**

As with the `find_element()` and `find_elements()` functions, these functions are lazy, meaning that the elements are not fetched and filtered until they are needed.

Conditions can be functions or function calls (see `elem_expect()` for more details).

**Value**

`elem_filter()` and `[` return a `selenider_elements` object, since they can result in multiple elements. `elem_find()` and `[[` return a single `selenider_element` object.
See Also

- `find_elements()` and `ss()` to get elements to filter.
- `is_present()` and other conditions for predicates for HTML elements. (If you scroll down to the See also section, you will find the rest).

Examples

```r
html <- "
<button disabled>Button 1</button>
<button>Button 2</button>
<p>Text</p>
<div style='display:none;'></div>
"

session <- minimal_selenider_session(html)

elements <- ss("./*")

# Gives the same result as s()
elements[[1]]

elements[1:3]

elements[-2]

elements |> 
  elem_filter(is_visible)

elements |> 
  elem_find(is_visible)

# The above is equivalent to:
visible_elems <- elements |> 
  elem_filter(is_visible)
visible_elems[[1]]

# In R >= 4.3.0, we can instead do:
# ss(".class1") |> 
#   elem_filter(is_visible) |> 
#   _[[1]]

ss("button") |> 
  elem_filter(is_enabled)
```

---

elem_flatten

**Combine multiple HTML elements**
Description
Combine a set of selenider_element/selenider_elements objects into a single selenider_elements object, allowing you to perform actions on them at once. c() and elem_flatten() do the same thing, but elem_flatten() works when given a list of selenider_element/selenider_elements objects.

Usage
elem_flatten(...)

## S3 method for class 'selenider_element'
c(...)

## S3 method for class 'selenider_elements'
c(...)

Arguments

...<dynamic-dots> selenider_element or selenider_elements objects to be combined, or lists of such objects.

Value
A selenider_elements object.

See Also
• as.list.selenider_elements() to iterate over element collections.

Examples

html <- "
<div id='id1'></div>
<div class='.class2'></div>
<button id='button1'>Click me!</button>
<div class='button-container'>
  <button id='button2'>No, click me!</button>
</div>
"

session <- minimal_selenider_session(html)

button_1 <- s("#button1")
button_2 <- s("#button2")

buttons <- elem_flatten(button_1, button_2)

buttons |> elem_expect_all(is_enabled)
buttons |> 
  as.list() |> 
  lapply(elem_click)

# Doesn't just have to be single elements 
first_2_divs <- ss("div")[1:2] 

elem_flatten(first_2_divs, button_2) |> 
  length()

# We would like to use multiple css selectors and combine the results 
selectors <- c("id1", "#id1", # Will select 1 element 
  "button", # Will select 2 elements  
  "p" # Will select 0 elements
)

lapply(selectors, ss) |> 
  elem_flatten() |> 
  length() # 3

---

**elem_hover**

*Hover over an element*

**Description**

Move the mouse over to an HTML element and hover over it, without actually clicking or interacting with it.

**Usage**

elem_hover(x, js = FALSE, timeout = NULL)

**Arguments**

- **x**: A selenider_element object.
- **js**: Whether to hover the element using JavaScript.
- **timeout**: How long to wait for the element to exist.

**Value**

x, invisibly.

**See Also**

Other actions: elem_click(), elem_scroll_to(), elem_set_value(), elem_submit()
### elem_name

**Get the tag name of an element**

Get the tag name (e.g. "p" for a `<p>` tag) of a `selenider_element` object.

**Usage**

```r
elem_name(x, timeout = NULL)
```

**Arguments**

- `x` A `selenider_element` object.
- `timeout` The time to wait for `x` to exist.

**Value**

A string.

**See Also**

Other properties: `elem_attr()`, `elem_css_property()`, `elem_size()`, `elem_text()`
elem_scroll_to

**Scroll to an element**

### Description

Scrolls to an HTML element.

### Usage

```r
elem_scroll_to(x, js = FALSE, timeout = NULL)
```

### Arguments

- `x`: A `selenider_element` object.
- `js`: Whether to scroll to the element using JavaScript.
- `timeout`: How long to wait for the element to exist.

### Value

`x`, invisibly.

### See Also

Other actions: `elem_click()`, `elem_hover()`, `elem_set_value()`, `elem_submit()`

### Examples

```
html <- "
\<div class='mydiv'></div>"
session <- minimal_selenider_session(html)

s(".mydiv") |> elem_name()
```

```
js <- "
\<button onclick='checkScrolled()'></button>\n\<p>Scroll down to find me!</p>"
```

```
html <- "
\<div style='height:100%; min-height:100vh'></div>"
```

```
js <- "
```
function checkScrolled() {
    let element = document.getElementsByTagName('p').item(0);
    let rect = element.getBoundingClientRect();
    // If paragraph is in view
    if (rect.bottom <= (window.innerHeight || document.documentElement.clientHeight)) {
        element.innerText = 'You found me!';
    }
}

session <- minimal_selenider_session(html, js = js)

s("p") |> elem_scroll_to()

s("button") |> elem_click()

elem_expect(s("p"), has_text("You found me!"))

---

**elem_set_value**  
*Set the value of an input*

**Description**

`elem_set_value()` sets the value of an HTML input element to a string.

**Usage**

```r
elem_set_value(x, text, timeout = NULL)
```

```r
elem_send_keys(x, ..., modifiers = NULL, timeout = NULL)
```

```r
elem_clear_value(x, timeout = NULL)
```

**Arguments**

- **x**  
  A selenider_element object. For `elem_send_keys()`, this can be NULL, meaning that the keys will be sent to the current page (or the currently focused element) instead of a specific element.

- **text**  
  A string to set the value of the input element to.

- **timeout**  
  How long to wait for the element to exist.

- **...**  
  A set of inputs to send to `x`.

- **modifiers**  
  A character vector; one or more of "shift", "ctrl"/"control", "alt", and "command"/"meta". Note that when using chromote as a backend, these do not work on Mac OS.
Details

elem_send_keys() sends a set of inputs to an element.

elem_clear_value() sets the value of an HTML element to "", removing any existing content.

Value

x, invisibly.

See Also

Other actions: elem_click(), elem_hover(), elem_scroll_to(), elem_submit()

Examples

html <- "
<input type='text' oninput='recordChange(event)' onkeypress='return checkEnter(event);'>
<p>
"

js <- "
function recordChange(e) {
    document.getElementsByTagName('p').item(0).innerText = e.target.value;
}

function checkEnter(e) {
    // If the key pressed was Enter
    if (e.keyCode == 13) {
        document.getElementsByTagName('p').item(0).innerText = 'Enter pressed!';
        return false;
    }
    return true;
}
"

session <- minimal_selenider_session(html, js = js)

elem_expect(s("p"), has_exact_text(''))

input <- s("input")

elem_set_value(input, "my text")

elem_expect(s("p"), has_text("my text"))

elem_clear_value(input)

elem_expect(s("p"), has_exact_text(''))

elem_send_keys(input, keys$enter)
elem_size

elem_expect(s("p"), has_text("Enter pressed!"))

---

elem_size  Get the number of elements in a collection

Description
Get the number of elements in a HTML element collection, waiting for the parent elements (if any) to exist before returning a value.

length() and elem_size() can be used interchangeably, the only difference being that elem_size() allows you to specify a timeout.

Usage

elem_size(x, timeout = NULL)

## S3 method for class 'selenider_elements'
length(x)

Arguments

x  A selenider_elements object.
timeout  The time to wait for the parent of x (if any) to exist.

Value
An integer representing the number of elements in the collection.

See Also
Other properties: elem_attr(), elem_css_property(), elem_name(), elem_text()

Examples

html <- "
<div></div>
<div></div>
<div></div>
<div></div>
"

session <- minimal_selenider_session(html)

ss("div") |> length()
elem_submit

Submit an element

Description
If an element is an ancestor of a form, submits the form. Works by walking up the DOM, checking each ancestor element until the element is a `<form>` element, which it then submits. If such an element does not exist, an error is thrown.

Usage
elem_submit(x, js = FALSE, timeout = NULL)

Arguments
- `x` A selenider_element object.
- `js` Whether to submit the form using JavaScript.
- `timeout` How long to wait for the element to exist.

Value
`x`, invisibly.

See Also
Other actions: `elem_click()`, `elem_hover()`, `elem_scroll_to()`, `elem_set_value()`

Examples
```r
html <- "
<form>
<input type='submit'>
<p>Random text</p>
</form>
<a>Random link</a>
"

doc <- minimal_selenider_session(html)
elem_submit(s("input"))
elem_submit(s("p"))

# Won't work since the element doesn't have a form ancestor
try(elem_submit(s("a"), timeout = 0.5))
```
**Description**

Get the inner text of a `selenium_element` object.

**Usage**

`elem_text(x, timeout = NULL)`

**Arguments**

- `x` A `selenium_element` object.
- `timeout` The time to wait for `x` to exist.

**Value**

A string.

**See Also**

Other properties: `elem_attr()`, `elem_css_property()`, `elem_name()`, `elem_size()`

**Examples**

```r
html <- "<p>Example text</p>
"

session <- minimal_selenium_session(html)

s("p") |>  
  elem_text()
```
execute_js_fn

Execute a JavaScript function

Description

[Experimental]

Execute a JavaScript function on zero or more arguments.

execute_js_expr() is a simpler version of execute_js_fn() that can evaluate simple expressions (e.g. "alert()"). To return a value, you must do so explicitly using "return".

These functions are experimental because their names and parameters are liable to change. Additionally, their behaviour can be inconsistent between different session types (chromote and selenium) and different browsers.

Usage

execute_js_fn(fn, ..., .timeout = NULL, .session = NULL, .debug = FALSE)
execute_js_expr(expr, ..., .timeout = NULL, .session = NULL, .debug = FALSE)

Arguments

fn A string defining the function.
...
Arguments to the function/expression. These must be unnamed, since JavaScript does not support named arguments.
.timeout How long to wait for any elements to exist in the DOM.
.session The session to use, if ... does not contain any selenider elements.
.debug Whether to print the final expression that is executed. Mostly used for debugging the functions themselves, but can also be used to identify problems in your own JavaScript code.

expr An expression to execute.

Details

... can contain selenider_element/selenider_elements objects, which will be collected and then passed into the function. However, more complex objects (e.g. lists of selenider elements) will not be moved into the JavaScript world correctly.

Similarly, nodes and lists of nodes returned from a JavaScript function will be converted into their corresponding selenider_element/selenider_elements objects, while more complex objects will not. These elements are not lazy (see elem_cache()), so make sure you only use them while you are sure they are still on the page.

Value

The return value of the JavaScript function, turned back into an R object.
See Also

Other global actions: `back()`, `get_page_source()`, `open_url()`, `reload()`, `takeScreenshot()`

Examples

```r
html <- "
<button class='mybutton'>Click me</button>
"  
session <- minimal_selenider_session(html)
execute_js_fn("(x, y) => x + y", 1, 1)
execute_js_expr("arguments[0] + arguments[1]", 1, 1)
execute_js_fn("x => x.click()",{.mybutton})
execute_js_expr("arguments[0].click()", {.mybutton})
```

---

**find_element**  
*Find a single HTML child element*

Description

Find the first HTML element using a CSS selector, an XPath, or a variety of other methods.

Usage

```r
find_element(x, ...)
```

## S3 method for class 'selenider_session'

```r
find_element(  
x,  
css = NULL,  
xpath = NULL,  
id = NULL,  
class_name = NULL,  
name = NULL,  
link_text = NULL,  
...  
)
```

## S3 method for class 'selenider_element'

```r
find_element(  
x,  
```
find_element

css = NULL,
xpath = NULL,
id = NULL,
class_name = NULL,
name = NULL,
link_text = NULL,
...
)

Arguments

x A selenider session or element.
... Arguments passed to methods.
css A css selector.
xpath An XPath.
id The id of the element you want to select.
class_name The class name of the element you want to select.
name The name attribute of the element you want to select.
link_text The link text of the link element that you would like to select.

Details

If more than one method is used to select an element (e.g. css and xpath), the first element which satisfies all conditions will be found.

CSS selectors are generally recommended over other options, since they are usually the easiest to read. Use "tag_name" to select by tag name, ".class" to select by class, and "#id" to select by id.

Value

A selenider_element object.

See Also

- `s()` to quickly select an element without specifying the session.
- `find_elements()` to select multiple elements.
- `selenider_session()` to begin a session.

Examples

html <- "
<div class='class1'>
  <div id='id1'>
    <a href='https://r-project.org'>Click me!</a>
  </div>
</div>
<p>Example text</p>"
find_elements

Find multiple HTML child elements

Description

Find every available HTML element using a CSS selector, an XPath, or a variety of other methods.

Usage

find_elements(x, ...)

## S3 method for class 'selenider_session'
find_elements(
  x,
  css = NULL,
  xpath = NULL,
  id = NULL,
  class_name = NULL,
  name = NULL,
  link_text = NULL,

```r
<\/div>
"

session <- minimal_selenider_session(html)

session |>
  find_element("div")

session |>
  find_element(name = "div") |>
  find_element(xpath = ".\./p")

s("div") |>
  find_element("#id1")

s("div") |>
  find_element(id = "id1") |>
  find_element(link_text = "Click me!")

# Complex Xpath expressions are easier to read as chained CSS selectors.
s("//div[contains(@class, 'class1')]/div/a")

s("div.class1") |>
  find_element("div") |>
  find_element("a")
```
find_elements


## S3 method for class 'selenider_element'
find_elements(
  x,
  css = NULL,
  xpath = NULL,
  id = NULL,
  class_name = NULL,
  name = NULL,
  link_text = NULL,
  ...
)

Arguments

  x          A selenider session or element.
  ...        Arguments passed to methods.
  css        A css selector.
  xpath      An XPath.
  id         The id of the element you want to select.
  class_name The class name of the element you want to select.
  name       The name attribute of the element you want to select.
  link_text  The link text of the link element that you would like to select.

Details

If more than one method is used to select an element (e.g. css and xpath), the first element which satisfies every condition will be found.

Value

A selenider_elements object.

See Also

- ss() to quickly select multiple elements without specifying the session.
- find_element() to select multiple elements.
- selenider_session() to begin a session.
- elem_children() and family to select elements using their relative position in the DOM.
- elem_filter() and elem_find() for filtering element collections.
Examples

```r
html <- "
<div id='outer-div'>
    <div>
        <p>Text 1</p>
        <p>Text 2</p>
        <p>Text 3</p>
    </div>
</div>

session <- minimal_selenider_session(html)

session |> find_elements("div")

# Or:
ss("div")

session |> find_element("#outer-div") |> find_elements("p")

# The above can be shortened to:
s("#outer-div") |
    find_elements("p")
```

---

**get_actual_element**

*Get the element associated with a selenider element*

**Description**

Turn a lazy selenium element or element collection into a backendNodeId (chromote) or an *RSele-nium::webElement*. Use this to perform certain actions on the element that are not implemented in selenider.

*get_actual_element()* turns a selenider_element object into a single backendNodeId or *RSe-le-nium::webElement* object. The function will wait for the object to exist in the DOM.

*get_actual_elements()* turns a selenider_elements object into a list of *RSe-le-nium::webElement* objects, waiting for any parent objects to exist in the DOM.
get_actual_element

Usage

get_actual_element(x, timeout = NULL)

get_actual_elements(x, timeout = NULL)

Arguments

x A selenider_element or selenider_elements object, produced by find_element() / find_elements().

timeout The timeout to use while asserting that the item exists. If NULL, the timeout of the selenider_element will be used.

Value

An integer (backendNodeId), or an RSelenium::webElement object. get_actual_elements() returns a list of such objects.

See Also

- s(), ss(), find_element() and find_elements() to select selenider elements.
- elem_cache() and elem_cache() to cache these values.
- The Chrome Devtools Protocol documentation for the operations that can be performed using a backend node id. Note that this requires the chromote::ChromoteSession object, which can be retrieved using <selenider_session>$driver.
- The documentation for RSelenium::webElement() to see the things you can do with a webElement.

Examples

```r
html <- "
<div>
<p>Text</p>
<p>More text</p>
</div>
"

session <- minimal_selenider_session(html)

elem <- s("div") |> get_actual_element()

# The webDriver/ChromoteSession can be accessed using session$driver
driver <- session$driver

if (inherits(driver, "ChromoteSession")) {
  driver$DOM$getBoxModel(backendNodeid = elem)
} else {
  elem$getElementLocation()
```
get_page_source

}  

elems <- ss("p") |>  
  get_actual_elements()

if (inherits(driver, "ChromoteSession")) {  
  driver$DOM$describeNode(backendNodeID = elems[[1]])
} else {  
  elems[[1]]$describeElement()
}

---

get_page_source Read the HTML of a session

Description

Uses `xml2::read_html()` to read the page source of the session.

Usage

```r
get_page_source(session = NULL, ...)
```

Arguments

- `session` Optionally, a `selenider_session` object.
- `...` Passed into `xml2::read_html()`

Value

An XML document.

See Also

Other global actions: `back()`, `execute_js_fn()`, `open_url()`, `reload()`, `take_screenshot()`

Examples

```r
html <- 
  "<p>Example text</p>
  "

session <- minimal_selenider_session(html)
get_page_source()
```
get_session  

Get or set the local selenider session

Description

Change the locally defined `selenider_session()` object, allowing it to be used in functions like `s()` without explicitly providing it.

`get_session()` retrieves the current local session. If none have been created, a session is created automatically.

`local_session()` sets the local session. The function uses `withr::defer()` to make sure the session is closed and the local session is set to its previous value when it is no longer needed.

`with_session()` runs some code with a temporary local session. The session is closed and the local session is set to its previous value when the code finishes executing.

Usage

```r
get_session(create = TRUE, .env = rlang::caller_env())

local_session(session, .local_envir = rlang::caller_env(), close = TRUE)

with_session(session, code, close = TRUE)
```

Arguments

- `create`: If a session is not found, should we create a new one? If this is `FALSE` and a session is not found, `NULL` is returned.
- `.env`: If `get_session()` creates a session, the environment where this session is being used.
- `session`: The `selenider_session()` object to use.
- `.local_envir`: The environment where the session is being used. When the function associated with this environment finishes execution, the session will be reset.
- `close`: Should we close session when the local session is reset? Set this to `FALSE` if you want to use the session even if it is no longer the local session. If you want to close the session manually, use `close_session()`.
- `code`: The code to run with the local session set.

Details

Use `withr::deferred_run()` to reset any local sessions set using `local_session()`.

Value

- `get_session()` returns the local `selenider_session()` object (or a newly created session).
- `local_session()` returns the previous local session object (or `NULL`). This is the same as running `get_session()` before this function.
- `with_session()` returns the result of code.
See Also

`selenider_session()`, which calls `local_session()` unless otherwise specified.

Examples

```r
# Don't set the local session, since we want to do it manually.
session_1 <- selenider_session(local = FALSE)
session_2 <- selenider_session(local = FALSE)

get_session(create = FALSE) # NULL
local_session(session_1, close = FALSE)

get_session(create = FALSE)

withr::deferred_run()

get_session(create = FALSE) # NULL

# By default, the local session is only set inside the function that it is # called.
# If we want to set the local session outside the scope of a function, we # need to use the `.local_envir` argument.
set_my_session <- function(env = rlang::caller_env()) {
  # caller_env() is the environment where the function is called.
  local_session(session_1, .local_envir = env, close = FALSE)
}

set_my_session()

with_session(
  session_2,
  {get_session(create = FALSE),
   close = FALSE}
) # session_2

get_session(create = FALSE) # session_1
```

---

### has_attr

*Does an element's attribute match a value?*

**Description**

`has_attr()` checks that an element's attribute matches a value, while `attr_contains()` checks that an element's attribute contains a value.

`has_value()` is a shortcut for `has_attr("value")`: it checks that an element's value matches a string or number.
Usage

has_attr(x, name, value)
attr_contains(x, name, value)
has_value(x, value)

Arguments

x A selenider_element object.
name The name of the attribute.
value The value of the attribute. For has_attr() and has_value(), this can be a string or a numeric value, while attr_contains() can only take a string.

Value

A boolean value: TRUE or FALSE.

See Also

Other conditions: has_css_property(), has_length(), has_name(), has_text(), is_enabled(), is_present(), is_visible()
has_css_property

Description

Check that the CSS property (e.g. "background-color") of an element matches a value.

Usage

has_css_property(x, property, value)

Arguments

x A selenider_element object.
property The name of the CSS property
value The value of the attribute.

Value

A boolean value: TRUE or FALSE.

See Also

Other conditions: has_attr(), has_length(), has_name(), has_text(), is_enabled(), is_present(), is_visible()

Examples

html <- "
<div style='display:none;'>"</div>
"
session <- minimal_selenider_session(html)

has_css_property(s("div"), "display", "none")
has_length

Does a collection have a certain number of elements?

Description

has_length() and has_size() checks that a collection of HTML elements contains a certain number of elements.

Usage

has_length(x, n)

has_size(x, n)

has_at_least(x, n)

Arguments

x A selenider_elements object.

n A numeric vector of possible lengths of x. For has_at_least(), this must be a single number to compare to the length of x.

Details

has_at_least() checks that a collection contains at least n elements.

These functions do not implement a retry mechanism, and only test a condition once. Use elem_expect() or elem_wait_until() to use these conditions in tests.

Value

A boolean value: TRUE or FALSE

See Also

Other conditions: has_attr(), has_css_property(), has_name(), has_text(), is_enabled(), is_present(), is_visible()

Examples

html <- "
<div class='div1'></div>
<div class='div2'></div>
<div class='div3'></div>
"

session <- minimal_selenider_session(html)

has_length(ss("div"), 3)
has_name

has_at_least(ss("div"), 2)

has_name

Does an element have a tag name?

Description

Check that an element has a specified tag name

Usage

has_name(x, name)

Arguments

x       A selenider_element object.
name    A string.

Value

A boolean value.

See Also

Other conditions: has_attr(), has_css_property(), has_length(), has_text(), is_enabled(), is_present(), is_visible()

Examples

html <- "
<div id='mydiv'></div>
"

session <- minimal_selenider_session(html)

has_name(s("#mydiv"), "p")

has_name(s("#mydiv"), "div")
**Description**

`has_text()` checks that an element’s inner text contains a string, while `has_exact_text()` checks that the inner text only contains the string. Both functions throw an error if the element does not exist in the DOM.

**Usage**

```r
has_text(x, text)
has_exact_text(x, text)
```

**Arguments**

- `x`: A `selenider_element` object.
- `text`: A string, used to test the element’s inner text.

**Details**

These functions do not implement a retry mechanism, and only test a condition once. Use `elem_expect()` or `elem_wait_until()` to use these conditions in tests.

**Value**

A boolean value: TRUE or FALSE.

**See Also**

Other conditions: `has_attr()`, `has_css_property()`, `has_length()`, `has_name()`, `is_enabled()`, `is_present()`, `is_visible()`

**Examples**

```r
h <- "
<p>Example text</p>
<p class='empty'></p>
"

t <- minimal_selenider_session(h)

has_text(t("p"), "Example") # TRUE
has_exact_text(t("p"), "Example") # FALSE
```
is_enabled

    has_exact_text(s("p"), "Example text") # TRUE
    # has_exact_text() is useful for checking when there is no text,
    # since has_text("") will always be TRUE.
    has_exact_text(s(".empty"), "")

---

### is_enabled

**Is an element enabled?**

#### Description

`is_disabled()` checks that an element has the disabled attribute set to TRUE, while `is_enabled()` checks that it does not. Both functions throw an error if the element does not exist in the DOM.

#### Usage

```r
is_enabled(x)
```

```r
is_disabled(x)
```

#### Arguments

- `x` A selenider_element object.

#### Details

These functions do not implement a retry mechanism, and only test a condition once. Use `elem_expect()` or `elem_wait_until()` to use these conditions in tests.

#### Value

A boolean value: TRUE or FALSE.

#### See Also

Other conditions: `has_attr()`, `has_css_property()`, `has_length()`, `has_name()`, `has_text()`, `is_present()`, `is_visible()`

#### Examples

```r
html <- "
<button></button>
<button disabled></button>
"
session <- minimal_selenider_session(html)

is_enabled(s("button")) # TRUE

is_disabled(ss("button")[[2]]) # TRUE

---

**is_present**  
_Does an element exist?_

**Description**

`is_present()` and `is_in_dom()` checks if an element is present on the page, while `is_missing()` and `is_absent()` checks the opposite.

**Usage**

```r
is_present(x)

is_in_dom(x)

is_absent(x)
```

**Arguments**

- **x**  
  A selenider_element object.

**Details**

These functions do not implement a retry mechanism, and only test a condition once. Use `elem_expect()` or `elem_wait_until()` to use these conditions in tests.

**Value**

A boolean value: `TRUE` or `FALSE`.

**See Also**

Other conditions: `has_attr()`, `has_css_property()`, `has_length()`, `has_name()`, `has_text()`, `is_enabled()`, `is_visible()`
Examples

```r
html <- "
<p class='class1'></p>
"

session <- minimal_selenider_session(html)

is_present(s(".class1")) # TRUE
is_in_dom(s(".class2")) # FALSE
is_absent(s(".class2")) # TRUE
```

<table>
<thead>
<tr>
<th>is_visible</th>
<th>Is an element visible?</th>
</tr>
</thead>
</table>

Description

`is_visible()` and `is_displayed()` checks that an element can be seen on the page, while `is_invisible()` and `is_hidden()` checks the opposite. All functions throw an error if the element is not in the DOM.

Usage

```r
is_visible(x)

is_displayed(x)

is_hidden(x)

is_invisible(x)
```

Arguments

- `x` A `selenider_element` object.

Details

These functions do not implement a retry mechanism, and only test a condition once. Use `elem_expect()` or `elem_wait_until()` to use these conditions in tests.

Value

A boolean value: TRUE or FALSE.
See Also

Other conditions: `has_attr()`, `has_css_property()`, `has_length()`, `has_name()`, `has_text()`, `is_enabled()`, `is_present()`

Examples

```r
html <- "
<div style='visibility:hidden;'>Content 1</div>
<div style='display:none'>Content 2</div>
<div>Content 3</div>
"

session <- minimal_selenider_session(html)

is_visible(s("div")) # FALSE
is_invisible(ss("div")[[2]]) # TRUE
is_visible(ss("div")[[3]]) # TRUE
```

---

### keys

#### Special keys

<table>
<thead>
<tr>
<th>keys</th>
<th>Special keys</th>
</tr>
</thead>
</table>

**Description**

List of special keys, for use with `elem_send_keys()`.

**Usage**

`keys`

**Format**

A list containing `selenider_key` objects.

**Examples**

`keys$backspace`
minimal_selenider_session

Create a session with custom HTML

Description

Create a selenider_session using custom HTML/JavaScript.

Usage

minimal_selenider_session(html, js = NULL, ..., .env = rlang::caller_env())

Arguments

html A string to use as HTML. Can also be an xml2 object.
js A string (or NULL) to use as JavaScript.
... Passed into selenider_session().
.env The environment in which the session will be used.

Details

The function works by combining html and js into a single string, then writing this to a temporary file (and opening it in the session’s browser).

Value

A selenider_session object.

See Also

selenider_session()

Examples

session <- minimal_selenider_session("<p>Example</p>"
open_url

Open a URL

Description

Navigate the browser to specified URL, waiting until the page is considered open before finishing.

Usage

open_url(url, session = NULL)

Arguments

url  
The URL to navigate to: a string.

session  
A `selenider_session` object. If not specified, the global session object (the result of `get_session()`) is used.

Value

The session object, invisibly.

See Also

Other global actions: `back()`, `execute_js_fn()`, `get_page_source()`, `reload()`, `take_screenshot()`

Examples

```r
session <- selenider_session()
open_url("https://r-project.org")

# Or:
open_url(session = session, "https://r-project.org")
```
Description

*xml2::read_html()* can be used on a selenider session to read the HTML of the entire page, or on a selenider element to get the HTML of that element.

Usage

```r
read_html.selenider_session(
  x,
  encoding = "",
  ..., 
  options = c("RECOVER", "NOERROR", "NOBLANKS")
)
```

```r
read_html.selenider_element( 
  x, 
  encoding = "",
  timeout = NULL, 
  outer = TRUE, 
  ..., 
  options = c("RECOVER", "NOERROR", "NOBLANKS")
)
```

Arguments

- `x` A `selenider_session/selenider_element` object.
- `encoding`, `...`, `options` Passed into `xml2::read_html()`.
- `timeout` How long to wait for `x` to exist in the DOM before throwing an error.
- `outer` Whether to read the inner (all children of the current element) or outer (including the element itself) HTML of `x`.

Value

`read_html()` returns an XML document. Note that HTML will always be wrapped in a `<html>` and `<body>` tag, if it isn’t already.

Examples

```r
library(rvest)

html <- ""
```
reload

Reload the current page

Description
reload() and refresh() both reload the current page.

Usage
reload(session = NULL)

refresh(session = NULL)

Arguments

session A `selenider_session` object. If not specified, the global session object (the result of `get_session()`) is used.

Value
The session object, invisibly.

See Also
Other global actions: back(), execute_js_fn(), get_page_source(), open_url(), take_screenshot()

Examples

```r
session <- selenider_session()
open_url("https://r-project.org")
reload()
```
**s**  

*Select HTML elements*

**Description**

Both `s()` and `ss()` allow you to select elements without specifying a session object.

`s()` selects a single element, being a shorthand for `find_element()` on the current session.

`ss()` selects multiple elements, being a shorthand for `find_elements()`.

**Usage**

```r
s(
    css = NULL,
    xpath = NULL,
    id = NULL,
    class_name = NULL,
    name = NULL,
    link_text = NULL
)

ss(
    css = NULL,
    xpath = NULL,
    id = NULL,
    class_name = NULL,
    name = NULL,
    link_text = NULL
)
```

**Arguments**

- `css`  
  A css selector.
- `xpath`  
  An XPath.
- `id`  
  The id of the element you want to select.
- `class_name`  
  The class name of the element you want to select.
- `name`  
  The name attribute of the element you want to select.
- `link_text`  
  The link text of the link element that you would like to select.

**Details**

Both functions allow the starting point for chains of selectors to be made more concise. Both use `get_session()` to get the global session object.

**Value**

`s()` returns a `selenium_element` object. `ss()` returns a `selenium_elements` object.
See Also

- `find_element()` and `find_elements()`
- `selenider_session()` to begin a session.

Examples

```r
html <- "
<div>
  <p id='id1' class='inner'></p>
  <div class='child'>
    <p class='inner'></p>
  </div>
</div>
"

session <- minimal_selenider_session(html)

s("#id1")

# This is the equivalent of:
find_element(session, "#id1")

ss(".inner")

# This is the equivalent of:
find_element(session, ".inner")

# This provides a more concise way to begin a chain of selectors
s("div") |> 
  find_element(".child") |> 
  find_element(".inner")
```

Selenider options

<table>
<thead>
<tr>
<th>Selenider options</th>
</tr>
</thead>
<tbody>
<tr>
<td>selenider-config</td>
</tr>
</tbody>
</table>

Description

`selenider` has a few options, allowing you to specify the session and browser to use without having to tell `selenider_session()` this information every time.

- `selenider.session` - The package to use as a backend: either "chromote" or "selenium".
- `selenider.browser` - The name of the browser to run the session in: one of "chrome", "firefox", "phantomjs" or "internet explorer" (only on Windows).
**selenider_available**  
*Check if selenider can be used*

**Description**
Checks if selenider’s dependencies are available, and that we are in an environment where it makes sense to open a selenider session.
skip_if_selenider_unavailable() skips a test that test if selenider_available() returns FALSE.

**Usage**
selenider_available(session = c("chromote", "selenium"), online = TRUE)
skip_if_selenider_unavailable(session = c("chromote", "selenium"))

**Arguments**

- **session** Which session we should check. "chromote" is used by default.
- **online** Whether we need to check for an internet connection.

**Details**
Specifically, the following is checked:

- The `SELENIDER_AVAILABLE` environment variable. Set this to "TRUE" or "FALSE" to override this function.
- Whether we are on CRAN (using the `NOT_CRAN` environment variable). If we are, the function returns FALSE.
- Whether an internet connection is available (using `curl::nslookup()`).

If `session` is "chromote", we also check:

- Whether `chromote` is installed.
- Whether `chromote::find_chrome()` does not error.

If `session` is "selenium", we check:

- Whether `RSelenium` is installed.
- Whether we can find a valid browser that is supported by `RSelenium`.

**Value**
A boolean flag: TRUE or FALSE.

**Examples**
selenider_available()
selenider_session

Start a session

Description

Begin a session in selenider, setting it as the local session unless otherwise specified, allowing the session to be accessed globally in the environment where it was defined.

create_chromote_session(), create_selenium_client() and create_selenium_server() are low-level functions that allow more control over making a web driver, which can then be passed into the driver argument to selenider_session().

Usage

selenider_session(
    session = getOption("selenider.session"),
    browser = getOption("selenider.browser"),
    view = FALSE,
    timeout = 4,
    driver = NULL,
    local = TRUE,
    quiet = TRUE,
    .env = rlang::caller_env()
)

create_chromote_session(...)

create_selenium_server(
    browser,
    version = "latest",
    port = 4567L,
    quiet = TRUE,
    ...
)

create_selenium_client(browser, port = 4567L, ...)

Arguments

session The package to use as a backend: either "chromote" or "selenium". By default, chromote is used, since this tends to be faster and more reliable. Change the default value using the selenider.session option.

browser The name of the browser to run the session in; one of "chrome", "firefox", "phantomjs" or "internet explorer" (only on Windows). IF NULL, the function will try to work out which browser you have installed. If we are using chromote, this option is ignored, since chromote only works on Chrome. Change the default value of this parameter using the selenider.browser option.
Whether to open the browser and view it, for visual testing. This is ignored if session is "selenium", since selenium drivers cannot be headless.

The default time to wait when collecting an element.

A driver object to use instead of creating one manually. This can be one of:
- A `chromote::ChromoteSession` object (the result of `create_chromote_session()`).
- A `shinytest2::AppDriver` object.
- An `R Selenium::remoteDriver()` object (the result of `create_selenium_client()`).
- A Selenium server object (the result of `wdman::selenium()`, or `create_selenium_server()`).

In this case, the client object will be created using the server object.
- A list/environment containing the `R Selenium::remoteDriver()` object, the Selenium server object, or both. See Details for more information about providing a custom driver object.

Whether to set the session as the local session object, using `local_session()`.

Whether to let `R Selenium::rsDriver()` display messages. By default, this output is suppressed, as it is not usually useful. Chromote does not display any output when creating a session.

Passed into `local_session()` function, to define the environment in which the session is used. Change this if you want to create the session inside a function and then use it outside the function.

Arguments to finetune the creation of the specific driver.
- For `create_chromote_session()`, these are passed into `chromote::ChromoteSession$new()`.
- For `create_selenium_server()`, these are passed into `wdman::selenium()`.
- For `create_selenium_server()`, these are passed into `R Selenium::remoteDriver()`.

The version of the webdriver (chromedriver, geckodriver, etc.) to use. You should only need to change this if Chrome is being used, as the version of the driver depends on the version of Chrome.

The port to run Selenium on.

A `selenider_session` object. Use `session$driver` to retrieve the driver object that controls the browser.

A `selenider_session` object is an S3 list, meaning its properties can be accessed using $. Most notably, using `session$driver` allows access to the driver object which actually controls the browser. If you are using Selenium, use `session$driver$client` to access the `remoteDriver` object. These objects are useful if you want to do something with the driver that is not directly supported by selenider. See `get_actual_element()` for some examples of this.

Custom drivers

Custom driver objects are good if you want more low-level control over the underlying functions that create the webdrivers that actually control the browser. However, it is recommended to use
the selenider functions (e.g. create_selenium_client()) over wdman::selenium() for better error messages and more reliable behaviour. See vignette("unit-testing", package = "selenider") for more information on using selenider with docker/Github Actions.

**Chromote:**
Supplying a custom chromote::ChromoteSession object can allow you to manage the underlying chromote::Chromote process that is used to spawn sessions. For example:

```r
my_chromote_object <- chromote::Chromote$new()

session <- selenider_session(
  driver = create_chromote_session(parent = my_chromote_object)
)
```

You can also supply a shinytest2::AppDriver object, allowing selenider and shinytest2 to share a session:

```r
shiny_app <- shiny::shinyApp(
  ui = shiny::fluidPage(
      # ... Your UI
  ),
  server = function(input, output) {
      # ... Your server
  }
)

app <- shinytest2::AppDriver$new()

session <- selenider_session(
  driver = app
)
```

**RSelenium:**
If you want to manually create both the client and the server, you can do the equivalent of the following:

```r
session <- selenider_session(
  driver = list(
      client = create_selenium_client("chrome"),
      server = create_selenium_server("chrome")
  )
)
```

However, it can sometimes be useful to omit the server, for example when you are running the Selenium server using Docker. In this case, you need to make sure the ip and port are matched correctly.

```r
session <- selenider_session(
  driver = create_selenium_client(
      remoteServerAddr = "<IP ADDRESS>",
      port = 1234L
  )
)
```
See Also

- `close_session()` to close the session. Note that this will not reset the result of `get_session()`, which is why `withr::deferred_run()` is preferred.
- `local_session()` and `with_session()` to manually set the local session object (and `get_session()`) to get it.
- `open_url()`, `s()` and `find_elements()` to get started once you have created a session.

Examples

```r
session_1 <- selenider_session(timeout = 10)
# session_1 is the local session here
get_session() # Returns session 1

my_function <- function() {
  session_2 <- selenider_session()
  # In here, session_2 is the local session
  get_session()
  # When the function finishes executing, the session is closed
}
my_function() # Returns `session_2`

# If we want to use a session outside the scope of a function, # we need to use the `.env` argument.
create_session <- function(timeout = 10, .env = rlang::caller_env()) {
  # caller_env() is the environment where the function is called
  selenider_session(timeout = timeout, .env = .env)
}

my_session <- create_session()

# We can now use this session outside the `create_session()` function
get_session()

# `my_session` will be closed automatically.
```

Description

Take a screenshot of the current session state, saving this image to a file.
Usage

\[
\text{take\_screenshot}(file = \text{NULL}, \text{view} = \text{FALSE}, \text{session} = \text{NULL})
\]

Arguments

- **file**: The file path to save the screenshot to.
- **view**: Whether to open the interactively view the screenshot. If this is TRUE and file is NULL, the screenshot will be deleted after viewing.
- **session**: A `selenider_session` object. If not specified, the global session object (the result of `get_session()`) is used.

Value

file, if it is not NULL. Otherwise, the session object is returned, invisibly.

See Also

Other global actions: `back()`, `execute_js_fn()`, `get_page_source()`, `open_url()`, `reload()`

Examples

```r
session <- selenider_session()

ośćpen_url("https://www.google.com")

file_path <- withr::local_tempfile(fileext = "png")

take_screenshot(file_path)
```
Index

* actions
  elem_click, 12
  elem_hover, 24
  elem_scroll_to, 26
  elem_set_value, 27
  elem_submit, 30

* conditions
  has_attr, 41
  has_css_property, 43
  has_length, 44
  has_name, 45
  has_text, 46
  is_enabled, 47
  is_present, 48
  is_visible, 49

* datasets
  keys, 50

* global actions
  back, 5
  execute_js_fn, 32
  get_page_source, 39
  open_url, 52
  reload, 54
  take_screenshot, 61

* properties
  elem_attr, 9
  elem_css_property, 13
  elem_name, 25
  elem_size, 29
  elem_text, 31
  ==.selenider_element(elem_equal), 14
  [.selenider_elements(elem_filter), 21
  [[.selenider_elements(elem_filter), 21

  anonymous function syntax, 16
  as.list.selenider_elements, 3
  as.list.selenider_elements(), 23
  attr_contains(has_attr), 41

  back, 5, 33, 39, 52, 54, 62

  c.selenider_element(elem_flatten), 22
  c.selenider_elements(elem_flatten), 22
  chromote::Chromote, 60
  chromote::ChromoteSession, 38, 59, 60
  chromote::ChromoteSession$new(), 59
  chromote::find_chrome(), 57
  close_session, 6
  close_session(), 40, 61
  create_chromote_session
    (selenider_session), 58
  create_selenium_client
    (selenider_session), 58
  create_selenium_server
    (selenider_session), 58
  curl::nslookup(), 57

  elem_ancestors, 7
  elem_attr, 9, 14, 25, 29, 31
  elem_attrs(elem_attr), 9
  elem_cache, 10
  elem_cache(), 32, 38
  elem_children(elem_ancestors), 7
  elem_children(), 36
  elem_clear_value(elem_set_value), 27
  elem_click, 12, 24, 26, 28, 30
  elem_css_property, 10, 13, 25, 29, 31
  elem_descendants(elem_ancestors), 7
  elem_double_click(elem_click), 12
  elem_equal, 14
  elem_expect, 15
  elem_expect(), 19–21, 44, 46–49
  elem_expect_all, 19
  elem_expect_all(), 17
  elem_filter, 21
  elem_filter(), 4, 8, 15, 17, 36
  elem_find(elem_filter), 21
  elem_find(), 4, 8, 15, 17, 36
  elem_flatmap
    (as.list.selenider_elements), 3
    elem_flatmap(), 3, 11
elem_flatten, 22
elem_flatten(), 3, 4
elem_hover, 12, 24, 26, 28, 30
elem_name, 10, 14, 25, 29, 31
elem_parent (elem_ancestors), 7
elem_right_click (elem_click), 12
elem_scroll_to, 12, 24, 26, 28, 30
elem_send_keys (elem_set_value), 27
elem_send_keys(), 27, 50
elem_set_value, 12, 24, 26, 27, 30
elem_siblings (elem_ancestors), 7
elem_size, 10, 14, 25, 29, 31
elem_submit, 12, 24, 26, 28, 30
elem_text, 10, 14, 25, 29, 31
elem_value (elem_attr), 9
elem_wait_until (elem_expect), 15
elem_wait_until (), 19, 20, 44, 46–49
elem_wait_until_all (elem_expect_all), 19
elem_wait_until_all (), 17
element_list
  (as.list.selenider_elements), 3
  element_list (), 11
execute_js_expr (execute_js_fn), 32
execute_js_fn, 5, 32, 39, 52, 54, 62
find_element, 33
find_element (), 8, 11, 21, 36, 38, 55, 56
find_elements, 35
find_elements (), 8, 11, 21, 22, 34, 38, 55, 56, 61
forward (back), 5
get_actual_element, 37
get_actual_element (), 59
get_actual_elements
  (get_actual_element), 37
get_page_source, 5, 33, 39, 52, 54, 62
get_session, 40
get_session (), 5, 52, 54, 55, 61, 62
has_at_least (has_length), 44
has_at_least (), 20
has_attr, 41, 43–48, 50
has_css_property, 42, 43, 44–48, 50
has_exact_text (has_text), 46
has_length, 42, 43, 44, 45–48, 50
has_name, 42–44, 45, 46–48, 50
has_size (has_length), 44
has_text, 42–45, 46, 47, 48, 50
has_value (has_attr), 41
is_absent (is_present), 48
is_disabled (is_enabled), 47
isDisplayed (is_visible), 49
is_enabled, 42–46, 47, 48, 50
is_hidden (is_visible), 49
is_in_dom (is_present), 48
is_invisible (is_visible), 49
is_present, 42–47, 48, 50
is_present (), 17, 20–22
is_visible, 42–48, 49
keys, 50
lapply (), 3
length.selenider_elements (elem_size), 29
local_session (get_session), 40
local_session (), 59, 61
minimal_selenider_session, 51
open_url, 5, 33, 39, 52, 54, 62
open_url (), 61
purrr::every (), 20
purrr::map (), 3
read_html.selenider_element
  (read_html.selenider_session), 53
read_html.selenider_session, 53
refresh (reload), 54
reload, 5, 33, 39, 52, 54, 62
RSelenium::remoteDriver (), 59
RSelenium::rsDriver (), 59
RSelenium::webElement, 37, 38
RSelenium::webElement (), 38
s, 55
s (), 34, 38, 40, 61
selenider-config, 56
selenider_available, 57
selenider_session, 58
selenider_session (), 6, 34, 36, 40, 41, 51, 56
shinytest2::AppDriver, 59, 60
INDEX

skip_if_selenider_unavailable
  (selenider_available), 57
ss(s), 55
ss(), 22, 36, 38

take_screenshot, 5, 33, 39, 52, 54, 61
testthat::is_testing(), 16, 19

wdman::selenium(), 59
with_session(get_session), 40
with_session(), 61
withr::defer(), 40
withr::deferred_run(), 40, 61

xml2::read_html(), 39, 53