

# Package ‘lenses’

May 8, 2026

**Version** 0.0.3

**Title** Elegant Data Manipulation with Lenses

**Description**

Provides tools for creating and using lenses to simplify data manipulation. Lenses are composable getter/setter pairs for working with data in a purely functional way. Inspired by the 'Haskell' library 'lens' (Kmett, 2012) <<https://hackage.haskell.org/package/lens>>. For a fairly comprehensive (and highly technical) history of lenses please see the 'lens' wiki <<https://github.com/ekmett/lens/wiki/History-of-Lenses>>.

**License** MIT + file LICENSE

**Encoding** UTF-8

**LazyData** true

**ByteCompile** true

**RoxygenNote** 6.0.1

**URL** <http://cfhammill.github.io/lenses>,  
<https://github.com/cfhammill/lenses>

**BugReports** <https://github.com/cfhammill/lenses/issues>

**Suggests** testthat

**Imports** magrittr, tidyselect, rlang

**Collate** 'verbs.R' 'lens.R' 'array-lenses.R' 'base-lenses.R' 'utils.R'  
'dataframe-lenses.R' 'utils-pipe.R'

**NeedsCompilation** no

**Author** Chris Hammill [aut, cre, trl],  
Ben Darwin [aut, trl]

**Maintainer** Chris Hammill <[cfhammill@gmail.com](mailto:cfhammill@gmail.com)>

**Repository** CRAN

**Date/Publication** 2019-03-06 14:40:03 UTC

## Contents

attributes_1	3
attr_1	3
body_1	4
class_1	4
colnames_1	5
cols_1	5
cond_il	6
c_1	6
diag_1	7
dimnames_1	7
dim_1	8
drop_while_il	8
env_1	9
filter_il	9
filter_1	10
first_1	10
formals_1	11
id_1	11
indexes_1	12
index_1	12
last_1	13
lens	13
levels_1	14
lower_tri_1	15
map_1	15
names_1	16
oscope	16
over	17
over_map	17
over_with	18
reshape_1	18
rev_1	19
rownames_1	19
rows_1	20
select_1	20
send	21
send_over	21
set	22
slab_1	22
slice_1	23
slot_1	23
take_1	24
take_while_il	24
to_1	25
transpose_1	25
t_1	26

<i>attributes_l</i>	3
unlist_l . . . . .	26
upper_tri_l . . . . .	27
view . . . . .	27
%.% . . . . .	28
<b>Index</b>	<b>29</b>

---

<i>attributes_l</i>	<i>Attributes lens</i>
---------------------	------------------------

---

**Description**

The lens equivalent of `attributes` and `attributes<-`

**Usage**

```
attributes_l
```

**Format**

An object of class lens of length 2.

**Examples**

```
(x <- structure(1:10, important = "attribute"))
view(x, attributes_l)
set(x, attributes_l, list(important = "feature"))
```

---

<i>attr_l</i>	<i>Construct a lens into an attribute</i>
---------------	---

---

**Description**

The lens version of `attr` and `attr<-`

**Usage**

```
attr_l(attrib)
```

**Arguments**

`attrib`            A length one character vector indicating the attribute to lens into.

**Examples**

```
(x <- structure(1:10, important = "attribute"))
view(x, attr_l("important"))
set(x, attr_l("important"), "feature")
```

---

`body_1`*Body lens*

---

**Description**

A lens into the body of a function. The lens equivalent of `body` and `body<-`. You probably shouldn't use this.

**Usage**`body_1`**Format**

An object of class `lens` of length 2.

**Examples**

```
inc2 <- function(x) x + 2
view(inc2, body_1)
inc4 <- set(inc2, body_1, quote(x + 4))
inc4(10)
```

---

`class_1`*Class lens*

---

**Description**

A lens into the class of an object. Lens equivalent of `class` and `class<-`.

**Usage**`class_1`**Format**

An object of class `lens` of length 2.

**Examples**

```
x <- 1:10
view(x, class_1)
set(x, class_1, "super_integer")
```

---

colnames_l	<i>A lens into the column names of an object</i>
------------	--

---

**Description**

The lens version of colnames and colnames<-

**Usage**

```
colnames_l
```

**Format**

An object of class lens of length 2.

**Examples**

```
x <- matrix(1:4, ncol = 2)
colnames(x) <- c("first", "second")
x

view(x, colnames_l)
set(x, colnames_l, c("premiere", "deuxieme"))
```

---

cols_l	<i>Column lens</i>
--------	--------------------

---

**Description**

Create a lens into a set of columns

**Usage**

```
cols_l(cols, drop = FALSE)
```

**Arguments**

cols	the columns to focus on
drop	whether or not to drop dimensions with length 1

**Examples**

```
x <- matrix(1:4, ncol = 2)
colnames(x) <- c("first", "second")
x

view(x, cols_l(1))
view(x, cols_l("second"))
set(x, cols_l(1), c(20,40))
```

---

<code>cond_il</code>	<i>Conditional lens</i>
----------------------	-------------------------

---

**Description**

`view` is equivalent to `Filter(f, d)`, `set` replaces elements that satisfy `f` with elements of `x`.

**Usage**

```
cond_il(f)
```

**Arguments**

`f`                      the predicate (logical) function

**Details**

This lens is illegal because `set-view` is not satisfied, multiple runs of the same lens will reference potentially different elements.

---

<code>c_l</code>	<i>Convenient lens composition</i>
------------------	------------------------------------

---

**Description**

A lens version of `purrr::pluck`. Takes a series element indicators and creates a composite lens.

**Usage**

```
c_l(...)
```

**Arguments**

`...`                      index vectors or lenses

**Details**

- length one vectors are converted to `index_l`,
- length one logical vectors and numeric vectors that are negative are converted to `indexes_l`,
- larger vectors are converted to `indexes_l`,
- lenses are composed as is.

See examples for more

**Examples**

```
view(iris, c_l("Petal.Length", 10:20, 3))
sepal_l <- index("Sepal.Length")
view(iris, c_l(sepal_l, id_l, 3))
```

---

diag_l	<i>Lens into the diagonal of a matrix</i>
--------	---

---

**Description**

A lens into a matrix's diagonal elements

**Usage**

```
diag_l
```

**Format**

An object of class lens of length 2.

---

dimnames_l	<i>Dimnames lens</i>
------------	----------------------

---

**Description**

A lens into the dimnames of an object. Lens equivalent of [dimnames](#) and [dimnames<-](#).

**Usage**

```
dimnames_l
```

**Format**

An object of class lens of length 2.

**Examples**

```
x <- matrix(1:4, ncol = 2)
colnames(x) <- c("first", "second")
x

view(x, dimnames_l)
set(x, dimnames_l, list(NULL, c("premiere", "deuxieme")))
```

dim\_1

*Dims lens*

---

**Description**

A lens into an objects dimensions

**Usage**

```
dim_1
```

**Format**

An object of class lens of length 2.

**Examples**

```
x <- 1:10  
  
(y <- set(x, dim_1, c(2,5)))  
view(y, dim_1)
```

---

drop\_while\_il

*Conditional trim lens*

---

**Description**

A lens into all elements starting from the first element that doesn't satisfy a predicate. Essentially the complement of [take\\_while\\_il](#)

**Usage**

```
drop_while_il(f)
```

**Arguments**

f                    the predicate (logical) function

---

env_l	<i>Environment lens</i>
-------	-------------------------

---

**Description**

A lens into the environment of an object. This is the lens version of [environment](#) and [environment<-](#)

**Usage**

```
env_l
```

**Format**

An object of class lens of length 2.

**Examples**

```
x <- 10
f <- (function(){x <- 2; function() x + 1})()
f

f()
view(f, env_l)$x

g <- over(f, env_l, parent.env)
g()
```

---

filter_il	<i>Filter lens</i>
-----------	--------------------

---

**Description**

Create an illegal lens into the result of a filter. Arguments are interpreted with non-standard evaluation as in [dplyr::filter](#)

**Usage**

```
filter_il(...)
```

**Arguments**

... unquoted NSE filter arguments

**Examples**

```
head(view(iris, filter_il(Species == "setosa")))
head(over(iris,
  filter_il(Species == "setosa") %.% select_1(-Species),
  function(x) x + 10))
```

---

 filter\_l

*Filter lens*


---

### Description

Create a lawful lens into the result of a filter. This focuses only columns not involved in the filter condition.

### Usage

```
filter_l(...)
```

### Arguments

... unquoted NSE filter arguments

### Examples

```
head(view(iris, filter_l(Species == "setosa"))) # Note Species is not seen
head(over(iris, filter_l(Species == "setosa"), function(x) x + 10))
```

---

 first\_l

*A lens into the first element*


---

### Description

Lens version of `x[[1]]` and `x[[1]] <- val` `x <- 1:10` `view(x, first_l)` `set(x, first_l, 50)`

```
[[1]: R:[1] [[1]: R:[1]
```

### Usage

```
first_l
```

### Format

An object of class lens of length 2.

---

`formals_1`*Formals lens*

---

**Description**

A lens equivalent of `formals` and `formals<-`, allowing you to change the formal arguments of a function. As with `body_1` you probably shouldn't use this.

**Usage**

```
formals_1
```

**Format**

An object of class `lens` of length 2.

**Examples**

```
f <- function(x) x + y + 7
view(f, formals_1)

g <- set(f, formals_1, list(x = 1, y = 2))
g()
```

---

`id_1`*The identity (trivial lens)*

---

**Description**

This lens focuses on the whole object

**Usage**

```
id_1
```

**Format**

An object of class `lens` of length 2.

**Examples**

```
x <- 1:10
view(x, id_1)
head(set(x, id_1, iris))
```

---

indexes\_l                      *Construct a lens into a subset of an object*

---

### Description

This is the lens version of []

### Usage

```
indexes_l(els)
```

```
indexes(els)
```

### Arguments

els                      a subset vector, can be integer, character or logical, pointing to one or more elements of the object

### Functions

- indexes: shorthand

### Examples

```
x <- 1:10
view(x, indexes_l(3:5))
set(x, indexes_l(c(1,10)), NA)
head(view(iris, indexes_l(c("Sepal.Length", "Species"))))
```

---

index\_l                      *Construct a lens into an index/name*

---

### Description

This is the lens version of [[]

### Usage

```
index_l(e1)
```

```
index(e1)
```

### Arguments

e1                      The element the lens should point to can be an integer or name.

**Functions**

- index: shorthand

**Examples**

```
x <- 1:10
view(x, index_1(1))
set(x, index(5), 50)
head(view(iris, index(2)))
```

---

last_1	<i>A lens into the last element</i>
--------	-------------------------------------

---

**Description**

Lens version of `x[[length(x)]]` and `x[[length(x)]] <- val`  
`[[length(x)]: R:[length(x)]]` `[[length(x)]: R:[length(x)]`

**Usage**

```
last_1
```

**Format**

An object of class lens of length 2.

**Examples**

```
x <- 1:10
view(x, last_1)
set(x, last_1, 50)
```

---

lens	<i>Construct a lens</i>
------	-------------------------

---

**Description**

A lens represents the process of focusing on a specific part of a data structure. We represent this via a view function and an set function, roughly corresponding to object-oriented "getters" and "setters" respectively. Lenses can be composed to access or modify deeply nested structures.

**Usage**

```
lens(view, set, getter = FALSE)
```

**Arguments**

view	A function that takes a data structure of a certain type and returns a subpart of that structure
set	A function that takes a data structure of a certain type and a value and returns a new data structure with the given subpart replaced with the given value. Note that set should not modify the original data.
getter	Default is FALSE, if TRUE the created lens cannot be set into.

**Details**

Lenses are popular in functional programming because they allow you to build pure, compositional, and re-usable "getters" and "setters".

As noted in the README, using lens directly incurs the following obligations (the "Lens laws"):

1. Get-Put: If you get (view) some data with a lens, and then modify (set) the data with that value, you get the input data back.
2. Put-Get: If you put (set) a value into some data with a lens, then get that value with the lens, you get back what you put in.
3. Put-Put: If you put a value into some data with a lens, and then put another value with the same lens, it's the same as only doing the second put.

"Lenses" which do not satisfy these properties should be documented accordingly. By convention, such objects present in this library are suffixed by "\_il" ("illegal lens").

**Examples**

```
third_l <- lens(view = function(d) d[[3]],
              set = function(d, x){ d[[3]] <- x; d })
view(1:10, third_l) # returns 3
set(1:10, third_l, 10) # returns c(1:2, 10, 4:10)
```

---

 levels\_1

*Levels lens*


---

**Description**

A lens into the levels of an object. Usually this is factor levels. Lens equivalent of [levels](#) and [levels<-](#).

**Usage**

```
levels_1
```

**Format**

An object of class lens of length 2.

**Examples**

```
x <- factor(c("a", "b"))
view(x, levels_l)
set(x, levels_l, c("A", "B"))
```

---

lower\_tri\_l                      *Lens into lower diagonal elements*

---

**Description**

Create a lens into the lower diagonal elements of a matrix

**Usage**

```
lower_tri_l(diag = FALSE)
```

**Arguments**

diag                      whether or not to include the diagonal

**Examples**

```
(x <- matrix(1:9, ncol = 3))
view(x, lower_tri_l())
view(x, lower_tri_l(diag = TRUE))
set(x, lower_tri_l(), c(100, 200, 300))
```

---

map\_l                              *Promote a lens to apply to each element of a list*

---

**Description**

Create a new lens that views and sets each element of the list.

**Usage**

```
map_l(l)
```

**Arguments**

l                              the lens to promote

**Details**

Uses [lapply](#) under the hood for [view](#) and [mapply](#) under the hood for [set](#). This means that [set](#) can be given a list of values to set, one for each element. If the input or update are lists this lens always returns a list. If the input and update are vectors this lens will return a vector.

**Examples**

```
(ex <- replicate(10, sample(1:5), simplify = FALSE))
view(ex, map_l(index(1)))
set(ex, map_l(index(1)), 11:20)
```

---

names_l	<i>A lens into the names of an object</i>
---------	---

---

**Description**

The lens versions of names and names<-.

**Usage**

```
names_l
```

**Format**

An object of class lens of length 2.

**Examples**

```
view(iris, names_l)
head(set(iris, names_l, LETTERS[1:5]))
```

---

oscope	<i>Bind data to a lens</i>
--------	----------------------------

---

**Description**

To flatten lens composition, you can prespecify the data the lens will be applied to by constructing an objectscope. These can be integrated easily with normal data pipelines.

**Usage**

```
oscope(d, l = id_l)
```

**Arguments**

d	The data for interest
l	The lens to bind the data to. Defaults to the identity lens

**Examples**

```
list(a = 5, b = 1:3, c = 8) %>%
  oscope()   %.%
  index_l("b") %.%
  index_l(1) %>%
  set(10)
```

---

`over`*Map a function over a lens*

---

**Description**

Get the data pointed to by a lens, apply a function and replace it with the result.

**Usage**

```
over(d, l, f)
```

**Arguments**

<code>d</code>	the data (or an <a href="#">oscope</a> )
<code>l</code>	the lens (or the function if <code>d</code> is an oscope)
<code>f</code>	the function (or nothing if <code>d</code> is an oscope)

**Examples**

```
third_l <- index(3)
over(1:5, third_l, function(x) x + 2)
# returns c(1:2, 5, 4:5)
```

---

`over_map`*Map a function over a list lens*

---

**Description**

Apply the specified function to each element of the subobject.

**Usage**

```
over_map(d, l, f)
```

**Arguments**

<code>d</code>	the data
<code>l</code>	the lens
<code>f</code>	the function to use, potentially a <code>~</code> specified anonymous function.

---

over_with	<i>Map a function over an in scope lens</i>
-----------	---

---

**Description**

Apply the specified function with named elements of the viewed data in scope. Similar to [dplyr::mutate](#)

**Usage**

```
over_with(d, l, f)
```

**Arguments**

d	the data
l	the lens
f	the function to use, potentially a ~ specified anonymous function. The function body is quoted, and evaluated with <code>rlang::eval_tidy(..., data = view(d,l))</code>

**Examples**

```
iris %>% over_with(id_l, ~ Sepal.Length)
```

---

reshape_1	<i>Lens into a new dimension(s)</i>
-----------	-------------------------------------

---

**Description**

Construct a lens that is a view of the data with a new set of dimensions. Both [view](#) and [set](#) check that the new dimensions match the number of elements of the data.

**Usage**

```
reshape_1(dims)
```

**Arguments**

dims	a vector with the new dimensions
------	----------------------------------

**Examples**

```
x <- 1:9
view(x, reshape_1(c(3,3)))
set(x, reshape_1(c(3,3)) %.% diag_1, 100)
```

---

rev_l	<i>Reverse lens</i>
-------	---------------------

---

**Description**

Lens into the [reverse](#) of an object.

**Usage**

```
rev_l
```

**Format**

An object of class lens of length 2.

**Examples**

```
x <- 1:10
view(x, rev_l)
set(x, rev_l, 11:20)
```

---

rownames_l	<i>A lens into the row names of an object</i>
------------	---

---

**Description**

The lens version of rownames and rownames<-

**Usage**

```
rownames_l
```

**Format**

An object of class lens of length 2.

**Examples**

```
x <- matrix(1:4, ncol = 2)
rownames(x) <- c("first", "second")
x

view(x, rownames_l)
set(x, rownames_l, c("premiere", "deuxieme"))
```

---

rows_l	<i>Row lens</i>
--------	-----------------

---

**Description**

Create a lens into a set of rows

**Usage**

```
rows_l(rows, drop = FALSE)
```

**Arguments**

rows	the rows to focus on
drop	whether or not to drop dimensions with length 1

**Examples**

```
x <- matrix(1:4, ncol = 2)
rownames(x) <- c("first", "second")
x

view(x, rows_l(1))
view(x, rows_l("second"))
set(x, rows_l(1), c(20,40))
```

---

select_l	<i>Tidysselect elements by name</i>
----------	-------------------------------------

---

**Description**

Create a lens into a named collection. On [set](#) names of the input are not changed. This generalizes [dplyr::select](#) to arbitrary named collections and allows updating.

**Usage**

```
select_l(...)
```

**Arguments**

...	An expression to be interpreted by <a href="#">tidysselect::vars_select</a> which is the same interpreter as <a href="#">dplyr::select</a>
-----	--

**Examples**

```
lets <- setNames(seq_along(LETTERS), LETTERS)
set(lets, select_l(G:F, A, B), 1:4) # A and B are 3,4 for a quick check
```

---

send	<i>Set one lens to the view of another</i>
------	--

---

**Description**

Set one lens to the view of another

**Usage**

```
send(d, l, m)
```

**Arguments**

d	the data
l	the lens to view through
m	the lens to set into

---

send_over	<i>Set one lens to the view of another (transformed)</i>
-----------	--

---

**Description**

Set one lens to the view of another (transformed)

**Usage**

```
send_over(d, l, m, f)
```

**Arguments**

d	the data
l	the lens to view through
m	the lens to set into
f	the function to apply to the viewed data

---

set	<i>Modify data with a lens</i>
-----	--------------------------------

---

### Description

Set the subcomponent of the data referred to by a lens with a new value. See [lens](#) for details. Merely dispatches to the set component of the lens.

### Usage

```
set(d, l, x)
```

### Arguments

d	the data, or an <a href="#">oscope</a>
l	the lens, or in the case of an oscope, the replacement
x	the replacement value, or nothing in the case of an oscope

---

slab_1	<i>Slab lens</i>
--------	------------------

---

### Description

Create a lens into a chunk of an array (hyperslab). Uses the same syntactic rules as `[`.

### Usage

```
slab_1(..., drop = FALSE)
```

### Arguments

...	arguments as they would be passed to <code>[</code> for example <code>x[3, 5, 7]</code> .
drop	whether or not to drop dimensions with length 1. Only applies to <code>view</code> .

### Examples

```
(x <- matrix(1:4, ncol = 2))
view(x, slab_1(2,)) # x[2,, drop = FALSE]
view(x, slab_1(2, 2)) # x[2,2, drop = FALSE]
set(x, slab_1(1,1:2), c(10,20))
```

---

slice_l	<i>Slice lens</i>
---------	-------------------

---

**Description**

Create a lens into a specific slice of a specific dimension of a multidimensional object. Not to be confused with `dplyr` slice.

**Usage**

```
slice_l(dimension, slice, drop = FALSE)
```

**Arguments**

dimension	the dimension to slice
slice	the slice index
drop	whether or not to drop dimensions with length 1. Only applies to <a href="#">view</a> .

**Examples**

```
(x <- matrix(1:4, ncol = 2))
view(x, slice_l(1, 2)) # x[,2, drop = FALSE]
view(x, slice_l(2, 2)) # x[,2, drop = FALSE]
set(x, slice_l(1,1), c(10,20))
```

---

slot_l	<i>Slot lens</i>
--------	------------------

---

**Description**

The lens equivalent of `@` and `@<-` for getting and setting S4 object slots.

**Usage**

```
slot_l(slot)
```

**Arguments**

slot	the name of the slot
------	----------------------

**Examples**

```
new_class <- setClass("new_class", slots = c(x = "numeric"))
(x <- new_class())

view(x, slot_l("x"))
set(x, slot_l("x"), 1:10)
```

---

take_l	<i>Construct a lens into a prefix of a vector</i>
--------	---

---

**Description**

This constructs a lens into the first  $n$  elements of an object or the if negative indexing is used, as many as  $\text{length}(x) - n$ .

**Usage**

```
take_l(n)
```

**Arguments**

n	number of elements to take, or if negative the number of elements at the end to not take.
---	---

**Examples**

```
x <- 1:10
view(x, take_l(3))
view(x, take_l(-7))
set(x, take_l(2), c(100,200))
set(x, take_l(-8), c(100,200))
```

---

take_while_il	<i>Conditional head lens</i>
---------------	------------------------------

---

**Description**

A lens into the elements from the beginning of a structure until the last element that satisfies a predicate.

**Usage**

```
take_while_il(f)
```

**Arguments**

f	the predicate (logical) function
---	----------------------------------

**Details**

This lens is illegal because `set-view` is not satisfied, multiple runs of the same lens will reference potentially different elements.

---

`to_l`*Promote a function to a getter lens*

---

**Description**

Create a getter lens from a function.

**Usage**

```
to_l(f)
```

**Arguments**

`f` The function to promote.

**Examples**

```
# This wouldn't make sense as a general legal lens, but fine as a `getter`
sqrt_l <- to_l(sqrt)
iris_root <- index(1) %>% index(1) %>% sqrt_l

sqrt(iris[[1]][[1]])
iris %>% view(iris_root)
tryCatch(iris %>% set(iris_root, 2)
  , error = function(e) "See, can't do that")
```

---

`transpose_l`*Lens into a list of rows*

---

**Description**

A lens that creates a list-of-rows view of a data.frame

**Usage**

```
transpose_l
```

**Format**

An object of class lens of length 2.

---

`t_1`                      *Matrix transpose lens*

---

**Description**

Lens into the transpose of a matrix

**Usage**

```
t_1
```

**Format**

An object of class `lens` of length 2.

**Examples**

```
(x <- matrix(1:4, ncol = 2))
view(x, t_1)
set(x, t_1, matrix(11:14, ncol = 2))
```

---

`unlist_1`                      *Unlist lens*

---

**Description**

A lens between a list and an unrecursively [unlisted](#) object.

**Usage**

```
unlist_1
```

**Format**

An object of class `lens` of length 2.

**Examples**

```
(x <- list(x = list(y = 1:10)))
view(x, unlist_1)
set(x, unlist_1 %.% unlist_1, rep("hello", 10))
```

---

upper_tri_l	<i>Lens into upper diagonal elements</i>
-------------	--

---

**Description**

Create a lens into the upper diagonal elements of a matrix

**Usage**

```
upper_tri_l(diag = FALSE)
```

**Arguments**

diag	whether or not to include the diagonal ( <code>x &lt;- matrix(1:9, ncol = 3)</code> ) <code>view(x, upper_tri_l())</code> <code>view(x, upper_tri_l(diag = TRUE))</code> <code>set(x, upper_tri_l(), c(100, 200, 300))</code>
------	---

---

view	<i>View data with a lens</i>
------	------------------------------

---

**Description**

Get the subcomponent of the data referred to by a lens. This function merely dispatches to the `view` component of the lens.

**Usage**

```
view(d, l)
```

**Arguments**

d	the data
l	the lens

%.%

---

*Compose lenses*

---

**Description**

Compose two lenses to produce a new lens which represents focussing first with the first lens, then with the second. A view using the resulting composite lens will first view using the first, then the second, while a set will view via the first lens, set into the resulting piece with the second, and then replace the updated structure in the first with set. Lens composition is analogous to the `.` syntax of object-oriented programming or to a flipped version of function composition.

**Usage**

```
l %>% m
```

**Arguments**

<code>l</code>	the first <a href="#">lens</a> (or an <a href="#">oscope</a> )
<code>m</code>	the second lens

**Examples**

```
lst <- list(b = c(3,4,5))
lns <- index_l("b") %>% index_l(2)
lst %>% view(lns)           # returns 4
lst %>% set(lns, 1)        # returns list(b = c(3,2,5))
lst                        # returns list(b = c(3,4,5))
```

# Index

## \* datasets

attributes\_l, 3  
body\_l, 4  
class\_l, 4  
colnames\_l, 5  
diag\_l, 7  
dim\_l, 8  
dimnames\_l, 7  
env\_l, 9  
first\_l, 10  
formals\_l, 11  
id\_l, 11  
last\_l, 13  
levels\_l, 14  
names\_l, 16  
rev\_l, 19  
rownames\_l, 19  
t\_l, 26  
transpose\_l, 25  
unlist\_l, 26

%.%, 28

attr\_l, 3  
attributes, 3  
attributes<-, 3  
attributes\_l, 3

body, 4  
body<-, 4  
body\_l, 4, 11

c\_l, 6  
class, 4  
class<-, 4  
class\_l, 4  
colnames\_l, 5  
cols\_l, 5  
cond\_il, 6

diag\_l, 7

dim\_l, 8  
dimnames, 7  
dimnames<-, 7  
dimnames\_l, 7  
dplyr::filter, 9  
dplyr::mutate, 18  
dplyr::select, 20  
drop\_while\_il, 8

env\_l, 9  
environment, 9  
environment<-, 9

filter\_il, 9  
filter\_l, 10  
first\_l, 10  
formals, 11  
formals<-, 11  
formals\_l, 11

id\_l, 11  
index (index\_l), 12  
index\_l, 6, 12  
indexes (indexes\_l), 12  
indexes\_l, 6, 12

lapply, 15  
last\_l, 13  
lens, 13, 22, 28  
levels, 14  
levels<-, 14  
levels\_l, 14  
lower\_tri\_l, 15

map\_l, 15  
mapply, 15

names\_l, 16

oscope, 16, 17, 22, 28  
over, 17

over\_map, 17  
over\_with, 18

purrr::pluck, 6

reshape\_l, 18  
rev, 19  
rev\_l, 19  
rownames\_l, 19  
rows\_l, 20

select\_l, 20  
send, 21  
send\_over, 21  
set, 6, 15, 18, 20, 22  
slab\_l, 22  
slice\_l, 23  
slot\_l, 23

t\_l, 26  
take\_l, 24  
take\_while\_il, 8, 24  
tidyselect::vars\_select, 20  
to\_l, 25  
transpose\_l, 25

unlist, 26  
unlist\_l, 26  
upper\_tri\_l, 27

view, 6, 15, 18, 23, 27