

# Package ‘loon’

October 16, 2019

**Type** Package

**Title** Interactive Statistical Data Visualization

**Version** 1.2.3

**Date** 2019-09-10

**URL** <http://great-northern-diver.github.io/loon/>

**Description** An extendable toolkit for interactive data visualization and exploration.

**License** GPL-2

**Depends** R (>= 3.4.0), methods, tcltk

**Imports** tools, graphics, grDevices, utils, stats, gridExtra

**Suggests** maps, sp, graph, scagnostics, PairViz, RColorBrewer,  
loon.data, rworldmap, rgl, Rgraphviz, RDRToolbox, kernlab,  
scales, MASS, testthat, knitr, rmarkdown, png, ElemStatLearn,  
formatR

**BugReports** <https://github.com/great-northern-diver/loon/issues>

**Encoding** UTF-8

**LazyData** true

**RoxygenNote** 6.1.1

**VignetteBuilder** knitr

**NeedsCompilation** no

**Author** Adrian Waddell [aut],  
R. Wayne Oldford [aut, cre, ths],  
Zehao Xu [ctb],  
Martin Gauch [ctb]

**Maintainer** R. Wayne Oldford <rwoldford@uwaterloo.ca>

**Repository** CRAN

**Date/Publication** 2019-10-16 04:30:02 UTC

**R topics documented:**

as.graph	7
as.loongraph	8
color_loon	9
complement	10
complement.loongraph	11
completegraph	11
condGrob	12
graphreduce	13
grid.loon	14
hex12tohex6	15
L2_distance	15
linegraph	16
linegraph.loongraph	17
loon	18
loongraph	19
loonGrob	20
loonGrob_layoutType	24
loon_palette	25
l_after_idle	26
l_aspect	26
l_aspect<-	27
l_bind_canvas	28
l_bind_canvas_delete	29
l_bind_canvas_get	29
l_bind_canvas_ids	30
l_bind_canvas_reorder	31
l_bind_context	32
l_bind_context_delete	33
l_bind_context_get	33
l_bind_context_ids	34
l_bind_context_reorder	35
l_bind_glyph	35
l_bind_glyph_delete	36
l_bind_glyph_get	37
l_bind_glyph_ids	37
l_bind_glyph_reorder	38
l_bind_item	39
l_bind_item_delete	40
l_bind_item_get	40
l_bind_item_ids	41
l_bind_item_reorder	42
l_bind_layer	42
l_bind_layer_delete	43
l_bind_layer_get	44
l_bind_layer_ids	44
l_bind_layer_reorder	45

<code>l_bind_navigator</code>	46
<code>l_bind_navigator_delete</code>	46
<code>l_bind_navigator_get</code>	47
<code>l_bind_navigator_ids</code>	48
<code>l_bind_navigator_reorder</code>	48
<code>l_bind_state</code>	49
<code>l_bind_state_delete</code>	50
<code>l_bind_state_get</code>	50
<code>l_bind_state_ids</code>	51
<code>l_bind_state_reorder</code>	52
<code>l_cget</code>	52
<code>l_colRemoveAlpha</code>	53
<code>l_configure</code>	54
<code>l_context_add_context2d</code>	54
<code>l_context_add_geodesic2d</code>	55
<code>l_context_add_slicing2d</code>	56
<code>l_context_delete</code>	57
<code>l_context_getLabel</code>	57
<code>l_context_ids</code>	58
<code>l_context_relabel</code>	58
<code>l_copyStates</code>	59
<code>l_createCompoundGrob</code>	61
<code>l_create_handle</code>	62
<code>l_currentindex</code>	63
<code>l_currenttags</code>	64
<code>l_data</code>	65
<code>l_export</code>	65
<code>l_export_valid_formats</code>	66
<code>l_getColorList</code>	66
<code>l_getGraph</code>	67
<code>l_getLinkedStates</code>	67
<code>l_getLocations</code>	68
<code>l_getOption</code>	69
<code>l_getOptionNames</code>	70
<code>l_getPlots</code>	70
<code>l_get_arrangeGrobArgs</code>	71
<code>l_glyphs_inspector</code>	71
<code>l_glyphs_inspector_image</code>	72
<code>l_glyphs_inspector_pointrange</code>	73
<code>l_glyphs_inspector_serialaxes</code>	73
<code>l_glyphs_inspector_text</code>	74
<code>l_glyph_add</code>	75
<code>l_glyph_add.default</code>	76
<code>l_glyph_add_image</code>	77
<code>l_glyph_add_pointrange</code>	78
<code>l_glyph_add_polygon</code>	79
<code>l_glyph_add_serialaxes</code>	80
<code>l_glyph_add_text</code>	81

<code>l_glyph_delete</code>	82
<code>l_glyph_getLabel</code>	83
<code>l_glyph_getType</code>	83
<code>l_glyph_ids</code>	84
<code>l_glyph_relabel</code>	84
<code>l_graph</code>	85
<code>l_graph.default</code>	85
<code>l_graph.graph</code>	86
<code>l_graph.loongraph</code>	87
<code>l_graphswitch</code>	87
<code>l_graphswitch_add</code>	88
<code>l_graphswitch_add.default</code>	89
<code>l_graphswitch_add.graph</code>	90
<code>l_graphswitch_add.loongraph</code>	90
<code>l_graphswitch_delete</code>	91
<code>l_graphswitch_get</code>	92
<code>l_graphswitch_getLabel</code>	92
<code>l_graphswitch_ids</code>	93
<code>l_graphswitch_move</code>	93
<code>l_graphswitch_relabel</code>	94
<code>l_graphswitch_reorder</code>	94
<code>l_graphswitch_set</code>	95
<code>l_graph_inspector</code>	95
<code>l_graph_inspector_analysis</code>	96
<code>l_graph_inspector_navigators</code>	96
<code>l_help</code>	97
<code>l_hexcolor</code>	98
<code>l_hist</code>	98
<code>l_hist_inspector</code>	100
<code>l_hist_inspector_analysis</code>	101
<code>l_imageviewer</code>	101
<code>l_image_import_array</code>	102
<code>l_image_import_files</code>	103
<code>l_info_states</code>	104
<code>l_isLoonWidget</code>	105
<code>l_layer</code>	105
<code>l_layer.density</code>	107
<code>l_layer.Line</code>	108
<code>l_layer.Lines</code>	109
<code>l_layer.map</code>	110
<code>l_layer.Polygon</code>	111
<code>l_layer.Polygons</code>	112
<code>l_layer.SpatialLines</code>	113
<code>l_layer.SpatialLinesDataFrame</code>	114
<code>l_layer.SpatialPoints</code>	115
<code>l_layer.SpatialPointsDataFrame</code>	116
<code>l_layer.SpatialPolygons</code>	117
<code>l_layer.SpatialPolygonsDataFrame</code>	118

<code>l_layers_inspector</code>	119
<code>l_layer_bbox</code>	120
<code>l_layer_contourLines</code>	121
<code>l_layer_delete</code>	122
<code>l_layer_demote</code>	123
<code>l_layer_expunge</code>	124
<code>l_layer_getChildren</code>	124
<code>l_layer_getLabel</code>	125
<code>l_layer_getParent</code>	126
<code>l_layer_getType</code>	127
<code>l_layer_group</code>	128
<code>l_layer_groupVisibility</code>	129
<code>l_layer_heatImage</code>	130
<code>l_layer_hide</code>	131
<code>l_layer_ids</code>	132
<code>l_layer_index</code>	134
<code>l_layer_isVisible</code>	134
<code>l_layer_layerVisibility</code>	135
<code>l_layer_line</code>	136
<code>l_layer_lines</code>	137
<code>l_layer_lower</code>	138
<code>l_layer_move</code>	139
<code>l_layer_oval</code>	140
<code>l_layer_points</code>	141
<code>l_layer_polygon</code>	142
<code>l_layer_polygons</code>	143
<code>l_layer_printTree</code>	145
<code>l_layer_promote</code>	145
<code>l_layer_raise</code>	146
<code>l_layer_rasterImage</code>	147
<code>l_layer_rectangle</code>	148
<code>l_layer_rectangles</code>	149
<code>l_layer_relabel</code>	151
<code>l_layer_show</code>	152
<code>l_layer_text</code>	153
<code>l_layer_texts</code>	154
<code>l_loon_inspector</code>	155
<code>l_make_glyphs</code>	155
<code>l_move_grid</code>	157
<code>l_move_halign</code>	158
<code>l_move_hdist</code>	159
<code>l_move_jitter</code>	160
<code>l_move_reset</code>	161
<code>l_move_valign</code>	162
<code>l_move_vdist</code>	163
<code>l_navgraph</code>	164
<code>l_navigator_add</code>	164
<code>l_navigator_delete</code>	165

<code>l_navigator_getLabel</code>	166
<code>l_navigator_getPath</code>	166
<code>l_navigator_ids</code>	167
<code>l_navigator_relabel</code>	167
<code>l_navigator_walk_backward</code>	168
<code>l_navigator_walk_forward</code>	168
<code>l_navigator_walk_path</code>	169
<code>l_nestedTclList2Rlist</code>	169
<code>l_ng_plots</code>	170
<code>l_ng_plots.default</code>	171
<code>l_ng_plots.measures</code>	172
<code>l_ng_plots.scagnostics</code>	174
<code>l_ng_ranges</code>	175
<code>l_ng_ranges.default</code>	175
<code>l_ng_ranges.measures</code>	176
<code>l_ng_ranges.scagnostics</code>	178
<code>l_pairs</code>	179
<code>l_plot</code>	180
<code>l_plot.decomposed.ts</code>	181
<code>l_plot.default</code>	183
<code>l_plot.density</code>	185
<code>l_plot.map</code>	186
<code>l_plot.stl</code>	187
<code>l_plot3D</code>	189
<code>l_plot3D.default</code>	191
<code>l_plot_arguments</code>	193
<code>l_plot_inspector</code>	195
<code>l_plot_inspector_analysis</code>	196
<code>l_plot_ts</code>	197
<code>l_primitiveGlyphs</code>	198
<code>l_redraw</code>	199
<code>l_resize</code>	199
<code>l_Rlist2nestedTclList</code>	200
<code>l_saveStatesRDS</code>	200
<code>l_scale3D</code>	202
<code>l_scaletto_active</code>	203
<code>l_scaletto_layer</code>	203
<code>l_scaletto_plot</code>	204
<code>l_scaletto_selected</code>	204
<code>l_scaletto_world</code>	204
<code>l_serialaxes</code>	205
<code>l_serialaxes_inspector</code>	209
<code>l_setAspect</code>	209
<code>l_setColorList</code>	210
<code>l_setColorList_baseR</code>	212
<code>l_setColorList_ColorBrewer</code>	212
<code>l_setColorList_ggplot2</code>	213
<code>l_setColorList_hcl</code>	214

l_setColorList_loon . . . . .	214
l_setLinkedStates . . . . .	215
l_setOption . . . . .	216
l_setTitleFont . . . . .	216
l_size . . . . .	217
l_size<- . . . . .	218
l_state_names . . . . .	218
l_subwin . . . . .	219
l_throwErrorIfNotLoonWidget . . . . .	219
l_toR . . . . .	220
l_userOptionDefault . . . . .	220
l_userOptions . . . . .	221
l_web . . . . .	221
l_widget . . . . .	222
l_worldview . . . . .	223
l_zoom . . . . .	223
measures1d . . . . .	224
measures2d . . . . .	225
minority . . . . .	226
names.loon . . . . .	227
ndtransitiongraph . . . . .	227
olive . . . . .	228
oliveAcids . . . . .	229
oliveLocations . . . . .	230
plot.loon . . . . .	230
plot.loongraph . . . . .	231
print.l_layer . . . . .	232
print.measures1d . . . . .	232
print.measures2d . . . . .	233
scagnostics2d . . . . .	233
tkcolors . . . . .	234
UsAndThem . . . . .	235

**Index****237**

as.graph

*Convert a loongraph object to an object of class graph***Description**

Loon's native graph class is fairly basic. The graph package (on bioconductor) provides a more powerful alternative to create and work with graphs. Also, many other graph theoretic algorithms such as the complement function and some graph layout and visualization methods are implemented for the graph objects in the RBGL and Rgraphviz R packages. For more information on packages that are useful to work with graphs see the *gRaphical Models in R* CRAN Task View at <https://CRAN.R-project.org/view=gR>.

**Usage**

```
as.graph(loongraph)
```

**Arguments**

loongraph      object of class loongraph

**Details**

See <http://www.bioconductor.org/packages/release/bioc/html/graph.html> for more information about the graph R package.

**Value**

graph object of class loongraph

**Examples**

```
if (requireNamespace("graph", quietly = TRUE)) {  
  g <- loongraph(letters[1:4], letters[1:3], letters[2:4], FALSE)  
  g1 <- as.graph(g)  
}
```

---

as.loongraph

*Convert a graph object to a loongraph object*

---

**Description**

Sometimes it is simpler to work with objects of class loongraph than to work with object of class graph.

**Usage**

```
as.loongraph(graph)
```

**Arguments**

graph      object of class graph (defined in the graph library)

**Details**

See <http://www.bioconductor.org/packages/release/bioc/html/graph.html> for more information about the graph R package.

For more information run: `l_help("learn_R_display_graph.html.html#graph-utilities")`

**Value**

graph object of class loongraph



## Examples

```
if (requireNamespace("graph", quietly = TRUE)) {  
  graph_graph = graph::randomEGraph(LETTERS[1:15], edges=100)  
  loon_graph <- as.loongraph(graph_graph)  
}
```

---

color\_loon

*Create a palette with loon's color mapping*

---

## Description

Used to map nominal data to colors. By default these colors are chosen so that the categories can be well differentiated visually (e.g. to highlight the different groups)

## Usage

```
color_loon()
```

## Details

This is the function that loon uses by default to map values to colors. Loon's mapping algorithm is as follows:

1. if all values already represent valid Tk colors (see [tkcolors](#)) then those colors are taken
2. if the number of distinct values is less than the number of values in loon's color mapping list then they get mapped according to the color list, see [l\\_setColorList](#) and [l\\_getColorList](#).
3. if there are more distinct values than there are colors in loon's color mapping list then loon's own color mapping algorithm is used. See [loon\\_palette](#) and the details section in the documentation of [l\\_setColorList](#).

For other mappings see the [col\\_numeric](#) and [col\\_factor](#) functions from the scales package.

## Value

A function that takes a vector with values and maps them to a vector of 6 digit hexadecimal encoded color representation (strings). Note that loon uses internally 12 digit hexadecimal encoded color values. If all the values that get passed to the function are valid color names in Tcl then those colors get returned hexencoded. Otherwise, if there is one or more elements that is not a valid color name it uses the loons default color mapping algorithm.

## See Also

[l\\_setColorList](#), [l\\_getColorList](#), [loon\\_palette](#)

**Examples**

```

pal <- color_loon()
pal(letters[1:4])
pal(c('a','a','b','c'))
pal(c('green', 'yellow'))

# show color choices for different n's
if (requireNamespace("grid", quietly = TRUE)) {
  grid::grid.newpage()
  grid::pushViewport(grid::plotViewport())
  grid::grid.rect()
  n <- 2^(1:5)
  grid::pushViewport(grid::dataViewport(xscale=c(0, max(n)+1),
                                         yscale=c(0, length(n)+1)))
  grid::grid.yaxis(at=c(1:length(n)), label=paste("n =", n))
  for (i in rev(seq_along(n))) {
    cols <- pal(1:n[i])
    grid::grid.points(x = 1:n[i], y = rep(i, n[i]),
                      default.units = "native", pch=15,
                      gp=grid::gpar(col=cols))
  }
  grid::grid.text("note the first i colors are shared for each n",
                  y = grid::unit(1,"npc") + grid::unit(1, "line"))
}

```

---

complement

*Create the Complement Graph of a Graph*


---

**Description**

Creates a complement graph of a graph

**Usage**

```
complement(x)
```

**Arguments**

x                    graph or loongraph object

**Value**

graph object

---

complement.loongraph    *Create the Complement Graph of a loon Graph*

---

**Description**

Creates a complement graph of a graph

**Usage**

```
## S3 method for class 'loongraph'
complement(x)
```

**Arguments**

x                    loongraph object

**Details**

This method is currently only implemented for undirected graphs.

**Value**

graph object of class loongraph

---

completograph            *Create a complete graph or digraph with a set of nodes*

---

**Description**

From Wikipedia: "a complete graph is a simple undirected graph in which every pair of distinct vertices is connected by a unique edge. A complete digraph is a directed graph in which every pair of distinct vertices is connected by a pair of unique edges (one in each direction)

**Usage**

```
completograph(nodes, isDirected = FALSE)
```

**Arguments**

nodes                    a character vector with node names, each element defines a node hence the elements need to be unique

isDirected                a boolean scalar to indicate wheter the returned object is a complete graph (undirected) or a complete digraph (directed).

**Details**

Note that this function masks the `completegraph` function of the `graph` package. Hence it is a good idea to specify the package namespace with `::`, i.e. `loon::completegraph` and `graph::completegraph`. For more information run: `l_help("learn_R_display_graph.html.html#graph-utilities")`

**Value**

graph object of class `loongraph`

**Examples**

```
g <- loon::completegraph(letters[1:5])
```

---

condGrob

*Create a named grob or a template grob depending on a test*

---

**Description**

Creates and returns a grid object using the function given by `'grobFun'` when `'test'` is `'TRUE'`. Otherwise a simple `'grob()'` is produced with the same parameters. All grob parameters are given in `'...'`.

**Usage**

```
condGrob(test = TRUE, grobFun = grob, name = "grob name", ...)
```

**Arguments**

<code>test</code>	Either <code>'TRUE'</code> or <code>'FALSE'</code> to indicate whether <code>'grobFun'</code> is to be used (default <code>'TRUE'</code> ) or not.
<code>grobFun</code>	The function to be used to create the grob when <code>'test = TRUE'</code> (e.g. <code>'textGrob'</code> , <code>'polygonGrob'</code> , etc.).
<code>name</code>	The name to be used for the returned grob.
<code>...</code>	The arguments to be given to the <code>'grobFun'</code> (or to <code>'grob()'</code> when <code>'test = FALSE'</code> ).

**Value**

A grob as produced by either the `'grobFun'` given or by `'grob()'` using the remaining arguments. If `'test = FALSE'` then the name is suffixed by `": 'grobFun name' arguments"`.

**Examples**

```
myGrob <- condGrob(test = (runif(1) > 0.5),
                  grobFun = textGrob,
                  name = "my label",
                  label = "Some random text")
myGrob
```

---

graphreduce	<i>Make each space in a node appear only once</i>
-------------	---

---

### Description

Reduce a graph to have unique node names

### Usage

```
graphreduce(graph, separator)
```

### Arguments

graph	graph of class loongraph
separator	one character that separates the spaces in node names

### Details

Note this is a string based operation. Node names must not contain the separator character!

### Value

graph object of class loongraph

### Examples

```
G <- completegraph(nodes=LETTERS[1:4])
LG <- linegraph(G)

LLG <- linegraph(LG)

graphreduce(LLG)

if (requireNamespace("Rgraphviz", quietly = TRUE)) {
  plot(graphreduce(LLG))
}
```

---

`grid.loon`*Create and optionally draw a grid grob from a loon widget handle*

---

**Description**

Create and optionally draw a grid grob from a loon widget handle

**Usage**

```
grid.loon(target, name = NULL, gp = gpar(), draw = TRUE, vp = NULL)
```

**Arguments**

<code>target</code>	either an object of class <code>loon</code> or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. <code>'.l0.plot'</code> ), the remaining objects by their ids.
<code>name</code>	a character identifier for the grob, or <code>NULL</code> . Used to find the grob on the display list and/or as a child of another grob.
<code>gp</code>	a <code>gpar</code> object, or <code>NULL</code> , typically the output from a call to the function <code>gpar</code> . This is basically a list of graphical parameter settings.
<code>draw</code>	a logical value indicating whether graphics output should be produced.
<code>vp</code>	a grid viewport object (or <code>NULL</code> ).

**Value**

a grid grob of the loon plot

**See Also**

[loonGrob](#), [plot.loon](#)

**Examples**

```
library(grid)
widget <- with(iris, l_plot(Sepal.Length, Sepal.Width))
grid.loon(widget)
```

---

hex12tohex6	<i>Convert 12 hexadecimal digit color representations to 6 hexadecimal digit color representations</i>
-------------	--

---

### Description

Tk colors must be in 6 hexadecimal format with two hexadecimal digits for each of the red, green, and blue components. Twelve hexadecimal digit colors have 4 hexadecimal digits for each. This function converts the 12 digit format to the 6 provided the color is preserved.

### Usage

```
hex12tohex6(x)
```

### Arguments

x                    a vector with 12 digit hexcolors

### Details

Function throws a warning if the conversion loses information. The [l\\_hexcolor](#) function converts any Tcl color specification to a 12 digit hexadecimal color representation.

### Examples

```
x <- l_hexcolor(c("red", "green", "blue", "orange"))
x
hex12tohex6(x)
```

---

L2_distance	<i>Euclidean distance between two vectors, or between column vectors of two matrices.</i>
-------------	---

---

### Description

Quickly calculates and returns the Euclidean distances between m vectors in one set and n vectors in another. Each set of vectors is given as the columns of a matrix.

### Usage

```
L2_distance(a, b, df = 0)
```

**Arguments**

a	A d by m numeric matrix giving the first set of m vectors of dimension d as the columns of a.
b	A d by n numeric matrix giving the second set of n vectors of dimension d as the columns of b.
df	Indicator whether to force the diagonals of the returned matrix to be zero (df = 1) or not (the default df = 0).

**Details**

This fully vectorized (VERY FAST!) function computes the Euclidean distance between two vectors by:

$$\|A-B\| = \sqrt{ \|A\|^2 + \|B\|^2 - 2*A.B }$$

Originally written as L2\_distance.m for Matlab by Roland Bunschoten of the University of Amsterdam, Netherlands.

**Value**

An m by n matrix containing the Euclidean distances between the column vectors of the matrix a and the column vectors of the matrix b.

**Author(s)**

Roland Bunschoten (original), Adrian Waddell, Wayne Oldford

**See Also**

[dist](#)

**Examples**

```
A <- matrix(rnorm(400), nrow = 10)
B <- matrix(rnorm(800), nrow = 10)
L2_distance(A[,1, drop = FALSE], B[,1, drop = FALSE])
d_AB <- L2_distance(A,B)
d_BB <- L2_distance(B,B, df = 1) # force diagonal to be zero
```

---

linegraph

*Create a linegraph*

---

**Description**

The line graph of G, here denoted L(G), is the graph whose nodes correspond to the edges of G and whose edges correspond to nodes of G such that nodes of L(G) are joined if and only if the corresponding edges of G are adjacent in G.



**Usage**

```
linegraph(x, ...)
```

**Arguments**

```
x          graph of class graph or loongraph  
...        arguments passed on to method
```

**Value**

```
graph object
```

---

```
linegraph.loongraph  Create a linegraph of a graph
```

---

**Description**

```
Create a linegraph of a loongraph
```

**Usage**

```
## S3 method for class 'loongraph'  
linegraph(x, separator = ":", ...)
```

**Arguments**

```
x          loongraph object  
separator  one character - node names in x get concatenated with this character  
...        additional arguments are not used for this method
```

**Details**

```
linegraph.loongraph needs the code part for directed graphs (i.e. isDirected=TRUE)
```

**Value**

```
graph object of class loongraph
```

**Examples**

```
g <- loongraph(letters[1:4], letters[1:3], letters[2:4], FALSE)  
linegraph(g)
```

## Description

Loon is a toolkit for highly interactive data visualization. Interactions with plots are provided with mouse and keyboard gestures as well as via command line control and with inspectors that provide graphical user interfaces (GUIs) for modifying and overseeing plots.

## Details

Currently, loon implements the following statistical graphs: histogram, scatterplot, serialaxes plot (star glyphs, parallel coordinates) and a graph display for creating navigation graphs.

Some of the implemented scatterplot features, for example, are zooming, panning, selection and moving of points, dynamic linking of plots, layering of visual information such as maps and regression lines, custom point glyphs (images, text, star glyphs), and event bindings. Event bindings provide hooks to evaluate custom code at specific plot state changes or mouse and keyboard interactions. Hence, event bindings can be used to add to or modify the default behavior of the plot widgets.

Loon's capabilities are very useful for statistical analysis tasks such as interactive exploratory data analysis, sensitivity analysis, animation, teaching, and creating new graphical user interfaces.

To get started using loon read the package vignettes or visit the loon website at <http://great-northern-diver.github.io/loon/>.

## Author(s)

**Maintainer:** R. Wayne Oldford <rwoldford@uwaterloo.ca> [thesis advisor]

Authors:

- Adrian Waddell <adrian@waddell.ch>

Other contributors:

- Zehao Xu <z267xu@uwaterloo.ca> [contributor]
- Martin Gauch <martin.gauch@student.kit.edu> [contributor]

## See Also

Useful links:

- <http://great-northern-diver.github.io/loon/>
- Report bugs at <https://github.com/great-northern-diver/loon/issues>

---

`loongraph`*Create a graph object of class loongraph*

---

### Description

The loongraph class provides a simple alternative to the graph class to create common graphs that are useful for use as navigation graphs.

### Usage

```
loongraph(nodes, from = character(0), to = character(0),
  isDirected = FALSE)
```

### Arguments

<code>nodes</code>	a character vector with node names, each element defines a node hence the elements need to be unique
<code>from</code>	a character vector with node names, each element defines an edge
<code>to</code>	a character vector with node names, each element defines an edge
<code>isDirected</code>	boolean scalar, defines whether from and to define directed edges

### Details

loongraph objects can be converted to graph objects (i.e. objects of class graph which is defined in the graph package) with the `as.graph` function.

For more information run: `l_help("learn_R_display_graph.html.html#graph-utilities")`

### Value

graph object of class loongraph

### See Also

[completegraph](#), [linegraph](#), [complement](#), [as.graph](#)

### Examples

```
g <- loongraph(
  nodes = c("A", "B", "C", "D"),
  from = c("A", "A", "B", "B", "C"),
  to = c("B", "C", "C", "D", "D")
)

## Not run:
# create a loon graph plot
p <- l_graph(g)
```

```
## End(Not run)

lg <- linegraph(g)
```

---

loonGrob

*Create a grid grob from a loon widget handle*

---

## Description

Grid grobs are useful to create publication quality graphics.

## Usage

```
loonGrob(target, name = NULL, gp = NULL, vp = NULL)
```

```
## S3 method for class 'l_compound'
loonGrob(target, name = NULL, gp = NULL,
  vp = NULL)
```

```
## S3 method for class 'l_layer_graph'
loonGrob(target, name = NULL, gp = NULL,
  vp = NULL)
```

```
## S3 method for class 'l_layer_histogram'
loonGrob(target, name = NULL, gp = NULL,
  vp = NULL)
```

```
## S3 method for class 'l_layer_scatterplot'
loonGrob(target, name = NULL, gp = NULL,
  vp = NULL)
```

```
## S3 method for class 'l_navgraph'
loonGrob(target, name = NULL, gp = NULL,
  vp = NULL)
```

```
## S3 method for class 'l_navigator'
loonGrob(target, name = NULL, gp = NULL,
  vp = NULL)
```

```
## S3 method for class 'l_serialaxes'
loonGrob(target, name = NULL, gp = NULL,
  vp = NULL)
```

```
## S3 method for class 'l_ts'
loonGrob(target, name = NULL, gp = NULL, vp = NULL)
```

**Arguments**

target	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. '.l0.plot'), the remaining objects by their ids.
name	a character identifier for the grob, or NULL. Used to find the grob on the display list and/or as a child of another grob.
gp	a gpar object, or NULL, typically the output from a call to the function gpar. This is basically a list of graphical parameter settings.
vp	a grid viewport object (or NULL).

**Value**

a grid grob

**See Also**

[grid.loon](#)

**Examples**

```
widget <- with(iris, l_plot(Sepal.Length, Sepal.Width))

lgrob <- loonGrob(widget)

library(grid)
grid.ls(lgrob, viewports=TRUE, fullNames=TRUE)
grid.newpage(); grid.draw(lgrob)

## Not run:
p <- demo("l_layers", ask = FALSE)$value

lgrob <- loonGrob(p)
grid.newpage(); grid.draw(lgrob)

p <- demo("l_glyph_sizes", ask = FALSE)$value

lgrob <- loonGrob(p)
grid.newpage()
grid.draw(lgrob)

## End(Not run)

## Not run:

library(grid)
## l_pairs (scatterplot matrix) examples

p <- l_pairs(iris[, -5], color=iris$Species)
```

```
lgrob <- loonGrob(p)
grid.newpage()
grid.draw(lgrob)

## Time series decomposition examples

decompose <- decompose(co2)
# or decompose <- stl(co2, "per")
p <- l_plot(decompose, title = "Atmospheric carbon dioxide over Mauna Loa")

# To print directly use either
plot(p)
# or
grid.loon(p)
# or to save structure
lgrob <- loonGrob(p)
grid.newpage()
grid.draw(lgrob)

## End(Not run)

## Not run:
## graph examples

G <- completegraph(names(iris[,-5]))
LG <- linegraph(G)
g <- l_graph(LG)

nav0 <- l_navigator_add(g)
l_configure(nav0, label = 0)
con0 <- l_context_add_geodesic2d(navigator=nav0, data=iris[,-5])

nav1 <- l_navigator_add(g, from = "Sepal.Length:Petal.Width",
  to = "Petal.Length:Petal.Width", proportion = 0.6)
l_configure(nav1, label = 1)
con1 <- l_context_add_geodesic2d(navigator=nav1, data=iris[,-5])

nav2 <- l_navigator_add(g, from = "Sepal.Length:Petal.Length",
  to = "Sepal.Width:Petal.Length", proportion = 0.5)
l_configure(nav2, label = 2)
con2 <- l_context_add_geodesic2d(navigator=nav2, data=iris[,-5])

# To print directly use either
plot(g)
# or
grid.loon(g)
# or to save structure
library(grid)
lgrob <- loonGrob(g)
grid.newpage(); grid.draw(lgrob)
```

```
## End(Not run)

## Not run:
## histogram examples

h <- l_hist(iris$Sepal.Length, color=iris$Species)

g <- loonGrob(h)

library(grid)
grid.newpage(); grid.draw(g)

h['showStackedColors'] <- TRUE

g <- loonGrob(h)

grid.newpage(); grid.draw(g)

h['colorStackingOrder'] <- c("selected", unique(h['color']))

g <- loonGrob(h)
grid.newpage(); grid.draw(g)

h['colorStackingOrder'] <- rev(h['colorStackingOrder'])

# To print directly use either
plot(h)
# or
grid.loon(h)

## End(Not run)

## l_plot scatterplot examples

p <- l_plot(x = c(0,1), y = c(0,1))
l_layer_rectangle(p, x = c(0,1), y = c(0,1))

g <- loonGrob(p)

library(grid)
grid.newpage(); grid.draw(g)

p['glyph'] <- "ctriangle"
p['color'] <- "blue"
p['size'] <- c(10, 20)
p['selected'] <- c(TRUE, FALSE)
g <- loonGrob(p)
grid.newpage(); grid.draw(g)
```

```
## Not run:
## navgraph examples

ng <- l_navgraph(oliveAcids, separator='-', color=olive$Area)

# To print directly use either
plot(ng)
# or
grid.loon(ng)
# or to save structure
lgrob <- loonGrob(ng)
library(grid)
grid.newpage()
grid.draw(lgrob)

## End(Not run)

## Serial axes (radial and parallel coordinate) examples

s <- l_serialaxes(data=oliveAcids, color=olive$Area, title="olive data")
sGrob_radial <- loonGrob(s)
library(grid)
grid.newpage(); grid.draw(sGrob_radial)
s['axesLayout'] <- 'parallel'
sGrob_parallel <- loonGrob(s)
grid.newpage(); grid.draw(sGrob_parallel)

## Not run:

## Time series decomposition examples

decompose <- decompose(co2)
# or decompose <- stl(co2, "per")
p <- l_plot(decompose, title = "Atmospheric carbon dioxide over Mauna Loa")

# To print directly use either
plot(p)
# or
grid.loon(p)
# or to save structure
lgrob <- loonGrob(p)
grid.newpage()
grid.draw(lgrob)

## End(Not run)
```

---



loonGrob\_layoutType    *A generic function used to distinguish whether only the locations of plots will be used to arrange them in a grob, or whether all arguments to 'gridExtra::arrangeGrob()' will be used.*

---

### Description

A generic function used to distinguish whether only the locations of plots will be used to arrange them in a grob, or whether all arguments to 'gridExtra::arrangeGrob()' will be used.

### Usage

```
loonGrob_layoutType(target)
```

### Arguments

target                    the (compound) loon plot to be laid out.

### Value

either the string "locations" (the default) or the string "arrangeGrobArgs". If "locations", then the generic function 'l\_getLocations()' will be called and only the location arguments of 'gridExtra::arrangeGrob()' used (i.e. a subset of 'c("ncol", "nrow", "layout\_matrix", "heights", "widths)'). The grobs to be laid out are constructed using the generic function 'l\_getPlots()'.

---

loon\_palette              *Loon's color generator for creating color palettes*

---

### Description

Loon has a color sequence generator implemented creates a color palettes where the first  $m$  colors of a color palette of size  $m+1$  are the same as the colors in a color palette of size  $m$ , for all positive natural numbers  $m$ . See the details in the [l\\_setColorList](#) documentation.

### Usage

```
loon_palette(n)
```

### Arguments

n                          numer of different colors in the palette

### Value

vector with hexencoded color values

**See Also**

[l\\_setColorList](#)

**Examples**

```
loon_palette(12)
```

---

l_after_idle	<i>Evaluate a function on once the processor is idle</i>
--------------	--

---

**Description**

It is possible for an observer to call the configure method of that plot while the plot is still in the configuration pipeline. In this case, a warning is thrown as unwanted side effects can happen if the next observer in line gets an outdated notification. In this case, it is recommended to use the `l_after_idle` function that evaluates some code once the processor is idle.

**Usage**

```
l_after_idle(fun)
```

**Arguments**

fun	function to be evaluated once tcl interpreter is idle
-----	---

---

l_aspect	<i>Query the aspect ratio of a plot</i>
----------	---

---

**Description**

The aspect ratio is defined by the ratio of the number of pixels for one data unit on the y axis and the number of pixels for one data unit on the x axes.

**Usage**

```
l_aspect(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

**Value**

aspect ratio

**Examples**

```
p <- with(iris, l_plot(Sepal.Length ~ Sepal.Width, color=Species))  
  
l_aspect(p)  
l_aspect(p) <- 1
```

---

`l_aspect<-`                    *Set the aspect ratio of a plot*

---

**Description**

The aspect ratio is defined by the ratio of the number of pixels for one data unit on the y axis and the number of pixels for one data unit on the x axes.

**Usage**

```
l_aspect(widget) <- value
```

**Arguments**

widget	widget path as a string or as an object handle
value	aspect ratio

**Details**

Changing the aspect ratio with `l_aspect<-` changes effectively the `zoomY` state to obtain the desired aspect ratio. Note that the aspect ratio in loon depends on the plot width, plot height and the states `zoomX`, `zoomY`, `deltaX`, `deltaY` and `swapAxes`. Hence, the aspect ratio can not be set permanently for a loon plot.

**Examples**

```
p <- with(iris, l_plot(Sepal.Length ~ Sepal.Width, color=Species))  
  
l_aspect(p)  
l_aspect(p) <- 1
```

---

l_bind_canvas	<i>Create a Canvas Binding</i>
---------------	--------------------------------

---

**Description**

Canvas bindings are triggered by a mouse/keyboard gesture over the plot as a whole.

**Usage**

```
l_bind_canvas(widget, event, callback)
```

**Arguments**

widget	widget path as a string or as an object handle
event	event patterns as defined for Tk canvas widget <a href="http://www.tcl.tk/man/tcl8.6/TkCmd/bind.htm#M5">http://www.tcl.tk/man/tcl8.6/TkCmd/bind.htm#M5</a> .
callback	callback function is an R function which is called by the Tcl interpreter if the event of interest happens. Note that in loon the callback functions support different optional arguments depending on the binding type, read the details for more information

**Details**

Canvas bindings are used to evaluate callbacks at certain X events on the canvas widget (underlying widget for all of loon's plot widgets). Such X events include re-sizing of the canvas and entering the canvas with the mouse.

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**Value**

canvas binding id

**See Also**

[l\\_bind\\_canvas\\_ids](#), [l\\_bind\\_canvas\\_get](#), [l\\_bind\\_canvas\\_delete](#), [l\\_bind\\_canvas\\_reorder](#)

**Examples**

```
# binding for when plot is resized
p <- l_plot(iris[,1:2], color=iris$Species)

printSize <- function(p) {
  size <- l_size(p)
  cat(paste('Size of widget ', p, ' is: ',
           size[1], 'x', size[2], ' pixels\n', sep=''))
}
```

```
l_bind_canvas(p, event='<Configure>', function(W) {printSize(W)})  
  
id <- l_bind_canvas_ids(p)  
id  
  
l_bind_canvas_get(p, id)
```

---

`l_bind_canvas_delete`    *Delete a canvas binding*

---

### Description

Remove a canvas binding

### Usage

```
l_bind_canvas_delete(widget, id)
```

### Arguments

widget	widget path as a string or as an object handle
id	canvas binding id

### Details

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

### See Also

[l\\_bind\\_canvas](#), [l\\_bind\\_canvas\\_ids](#), [l\\_bind\\_canvas\\_get](#), [l\\_bind\\_canvas\\_reorder](#)

---

`l_bind_canvas_get`    *Get the event pattern and callback Tcl code of a canvas binding*

---

### Description

This function returns the registered event pattern and the Tcl callback code that the Tcl interpreter evaluates after a event occurs that matches the event pattern.

### Usage

```
l_bind_canvas_get(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	canvas binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**Value**

Character vector of length two. First element is the event pattern, the second element is the Tcl callback code.

**See Also**

[l\\_bind\\_canvas](#), [l\\_bind\\_canvas\\_ids](#), [l\\_bind\\_canvas\\_delete](#), [l\\_bind\\_canvas\\_reorder](#)

**Examples**

```
# binding for when plot is resized
p <- l_plot(iris[,1:2], color=iris$Species)

printSize <- function(p) {
  size <- l_size(p)
  cat(paste('Size of widget ', p, ' is: ',
           size[1], 'x', size[2], ' pixels\n', sep=''))
}

l_bind_canvas(p, event='<Configure>', function(W) {printSize(W)})

id <- l_bind_canvas_ids(p)
id

l_bind_canvas_get(p, id)
```

---

`l_bind_canvas_ids`      *List canvas binding ids*

---

**Description**

List all user added canvas binding ids

**Usage**

```
l_bind_canvas_ids(widget)
```

**Arguments**

widget                    widget path as a string or as an object handle

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with canvas binding ids

**See Also**

[l\\_bind\\_canvas](#), [l\\_bind\\_canvas\\_get](#), [l\\_bind\\_canvas\\_delete](#), [l\\_bind\\_canvas\\_reorder](#)

**Examples**

```
# binding for when plot is resized
p <- l_plot(iris[,1:2], color=iris$Species)

printSize <- function(p) {
  size <- l_size(p)
  cat(paste('Size of widget ', p, ' is: ',
           size[1], 'x', size[2], ' pixels\n', sep=''))
}

l_bind_canvas(p, event='<Configure>', function(W) {printSize(W)})

id <- l_bind_canvas_ids(p)
id

l_bind_canvas_get(p, id)
```

---

`l_bind_canvas_reorder` *Reorder the canvas binding evaluation sequence*

---

**Description**

The order the canvas bindings defines how they get evaluated once an event matches event patterns of multiple canvas bindings.

**Usage**

```
l_bind_canvas_reorder(widget, ids)
```

**Arguments**

widget	widget path as a string or as an object handle
ids	new canvas binding id evaluation order, this must be a rearrangement of the elements returned by the <a href="#">l_bind_canvas_ids</a> function.

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with binding id evaluation order (same as the `id` argument)

**See Also**

[l\\_bind\\_canvas](#), [l\\_bind\\_canvas\\_ids](#), [l\\_bind\\_canvas\\_get](#), [l\\_bind\\_canvas\\_delete](#)

---

<code>l_bind_context</code>	<i>Add a context binding</i>
-----------------------------	------------------------------

---

**Description**

Creates a binding that evaluates a callback for particular changes in the collection of contexts of a display.

**Usage**

```
l_bind_context(widget, event, callback)
```

**Arguments**

widget	widget path as a string or as an object handle
event	a vector with one or more of the following evnets: 'add', 'delete', 'relabel'
callback	callback function is an R function which is called by the Tcl interpreter if the event of interest happens. Note that in loon the callback functions support different optional arguments depending on the binding type, read the details for more information

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

context binding id



**See Also**

[l\\_bind\\_context\\_ids](#), [l\\_bind\\_context\\_get](#), [l\\_bind\\_context\\_delete](#), [l\\_bind\\_context\\_reorder](#)

---

*l\_bind\_context\_delete*    *Delete a context binding*

---

**Description**

Remove a context binding

**Usage**

```
l_bind_context_delete(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	context binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**See Also**

[l\\_bind\\_context](#), [l\\_bind\\_context\\_ids](#), [l\\_bind\\_context\\_get](#), [l\\_bind\\_context\\_reorder](#)

---

*l\_bind\_context\_get*    *Get the event pattern and callback Tcl code of a context binding*

---

**Description**

This function returns the registered event pattern and the Tcl callback code that the Tcl interpreter evaluates after a event occurs that matches the event pattern.

**Usage**

```
l_bind_context_get(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	context binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

Character vector of length two. First element is the event pattern, the second element is the Tcl callback code.

**See Also**

[l\\_bind\\_context](#), [l\\_bind\\_context\\_ids](#), [l\\_bind\\_context\\_delete](#), [l\\_bind\\_context\\_reorder](#)

---

`l_bind_context_ids`      *List context binding ids*

---

**Description**

List all user added context binding ids

**Usage**

```
l_bind_context_ids(widget)
```

**Arguments**

`widget`                  widget path as a string or as an object handle

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with context binding ids

**See Also**

[l\\_bind\\_context](#), [l\\_bind\\_context\\_get](#), [l\\_bind\\_context\\_delete](#), [l\\_bind\\_context\\_reorder](#)

---

`l_bind_context_reorder`*Reorder the context binding evaluation sequence*

---

**Description**

The order the context bindings defines how they get evaluated once an event matches event patterns of multiple context bindings.

**Usage**

```
l_bind_context_reorder(widget, ids)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>ids</code>	new context binding id evaluation order, this must be a rearrangement of the elements returned by the <a href="#">l_bind_context_ids</a> function.

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**Value**

vector with binding id evaluation order (same as the `id` argument)

**See Also**

[l\\_bind\\_context](#), [l\\_bind\\_context\\_ids](#), [l\\_bind\\_context\\_get](#), [l\\_bind\\_context\\_delete](#)

---

`l_bind_glyph`*Add a glyph binding*

---

**Description**

Creates a binding that evaluates a callback for particular changes in the collection of glyphs of a display.

**Usage**

```
l_bind_glyph(widget, event, callback)
```

**Arguments**

widget	widget path as a string or as an object handle
event	a vector with one or more of the following evnets: 'add', 'delete', 'relabel'
callback	callback function is an R function which is called by the Tcl interpreter if the event of interest happens. Note that in loon the callback functions support different optional arguments depending on the binding type, read the details for more information

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**Value**

glyph binding id

**See Also**

[l\\_bind\\_glyph\\_ids](#), [l\\_bind\\_glyph\\_get](#), [l\\_bind\\_glyph\\_delete](#), [l\\_bind\\_glyph\\_reorder](#)

---

`l_bind_glyph_delete`    *Delete a glyph binding*

---

**Description**

Remove a glyph binding

**Usage**

```
l_bind_glyph_delete(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	glyph binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**See Also**

[l\\_bind\\_glyph](#), [l\\_bind\\_glyph\\_ids](#), [l\\_bind\\_glyph\\_get](#), [l\\_bind\\_glyph\\_reorder](#)

---

l_bind_glyph_get	<i>Get the event pattern and callback Tcl code of a glyph binding</i>
------------------	---

---

**Description**

This function returns the registered event pattern and the Tcl callback code that the Tcl interpreter evaluates after an event occurs that matches the event pattern.

**Usage**

```
l_bind_glyph_get(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	glyph binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**Value**

Character vector of length two. First element is the event pattern, the second element is the Tcl callback code.

**See Also**

[l\\_bind\\_glyph](#), [l\\_bind\\_glyph\\_ids](#), [l\\_bind\\_glyph\\_delete](#), [l\\_bind\\_glyph\\_reorder](#)

---

l_bind_glyph_ids	<i>List glyph binding ids</i>
------------------	-------------------------------

---

**Description**

List all user added glyph binding ids

**Usage**

```
l_bind_glyph_ids(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with glyph binding ids

**See Also**

[l\\_bind\\_glyph](#), [l\\_bind\\_glyph\\_get](#), [l\\_bind\\_glyph\\_delete](#), [l\\_bind\\_glyph\\_reorder](#)

---

`l_bind_glyph_reorder` *Reorder the glyph binding evaluation sequence*

---

**Description**

The order the glyph bindings defines how they get evaluated once an event matches event patterns of multiple glyph bindings.

**Usage**

```
l_bind_glyph_reorder(widget, ids)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>ids</code>	new glyph binding id evaluation order, this must be a rearrangement of the elements returned by the <a href="#">l_bind_glyph_ids</a> function.

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with binding id evaluation order (same as the `id` argument)

**See Also**

[l\\_bind\\_glyph](#), [l\\_bind\\_glyph\\_ids](#), [l\\_bind\\_glyph\\_get](#), [l\\_bind\\_glyph\\_delete](#)

---

`l_bind_item`*Create a Canvas Binding*

---

## Description

Canvas bindings are triggered by a mouse/keyboard gesture over the plot as a whole.

## Usage

```
l_bind_item(widget, tags, event, callback)
```

## Arguments

<code>widget</code>	widget path as a string or as an object handle
<code>tags</code>	item tags as explained in <code>l_help("learn_R_bind.html#item-bindings")</code>
<code>event</code>	event patterns as defined for Tk canvas widget <a href="http://www.tcl.tk/man/tcl8.6/TkCmd/bind.htm#M5">http://www.tcl.tk/man/tcl8.6/TkCmd/bind.htm#M5</a> .
<code>callback</code>	callback function is an R function which is called by the Tcl interpreter if the event of interest happens. Note that in loon the callback functions support different optional arguments depending on the binding type, read the details for more information

## Details

Item bindings are used for evaluating callbacks at certain mouse and/or keyboard gestures events (i.e. X events) on visual items on the canvas. Items on the canvas can have tags and item bindings are specified to be evaluated at certain X events for items with specific tags.

Note that item bindings get currently evaluated in the order that they are added.

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

## Value

item binding id

## See Also

[l\\_bind\\_item\\_ids](#), [l\\_bind\\_item\\_get](#), [l\\_bind\\_item\\_delete](#), [l\\_bind\\_item\\_reorder](#)

---

`l_bind_item_delete`      *Delete a item binding*

---

**Description**

Remove a item binding

**Usage**

```
l_bind_item_delete(widget, id)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>id</code>	item binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**See Also**

[l\\_bind\\_item](#), [l\\_bind\\_item\\_ids](#), [l\\_bind\\_item\\_get](#), [l\\_bind\\_item\\_reorder](#)

---

`l_bind_item_get`      *Get the event pattern and callback Tcl code of a item binding*

---

**Description**

This function returns the registered event pattern and the Tcl callback code that the Tcl interpreter evaluates after a event occurs that machtches the event pattern.

**Usage**

```
l_bind_item_get(widget, id)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>id</code>	item binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`



**Value**

Character vector of length two. First element is the event pattern, the second element is the Tcl callback code.

**See Also**

[l\\_bind\\_item](#), [l\\_bind\\_item\\_ids](#), [l\\_bind\\_item\\_delete](#), [l\\_bind\\_item\\_reorder](#)

---

<code>l_bind_item_ids</code>	<i>List item binding ids</i>
------------------------------	------------------------------

---

**Description**

List all user added item binding ids

**Usage**

```
l_bind_item_ids(widget)
```

**Arguments**

`widget`            widget path as a string or as an object handle

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with item binding ids

**See Also**

[l\\_bind\\_item](#), [l\\_bind\\_item\\_get](#), [l\\_bind\\_item\\_delete](#), [l\\_bind\\_item\\_reorder](#)

---

`l_bind_item_reorder`     *Reorder the item binding evaluation sequence*

---

### Description

The order the item bindings defines how they get evaluated once an event matches event patterns of multiple item bindings.

Reordering item bindings has currently no effect. Item bindings are evaluated in the order in which they have been added.

### Usage

```
l_bind_item_reorder(widget, ids)
```

### Arguments

<code>widget</code>	widget path as a string or as an object handle
<code>ids</code>	new item binding id evaluation order, this must be a rearrangement of the elements returned by the <code>l_bind_item_ids</code> function.

### Details

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

### Value

vector with binding id evaluation order (same as the `id` argument)

### See Also

[l\\_bind\\_item](#), [l\\_bind\\_item\\_ids](#), [l\\_bind\\_item\\_get](#), [l\\_bind\\_item\\_delete](#)

---

`l_bind_layer`     *Add a layer binding*

---

### Description

Creates a binding that evaluates a callback for particular changes in the collection of layers of a display.

### Usage

```
l_bind_layer(widget, event, callback)
```

**Arguments**

widget	widget path as a string or as an object handle
event	a vector with one or more of the following evnets: 'add', 'delete', 'move', 'hide', 'show', 'relabel'
callback	callback function is an R function which is called by the Tcl interpreter if the event of interest happens. Note that in loon the callback functions support different optional arguments depending on the binding type, read the details for more information

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**Value**

layer binding id

**See Also**

[l\\_bind\\_layer\\_ids](#), [l\\_bind\\_layer\\_get](#), [l\\_bind\\_layer\\_delete](#), [l\\_bind\\_layer\\_reorder](#)

---

`l_bind_layer_delete`     *Delete a layer binding*

---

**Description**

Remove a layer binding

**Usage**

```
l_bind_layer_delete(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	layer binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**See Also**

[l\\_bind\\_layer](#), [l\\_bind\\_layer\\_ids](#), [l\\_bind\\_layer\\_get](#), [l\\_bind\\_layer\\_reorder](#)

---

l_bind_layer_get	<i>Get the event pattern and callback Tcl code of a layer binding</i>
------------------	---

---

**Description**

This function returns the registered event pattern and the Tcl callback code that the Tcl interpreter evaluates after a event occurs that machtches the event pattern.

**Usage**

```
l_bind_layer_get(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	layer binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

**Value**

Character vector of length two. First element is the event pattern, the second element is the Tcl callback code.

**See Also**

[l\\_bind\\_layer](#), [l\\_bind\\_layer\\_ids](#), [l\\_bind\\_layer\\_delete](#), [l\\_bind\\_layer\\_reorder](#)

---

l_bind_layer_ids	<i>List layer binding ids</i>
------------------	-------------------------------

---

**Description**

List all user added layer binding ids

**Usage**

```
l_bind_layer_ids(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with layer binding ids

**See Also**

[l\\_bind\\_layer](#), [l\\_bind\\_layer\\_get](#), [l\\_bind\\_layer\\_delete](#), [l\\_bind\\_layer\\_reorder](#)

---

`l_bind_layer_reorder` *Reorder the layer binding evaluation sequence*

---

**Description**

The order the layer bindings defines how they get evaluated once an event matches event patterns of multiple layer bindings.

**Usage**

```
l_bind_layer_reorder(widget, ids)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>ids</code>	new layer binding id evaluation order, this must be a rearrangement of the elements returned by the <a href="#">l_bind_layer_ids</a> function.

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with binding id evaluation order (same as the `id` argument)

**See Also**

[l\\_bind\\_layer](#), [l\\_bind\\_layer\\_ids](#), [l\\_bind\\_layer\\_get](#), [l\\_bind\\_layer\\_delete](#)

---

<code>l_bind_navigator</code>	<i>Add a navigator binding</i>
-------------------------------	--------------------------------

---

**Description**

Creates a binding that evaluates a callback for particular changes in the collection of navigators of a display.

**Usage**

```
l_bind_navigator(widget, event, callback)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>event</code>	a vector with one or more of the following events: 'add', 'delete', 'relabel'
<code>callback</code>	callback function is an R function which is called by the Tcl interpreter if the event of interest happens. Note that in loon the callback functions support different optional arguments depending on the binding type, read the details for more information

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

navigator binding id

**See Also**

[l\\_bind\\_navigator\\_ids](#), [l\\_bind\\_navigator\\_get](#), [l\\_bind\\_navigator\\_delete](#), [l\\_bind\\_navigator\\_reorder](#)

---

<code>l_bind_navigator_delete</code>	<i>Delete a navigator binding</i>
--------------------------------------	-----------------------------------

---

**Description**

Remove a navigator binding

**Usage**

```
l_bind_navigator_delete(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	navigator binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**See Also**

[l\\_bind\\_navigator](#), [l\\_bind\\_navigator\\_ids](#), [l\\_bind\\_navigator\\_get](#), [l\\_bind\\_navigator\\_reorder](#)

---

`l_bind_navigator_get` *Get the event pattern and callback Tcl code of a navigator binding*

---

**Description**

This function returns the registered event pattern and the Tcl callback code that the Tcl interpreter evaluates after a event occurs that matches the event pattern.

**Usage**

```
l_bind_navigator_get(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	navigator binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

Character vector of length two. First element is the event pattern, the second element is the Tcl callback code.

**See Also**

[l\\_bind\\_navigator](#), [l\\_bind\\_navigator\\_ids](#), [l\\_bind\\_navigator\\_delete](#), [l\\_bind\\_navigator\\_reorder](#)

---

`l_bind_navigator_ids` *List navigator binding ids*

---

**Description**

List all user added navigator binding ids

**Usage**

`l_bind_navigator_ids(widget)`

**Arguments**

`widget` widget path as a string or as an object handle

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with navigator binding ids

**See Also**

[l\\_bind\\_navigator](#), [l\\_bind\\_navigator\\_get](#), [l\\_bind\\_navigator\\_delete](#), [l\\_bind\\_navigator\\_reorder](#)

---

`l_bind_navigator_reorder`

*Reorder the navigator binding evaluation sequence*

---

**Description**

The order the navigator bindings defines how they get evaluated once an event matches event patterns of multiple navigator bindings.

**Usage**

`l_bind_navigator_reorder(widget, ids)`

**Arguments**

`widget` widget path as a string or as an object handle

`ids` new navigator binding id evaluation order, this must be a rearrangement of the elements returned by the [l\\_bind\\_navigator\\_ids](#) function.



**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with binding id evaluation order (same as the `id` argument)

**See Also**

[l\\_bind\\_navigator](#), [l\\_bind\\_navigator\\_ids](#), [l\\_bind\\_navigator\\_get](#), [l\\_bind\\_navigator\\_delete](#)

---

l_bind_state	<i>Add a state change binding</i>
--------------	-----------------------------------

---

**Description**

The callback of a state change binding is evaluated when certain states change, as specified at binding creation.

**Usage**

```
l_bind_state(target, event, callback)
```

**Arguments**

target	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. <code>'.l0.plot'</code> ), the remaining objects by their ids.
event	vector with state names
callback	callback function is an R function which is called by the Tcl interpreter if the event of interest happens. Note that in loon the callback functions support different optional arguments depending on the binding type, read the details for more information

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

state change binding id

**See Also**

[l\\_info\\_states](#), [l\\_bind\\_state\\_ids](#), [l\\_bind\\_state\\_get](#), [l\\_bind\\_state\\_delete](#), [l\\_bind\\_state\\_reorder](#)

---

`l_bind_state_delete`     *Delete a state binding*

---

### Description

Remove a state binding

### Usage

```
l_bind_state_delete(target, id)
```

### Arguments

<code>target</code>	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. <code>'.l0.plot'</code> ), the remaining objects by their ids.
<code>id</code>	state binding id

### Details

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

### See Also

[l\\_bind\\_state](#), [l\\_bind\\_state\\_ids](#), [l\\_bind\\_state\\_get](#), [l\\_bind\\_state\\_reorder](#)

---

`l_bind_state_get`     *Get the event pattern and callback Tcl code of a state binding*

---

### Description

This function returns the registered event pattern and the Tcl callback code that the Tcl interpreter evaluates after a event occurs that matches the event pattern.

### Usage

```
l_bind_state_get(target, id)
```

### Arguments

<code>target</code>	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. <code>'.l0.plot'</code> ), the remaining objects by their ids.
<code>id</code>	state binding id

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

Character vector of length two. First element is the event pattern, the second element is the Tcl callback code.

**See Also**

[l\\_bind\\_state](#), [l\\_bind\\_state\\_ids](#), [l\\_bind\\_state\\_delete](#), [l\\_bind\\_state\\_reorder](#)

---

l_bind_state_ids	<i>List state binding ids</i>
------------------	-------------------------------

---

**Description**

List all user added state binding ids

**Usage**

```
l_bind_state_ids(target)
```

**Arguments**

target	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. <code>'l0.plot'</code> ), the remaining objects by their ids.
--------	--

**Details**

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation web-page, i.e. run `l_help("learn_R_bind")`

**Value**

vector with state binding ids

**See Also**

[l\\_bind\\_state](#), [l\\_bind\\_state\\_get](#), [l\\_bind\\_state\\_delete](#), [l\\_bind\\_state\\_reorder](#)

---

`l_bind_state_reorder` *Reorder the state binding evaluation sequence*

---

### Description

The order the state bindings defines how they get evaluated once an event matches event patterns of multiple state bindings.

### Usage

```
l_bind_state_reorder(target, ids)
```

### Arguments

<code>target</code>	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. <code>'l0.plot'</code> ), the remaining objects by their ids.
<code>ids</code>	new state binding id evaluation order, this must be a rearrangement of the elements returned by the <a href="#">l_bind_state_ids</a> function.

### Details

Bindings, callbacks, and binding substitutions are described in detail in loon's documentation webpage, i.e. run `l_help("learn_R_bind")`

### Value

vector with binding id evaluation order (same as the `ids` argument)

### See Also

[l\\_bind\\_state](#), [l\\_bind\\_state\\_ids](#), [l\\_bind\\_state\\_get](#), [l\\_bind\\_state\\_delete](#)

---

`l_cget` *Query a Plot State*

---

### Description

All of loon's displays have plot states. Plot states specify what is displayed, how it is displayed and if and how the plot is linked with other loon plots. Layers, glyphs, navigators and contexts have states too (also referred to as plot states). This function queries a single plot state.

### Usage

```
l_cget(target, state)
```

**Arguments**

target	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. '.l0.plot'), the remaining objects by their ids.
state	state name

**See Also**

[l\\_configure](#), [l\\_info\\_states](#), [l\\_create\\_handle](#)

**Examples**

```
p <- l_plot(iris, color = iris$Species)
l_cget(p, "color")
p['selected']
```

---

<code>l_colRemoveAlpha</code>	<i>Convert color representations having an alpha transparency level to 6 digit color representations</i>
-------------------------------	--

---

**Description**

Colors in the standard tk used by loon do not allow for alpha transparency. This function allows loon to use color palettes (e.g. [l\\_setColorList](#)) that produce colors with alpha transparency by simply using only the rgb.

**Usage**

```
l_colRemoveAlpha(col)
```

**Arguments**

col	a vector of colors (potentially) containing an alpha level
-----	--

**Examples**

```
x <- l_colRemoveAlpha(rainbow(6))
# Also works with ordinary color string representations
# since it just extracts the rgb values from the colors.
x <- l_colRemoveAlpha(c("red", "blue", "green", "orange"))
x
```

---

l_configure	<i>Modify one or multiple plot states</i>
-------------	---

---

### Description

All of loon's displays have plot states. Plot states specify what is displayed, how it is displayed and if and how the plot is linked with other loon plots. Layers, glyphs, navigators and contexts have states too (also referred to as plot states). This function modifies one or multiple plot states.

### Usage

```
l_configure(target, ...)
```

### Arguments

target	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. '.l0.plot'), the remaining objects by their ids.
...	state=value pairs

### See Also

[l\\_cget](#), [l\\_info\\_states](#), [l\\_create\\_handle](#)

### Examples

```
p <- l_plot(iris, color = iris$Species)
l_configure(p, color='red')
p['size'] <- ifelse(iris$Species == "versicolor", 2, 8)
```

---

l_context_add_context2d	
-------------------------	--

*Create a context2d navigator context*

---

### Description

A context2d maps every location on a 2d space graph to a list of xvars and a list of yvars such that, while moving the navigator along the graph, as few changes as possible take place in xvars and yvars.

Contexts are in more detail explained in the webmanual accessible with [l\\_help](#). Please read the section on context by running `l_help("learn_R_display_graph.html#contexts")`.

### Usage

```
l_context_add_context2d(navigator, ...)
```

**Arguments**

navigator      navigator handle object  
 ...            arguments passed on to modify context states

**Value**

context handle

**See Also**

[l\\_info\\_states](#), [l\\_context\\_ids](#), [l\\_context\\_add\\_geodesic2d](#), [l\\_context\\_add\\_slicing2d](#), [l\\_context\\_getLabel](#),  
[l\\_context\\_relabel](#)

---

l\_context\_add\_geodesic2d

*Create a geodesic2d navigator context*

---

**Description**

Geodesic2d maps every location on the graph as an orthogonal projection of the data onto a two-dimensional subspace. The nodes then represent the sub-space spanned by a pair of variates and the edges either a 3d- or 4d-transition of one scatterplot into another, depending on how many variates the two nodes connected by the edge share (see Hurley and Oldford 2011). The geodesic2d context inherits from the context2d context.

Contexts are in more detail explained in the webmanual accessible with [l\\_help](#). Please read the section on context by running `l_help("learn_R_display_graph.html#contexts")`.

**Usage**

```
l_context_add_geodesic2d(navigator, ...)
```

**Arguments**

navigator      navigator handle object  
 ...            arguments passed on to modify context states

**Value**

context handle

**See Also**

[l\\_info\\_states](#), [l\\_context\\_ids](#), [l\\_context\\_add\\_context2d](#), [l\\_context\\_add\\_slicing2d](#), [l\\_context\\_getLabel](#),  
[l\\_context\\_relabel](#)

---

`l_context_add_slicing2d`*Create a slicind2d navigator context*

---

### Description

The slicing2d context implements slicing using navigation graphs and a scatterplot to condition on one or two variables.

Contexts are in more detail explained in the webmanual accessible with [l\\_help](#). Please read the section on context by running `l_help("learn_R_display_graph.html#contexts")`.

### Usage

```
l_context_add_slicing2d(navigator, ...)
```

### Arguments

<code>navigator</code>	navigator handle object
<code>...</code>	arguments passed on to modify context states

### Value

context handle

### Examples

```
names(oliveAcids) <- c('p', 'p1', 's', 'o', 'l', 'l1', 'a', 'e')
nodes <- apply(combn(names(oliveAcids), 2), 2,
              function(x) paste(x, collapse=':'))
G <- completegraph(nodes)
g <- l_graph(G)
nav <- l_navigator_add(g)
con <- l_context_add_slicing2d(nav, data=oliveAcids)

# symmetric range proportion around nav['proportion']
con['proportion'] <- 0.2

con['conditioning4d'] <- "union"
con['conditioning4d'] <- "intersection"
```



---

l_context_delete	<i>Delete a context from a navigator</i>
------------------	--

---

**Description**

Navigators can have multiple contexts. This function removes a context from a navigator.

**Usage**

```
l_context_delete(navigator, id)
```

**Arguments**

navigator	navigator handle
id	context id

**Details**

For more information run: `l_help("learn_R_display_graph.html#contexts")`

**See Also**

[l\\_context\\_ids](#), [l\\_context\\_add\\_context2d](#), [l\\_context\\_add\\_geodesic2d](#), [l\\_context\\_add\\_slicing2d](#), [l\\_context\\_getLabel](#), [l\\_context\\_relabel](#)

---

l_context_getLabel	<i>Query the label of a context</i>
--------------------	-------------------------------------

---

**Description**

Context labels are eventually used in the context inspector. This function queries the label of a context.

**Usage**

```
l_context_getLabel(navigator, id)
```

**Arguments**

navigator	navigator handle
id	context id

**Details**

For more information run: `l_help("learn_R_display_graph.html#contexts")`

**See Also**

[l\\_context\\_getLabel](#), [l\\_context\\_add\\_context2d](#), [l\\_context\\_add\\_geodesic2d](#), [l\\_context\\_add\\_slicing2d](#), [l\\_context\\_delete](#)

---

<code>l_context_ids</code>	<i>List context ids of a navigator</i>
----------------------------	--

---

**Description**

Navigators can have multiple contexts. This function list the context ids of a navigator.

**Usage**

```
l_context_ids(navigator)
```

**Arguments**

<code>navigator</code>	navigator handle
------------------------	------------------

**Details**

For more information run: `l_help("learn_R_display_graph.html#contexts")`

**See Also**

[l\\_context\\_delete](#), [l\\_context\\_add\\_context2d](#), [l\\_context\\_add\\_geodesic2d](#), [l\\_context\\_add\\_slicing2d](#), [l\\_context\\_getLabel](#), [l\\_context\\_relabel](#)

---

<code>l_context_relabel</code>	<i>Change the label of a context</i>
--------------------------------	--------------------------------------

---

**Description**

Context labels are eventually used in the context inspector. This function relabels a context.

**Usage**

```
l_context_relabel(navigator, id, label)
```

**Arguments**

<code>navigator</code>	navigator handle
<code>id</code>	context id
<code>label</code>	context label shown

**Details**

For more information run: `l_help("learn_R_display_graph.html#contexts")`

**See Also**

[l\\_context\\_getLabel](#), [l\\_context\\_add\\_context2d](#), [l\\_context\\_add\\_geodesic2d](#), [l\\_context\\_add\\_slicing2d](#), [l\\_context\\_delete](#)

---

<code>l_copyStates</code>	<i>A generic function to transfer the values of the states of one ‘loon’ structure to another.</i>
---------------------------	--

---

**Description**

`l_copyStates` reads the values of the states of the ‘source’ and assigns them to the states of the same name on the ‘target’.

**Usage**

```
l_copyStates(source, target, states = NULL, exclude = NULL,
             excludeBasicStates = TRUE, returnNames = FALSE)
```

**Arguments**

<code>source</code>	the ‘loon’ object providing the values of the states.
<code>target</code>	the ‘loon’ object whose states are assigned the values of the ‘sources’ states of the same name.
<code>states</code>	a character vector of the states to be copied. If ‘NULL’ (the default), then all states in common (excluding those identified by exclusion parameters) are copied from the ‘source’ to the ‘target’.
<code>exclude</code>	a character vector naming those common states to be excluded from copying. Default is NULL.
<code>excludeBasicStates</code>	a logical indicating whether certain basic states are to be excluded from the copy (if ‘TRUE’, the default). These states include those derived from data variables (like "x", "xTemp", "zoomX", "panX", "deltaX", "xlabel", and the "y" counterparts) since these values determine coordinates in the plot and so are typically not to be copied. Similarly "swapAxes" is one of these basic states because in <code>l_compound</code> plots such as <code>l_pairs()</code> swapping axes can wreak havoc if unintended. Finally, an important pair of basic states to exclude are "linkingKey" and "linkingGroup" since such changes require proper synchronization. Setting ‘excludeBasicStates = TRUE’ is a simple way to avoid copying the values of these basic states. Setting ‘excludeBasicStates = FALSE’ will allow these to be copied as well.
<code>returnNames</code>	a logical to indicate whether to return the names of all states successfully copied for all plots. Default is ‘FALSE’

**Value**

a character vector of the names of the states successfully copied (for each plot whose states were affected), or NULL if none were copied or 'returnNames == FALSE'.

**See Also**

[l\\_saveStatesRDS](#) [l\\_info\\_states](#) [saveRDS](#)

**Examples**

```
if(interactive()){
# Source and target are `l_plots`
p <- with(iris,
  l_plot(x = Sepal.Width, y = Petal.Width,
        color = Species, glyph = "ccircle",
        size = 10, showGuides = TRUE,
        title = "Edgar Anderson's Iris data"
  )
)

p2 <- with(iris,
  l_plot(x = Sepal.Length, y = Petal.Length,
        title = "Fisher's Iris data"
  )
)

# Copy the states of p to p2
# First just the size and title
l_copyStates(source = p, target = p2,
  states = c("size", "title")
)

# Copy all but those associated with the variables
l_copyStates(source = p, target = p2)

# Suppose p had a linkingGroup, say "Edgar"
l_configure(p, linkingGroup = "Edgar", sync = "push")

# To force this linkingGroup to be copied to a new plot
p3 <- with(iris,
  l_plot(x = Sepal.Length, y = Petal.Length,
        title = "Fisher's Iris data"
  )
)

l_copyStates(source = p, target = p3,
  states = c("linkingGroup"),
  # To allow this to happen:
  excludeBasicStates = FALSE
)

h <- with(iris,
  l_hist((Petal.Width * Petal.Length),
        showStackedColors = TRUE,
        yshows = "density")
)
```

```

    )
  l_copyStates(source = p, target = h)

  sa <- l_serialaxes(iris, axes = "parallel")
  l_copyStates(p, sa)

  pp <- l_pairs(iris, showHistograms = TRUE)
  l_copyStates(p, pp) # makes no copy (not one to one)

  pp2 <- l_pairs(iris,
                color = iris$Species,
                showGuides = TRUE,
                title = "Iris data",
                glyph = "ctriangle")
  l_copyStates(pp2, pp)
  l_copyStates(pp2, p)
}

```

---

`l_createCompoundGrob` *For the target compound loon plot, creates the final grob from the class of the ‘target’ and the ‘arrangeGrob.args’*

---

### Description

For the target compound loon plot, creates the final grob from the class of the ‘target’ and the ‘arrangeGrob.args’

### Usage

```
l_createCompoundGrob(target, arrangeGrob.args)
```

### Arguments

`target` the (compound) loon plot  
`arrangeGrob.args` arguments as described by ‘gridExtra::arrangeGrob()’

### Value

a grob (or list of grobs) that can be handed to ‘gTree()’ as ‘children = gList(returnedValue)’ as the final grob constructed for the compound loon plot. Default for an ‘l\_compound’ is to simply execute ‘gridExtra::arrangeGrob(arrangeGrob.args)’.

---

l_create_handle	<i>Create a loon object handle</i>
-----------------	------------------------------------

---

### Description

This function can be used to create the loon object handles from a vector of the widget path name and the object ids (in the order of the parent-child relationships).

### Usage

```
l_create_handle(target)
```

### Arguments

target            loon object specification (e.g. ".l0.plot")

### Details

loon's plot handles are useful to query and modify plot states via the command line.

For more information run: `l_help("learn_R_intro.html#re-creating-object-handles")`

### Examples

```
# plot handle
p <- l_plot(x=1:3, y=1:3)
p_new <- l_create_handle(unclass(p))
p_new['showScales']

# glyph handle
gl <- l_glyph_add_text(p, text=LETTERS[1:3])
gl_new <- l_create_handle(c(as.vector(p), as.vector(gl)))
gl_new['text']

# layer handle
l <- l_layer_rectangle(p, x=c(1,3), y=c(1,3), color='yellow', index='end')
l_new <- l_create_handle(c(as.vector(p), as.vector(l)))
l_new['color']

# navigator handle
g <- l_graph(linegraph(completograph(LETTERS[1:3])))
nav <- l_navigator_add(g)
nav_new <- l_create_handle(c(as.vector(g), as.vector(nav)))
nav_new['from']

# context handle
con <- l_context_add_context2d(nav)
con_new <- l_create_handle(c(as.vector(g), as.vector(nav), as.vector(con)))
con_new['separator']
```

---

l_currentindex	<i>Get layer-relative index of the item below the mouse cursor</i>
----------------	--

---

### Description

Checks if there is a visual item below the mouse cursor and if there is, it returns the index of the visual item's position in the corresponding variable dimension of its layer.

### Usage

```
l_currentindex(widget)
```

### Arguments

widget            widget path as a string or as an object handle

### Details

For more details see `l_help("learn_R_bind.html#item-bindings")`

### Value

index of the visual item's position in the corresponding variable dimension of its layer

### See Also

[l\\_bind\\_item](#), [l\\_currenttags](#)

### Examples

```
p <- l_plot(iris[,1:2], color=iris$Species)

printEntered <- function(W) {
  cat(paste('Entered point ', l_currentindex(W), '\n'))
}

printLeave <- function(W) {
  cat(paste('Left point ', l_currentindex(W), '\n'))
}

l_bind_item(p, tags='model&&point', event='<Enter>',
  callback=function(W) {printEntered(W)})

l_bind_item(p, tags='model&&point', event='<Leave>',
  callback=function(W) {printLeave(W)})
```

---

l_currenttags	<i>Get tags of the item below the mouse cursor</i>
---------------	--

---

**Description**

Retrieves the tags of the visual item that at the time of the function evaluation is below the mouse cursor.

**Usage**

```
l_currenttags(widget)
```

**Arguments**

widget            widget path as a string or as an object handle

**Details**

For more details see `l_help("learn_R_bind.html#item-bindings")`

**Value**

vector with item tags of visual

**See Also**

[l\\_bind\\_item](#), [l\\_currentindex](#)

**Examples**

```
printTags <- function(W) {  
  print(l_currenttags(W))  
}  
  
p <- l_plot(x=1:3, y=1:3, title='Query Visual Item Tags')  
  
l_bind_item(p, 'all', '<ButtonPress>', function(W)printTags(W))
```



---

l_data	<i>Convert an R data.frame to a Tcl dictionary</i>
--------	--

---

**Description**

This is a helper function to convert an R data.frame object to a Tcl data frame object. This function is useful when changing a data state with [l\\_configure](#).

**Usage**

```
l_data(data)
```

**Arguments**

data	a data.frame object
------	---------------------

**Value**

a string that represents with data.frame with a Tcl dictionary data structure.

---

l_export	<i>Export a loon plot as an image</i>
----------	---------------------------------------

---

**Description**

The supported image formats are dependent on the system environment. Plots can always be exported to the PostScript format. Exporting displays as .pdfs is only possible when the command line tool epstopdf is installed. Finally, exporting to either png, jpg, bmp, tiff or gif requires the Img Tcl extension. When choosing one of the formats that depend on the Img extension, it is possible to export any Tk widget as an image including inspectors.

**Usage**

```
l_export(widget, filename, width, height)
```

**Arguments**

widget	widget path as a string or as an object handle
filename	path of output file
width	image width in pixels
height	image height in pixels

**Details**

Note that the CTRL-P key combination opens a dialog to export the graphic.

The native export format is to ps as this is what the Tk canvas offers. If the the `l_export` fails with other formats then please resort to a screen capture method for the moment.

**Value**

path to the exported file

**See Also**

[l\\_export\\_valid\\_formats](#), [plot.loon](#)

---

`l_export_valid_formats`

*Return a list of the available image formats when exporting a loon plot*

---

**Description**

The supported image formats are dependent on the system environment. Plots can always be exported to the Postscript format. Exporting displays as .pdfs is only possible when the command line tool `epstopdf` is installed. Finally, exporting to either png, jpg, bmp, tiff or gif requires the `Img Tcl` extension. When choosing one of the formats that depend on the `Img` extension, it is possible to export any Tk widget as an image including inspectors.

**Usage**

`l_export_valid_formats()`

**Value**

a vector with the image formats available for exporting a loon plot.

---

`l_getColorList`

*Get loon's color mapping list*

---

**Description**

The color mapping list is used by loon to convert nominal values to color values, see the documentation for [l\\_setColorList](#).

**Usage**

`l_getColorList()`

**Value**

a vector with hex-encoded colors

**See Also**

[l\\_setColorList](#)

---

l_getGraph	<i>Extract a loongraph or graph object from loon's graph display</i>
------------	--

---

**Description**

The graph display represents a graph with the nodes, from, to, and isDirected plot states. This function creates a loongraph or a graph object using these states.

**Usage**

```
l_getGraph(widget, asloongraph = TRUE)
```

**Arguments**

widget	a graph widget handle
asloongraph	boolean, if TRUE then the function returns a loongraph object, otherwise the function returns a graph object defined in the graph R package.

**Value**

a loongraph or a graph object

**See Also**

[l\\_graph](#), [loongraph](#)

---

l_getLinkedStates	<i>Query the States that are Linked with Loon's Standard Linking Model</i>
-------------------	--

---

**Description**

Loon's standard linking model is based on three levels, the linkingGroup and linkingKey states and the *used linkable states*. See the details in the documentation for [l\\_setLinkedStates](#).

**Usage**

```
l_getLinkedStates(widget)
```

**Arguments**

widget                    widget path as a string or as an object handle

**Value**

vector with state names that are linked states

**See Also**

[l\\_setLinkedStates](#)

---

l_getLocations	<i>For the target compound loon plot, determines location (only and excluding the grobs) arguments to pass to 'gridExtra::arrangeGrob()'</i>
----------------	--

---

**Description**

For the target compound loon plot, determines location (only and excluding the grobs) arguments to pass to 'gridExtra::arrangeGrob()'

**Usage**

```
l_getLocations(target)

## S3 method for class 'l_pairs'
l_getLocations(target)

## S3 method for class 'l_ts'
l_getLocations(target)
```

**Arguments**

target                    the (compound) loon plot whose locations are needed lay it out.

**Value**

a list of an appropriate subset of the named location arguments 'c("ncol", "nrow", "layout\_matrix", "heights", "widths)". There are as many heights and widths as there are plots returned by l\_getPlots(); these specify the relative height and width of each plot in the display. layout\_matrix is an nrow by ncol matrix whose entries identify the location of each plot in l\_getPlots() by their index.

**Examples**

```
if(interactive()) {  
  
  pp <- l_pairs(iris, showHistograms = TRUE)  
  ll <- l_getLocations(pp)  
  nplots <- length(l_getPlots(pp))  
  # the plots returned by l_getPlots(pp) are positioned  
  # in order by the layout_matrix  
  # ll$layout_matrix TODO  
}
```

---

**l\_getOption***Get the value of a loon display option*

---

**Description**

All of loon's displays access a set of common options. This function accesses and returns the current value of the named option.

**Usage**

```
l_getOption(option)
```

**Arguments**

**option**            the name of the option being queried.

**Value**

the value of the named option.

**See Also**

[l\\_getOptionNames](#), [l\\_userOptions](#), [l\\_userOptionDefault](#), [l\\_setOption](#)

**Examples**

```
l_getOption("background")
```

---

l_getOptionNames	<i>Get the names of all loon display options</i>
------------------	--

---

**Description**

All of loon's displays access a set of common options. This function accesses and returns the names of all loon options.

**Usage**

```
l_getOptionNames()
```

**Value**

a vector of all loon display option names.

**See Also**

[l\\_getOption](#), [l\\_userOptions](#), [l\\_userOptionDefault](#), [l\\_setOption](#)

**Examples**

```
l_getOptionNames()
```

---

l_getPlots	<i>For the target compound loon plot, determines all the loon plots in that compound plot.</i>
------------	--

---

**Description**

For the target compound loon plot, determines all the loon plots in that compound plot.

**Usage**

```
l_getPlots(target)

## S3 method for class 'l_pairs'
l_getPlots(target)

## S3 method for class 'l_ts'
l_getPlots(target)
```

**Arguments**

target            the (compound) loon plot to be laid out.

**Value**

a list of the named arguments and their values to be passed to 'gridExtra::arrangeGrob()'.

---

`l_get_arrangeGrobArgs` *For the target (compound) loon plot, determines all arguments (i.e. including the grobs) to be passed to 'gridExtra::arrangeGrob()' so as to determine the layout in 'grid' graphics.*

---

**Description**

For the target (compound) loon plot, determines all arguments (i.e. including the grobs) to be passed to 'gridExtra::arrangeGrob()' so as to determine the layout in 'grid' graphics.

**Usage**

```
l_get_arrangeGrobArgs(target)
```

**Arguments**

`target`            the (compound) loon plot to be laid out.

**Value**

a list of the named arguments and their values to be passed to 'gridExtra::arrangeGrob()'.

---

`l_glyphs_inspector`    *Create a Glyphs Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_glyphs_inspector(parent = NULL, ...)
```

**Arguments**

`parent`            parent widget path  
`...`              state arguments

**Value**

widget handle

**See Also**[l\\_create\\_handle](#)**Examples**

```
i <- l_glyphs_inspector()
```

---

`l_glyphs_inspector_image`*Create a Image Glyph Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_glyphs_inspector_image(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**[l\\_create\\_handle](#)**Examples**

```
i <- l_glyphs_inspector_image()
```



---

l\_glyphs\_inspector\_pointrange  
*Create a Pointrange Glyph Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_glyphs_inspector_pointrange(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_glyphs_inspector_pointrange()
```

---

l\_glyphs\_inspector\_serialaxes  
*Create a Serialaxes Glyph Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_glyphs_inspector_serialaxes(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_glyphs_inspector_serialaxes()
```

---

l\_glyphs\_inspector\_text

*Create a Text Glyph Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_glyphs_inspector_text(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_glyphs_inspector_text()
```

---

l\_glyph\_add

Add non-primitive glyphs to a scatterplot or graph display

---

### Description

Generic method for adding user-defined glyphs. See details for more information about non-primitive and primitive glyphs.

### Usage

```
l_glyph_add(widget, type, ...)
```

### Arguments

widget	widget path as a string or as an object handle
type	object used for method dispatch
...	arguments passed on to method

### Details

The scatterplot and graph displays both have the n-dimensional state 'glyph' that assigns each data point or graph node a glyph (i.e. a visual representation).

Loon distinguishes between primitive and non-primitive glyphs: the primitive glyphs are always available for use whereas the non-primitive glyphs need to be first specified and added to a plot before they can be used.

The primitive glyphs are:

```
'circle', 'ocircle', 'ccircle'
'square', 'osquare', 'csquare'
'triangle', 'otriangle', 'ctriangle'
'diamond', 'odiamond', 'cdiamond'
```

Note that the letter 'o' stands for outline only, and the letter 'c' stands for contrast and adds an outline with the 'foreground' color (black by default).

The non-primitive glyph types and their creator functions are:

Type	R creator function
Text	<a href="#">l_glyph_add_text</a>
Serialaxes	<a href="#">l_glyph_add_serialaxes</a>
Pointranges	<a href="#">l_glyph_add_pointrange</a>
Images	<a href="#">l_glyph_add_image</a>
Polygon	<a href="#">l_glyph_add_polygon</a>

When adding non-primitive glyphs to a display, the number of glyphs needs to match the dimension n of the plot. In other words, a glyph needs to be defined for each observations. See in the examples.

Currently loon does not support compound glyphs. However, it is possible to construct an arbitrary glyph using any system and save it as a png and then re-import them as image glyphs using [l\\_glyph\\_add\\_image](#).

For more information run: `l_help("learn_R_display_plot.html#glyphs")`

### Value

String with glyph id. Every set of non-primitive glyphs has an id (character).

### See Also

[l\\_glyph\\_add\\_text](#), [l\\_make\\_glyphs](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

### Examples

```
# Simple Example with Text Glyphs
p <- with(olive, l_plot(stearic, eicosenoic, color=Region))
g <- l_glyph_add_text(p, text=olive$Area, label="Area")
p['glyph'] <- g

## Not run:
demo("l_glyphs", package="loon")

## End(Not run)

# create a plot that demonstrates the primitive glyphs and the text glyphs
p <- l_plot(x=1:15, y=rep(0,15), size=10, showLabels=FALSE)
text_glyph <- l_glyph_add_text(p, text=letters [1:15])
p['glyph'] <- c(
  'circle', 'ocircle', 'ccircle',
  'square', 'osquare', 'csquare',
  'triangle', 'otriangle', 'ctriangle',
  'diamond', 'odiamond', 'cdiamond',
  rep(text_glyph, 3)
)
```

---

`l_glyph_add.default`     *Default method for adding non-primitive glyphs*

---

### Description

Generic function to write new glyph types using loon's primitive glyphs

### Usage

```
## Default S3 method:
l_glyph_add(widget, type, label = "", ...)
```

**Arguments**

widget	widget path as a string or as an object handle
type	loon-native non-primitive glyph type, one of 'text', 'serialaxes', 'image', '[polygon', or 'pointrange'
label	label of a glyph (currently shown only in the glyph inspector)
...	state arguments

**See Also**

Other glyph functions: [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

---

<code>l_glyph_add_image</code>	<i>Add an image glyphs</i>
--------------------------------	----------------------------

---

**Description**

Image glyphs are useful to show pictures or other sophisticated compound glyphs. Note that images in the Tk canvas support transparency.

**Usage**

```
l_glyph_add_image(widget, images, label = "", ...)
```

**Arguments**

widget	widget path as a string or as an object handle
images	Tk image references, see the <a href="#">l_image_import_array</a> and <a href="#">l_image_import_files</a> helper functions.
label	label of a glyph (currently shown only in the glyph inspector)
...	state arguments

**Details**

For more information run: `l_help("learn_R_display_plot.html#images")`

**See Also**

[l\\_glyph\\_add](#), [l\\_image\\_import\\_array](#), [l\\_image\\_import\\_files](#), [l\\_make\\_glyphs](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

**Examples**

```
## Not run:
p <- with(olive, l_plot(palmitic ~ stearic, color = Region))
img_paths <- list.files(file.path(find.package(package = 'loon'), "images"), full.names = TRUE)
imgs <- setNames(l_image_import_files(img_paths),
                tools::file_path_sans_ext(basename(img_paths)))
i <- pmatch(gsub("^[[:alpha:]]+", "", olive$Area), names(imgs), duplicates.ok = TRUE)

g <- l_glyph_add_image(p, imgs[i], label="Flags")
p['glyph'] <- g

## End(Not run)
```

---

l\_glyph\_add\_pointrange

*Add a Pointrange Glyph*


---

**Description**

Pointrange glyphs show a filled circle at the x-y location and also a y-range.

**Usage**

```
l_glyph_add_pointrange(widget, ymin, ymax, linewidth = 1,
                       showArea = TRUE, label = "", ...)
```

**Arguments**

widget	widget path as a string or as an object handle
ymin	vector with lower y-value of the point range.
ymax	vector with upper y-value of the point range.
linewidth	line width in pixel.
showArea	boolean, show a filled point or just the outline point
label	label of a glyph (currently shown only in the glyph inspector)
...	state arguments

**See Also**

[l\\_glyph\\_add](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

**Examples**

```
p <- l_plot(x = 1:3, color = c('red', 'blue', 'green'), showScales=TRUE)
g <- l_glyph_add_pointrange(p, ymin=(1:3)-(1:3)/5, ymax=(1:3)+(1:3)/5)
p['glyph'] <- g
```

---

`l_glyph_add_polygon`     *Add a Polygon Glyph*

---

**Description**

Add one polygon per scatterplot point.

**Usage**

```
l_glyph_add_polygon(widget, x, y, linewidth = 1, showArea = TRUE,
  label = "", ...)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>x</code>	nested list of x-coordinates of polygons (relative to ), one list element for each scatterplot point.
<code>y</code>	nested list of y-coordinates of polygons, one list element for each scatterplot point.
<code>linewidth</code>	linewidth of outline.
<code>showArea</code>	boolean, show a filled polygon or just the outline
<code>label</code>	label of a glyph (currently shown only in the glyph inspector)
<code>...</code>	state arguments

**Details**

A polygon can be a useful point glyph to visualize arbitrary shapes such as airplanes, animals and shapes that are not available in the primitive glyph types (e.g. cross). The `l_glyphs` demo has an example of polygon glyphs which we reuse here.

**See Also**

[l\\_glyph\\_add](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

**Examples**

```

x_star <-
  c(-0.000864304235090734, 0.292999135695765, 0.949870354364736,
    0.474503025064823, 0.586862575626621, -0.000864304235090734,
    -0.586430423509075, -0.474070872947277, -0.949438202247191,
    -0.29256698357822)
y_star <-
  c(-1, -0.403630077787381, -0.308556611927398, 0.153846153846154,
    0.808556611927398, 0.499567847882455, 0.808556611927398,
    0.153846153846154, -0.308556611927398, -0.403630077787381)
x_cross <-
  c(-0.258931143762604, -0.258931143762604, -0.950374531835206,
    -0.950374531835206, -0.258931143762604, -0.258931143762604,
    0.259651397291847, 0.259651397291847, 0.948934024776722,
    0.948934024776722, 0.259651397291847, 0.259651397291847)
y_cross <-
  c(-0.950374531835206, -0.258931143762604, -0.258931143762604,
    0.259651397291847, 0.259651397291847, 0.948934024776722,
    0.948934024776722, 0.259651397291847, 0.259651397291847,
    -0.258931143762604, -0.258931143762604, -0.950374531835206)
x_hexagon <-
  c(0.773552290406223, 0, -0.773552290406223, -0.773552290406223,
    0, 0.773552290406223)
y_hexagon <-
  c(0.446917314894843, 0.894194756554307, 0.446917314894843,
    -0.447637568424085, -0.892754249495822, -0.447637568424085)

p <- l_plot(1:3, 1:3)

gl <- l_glyph_add_polygon(p, x = list(x_star, x_cross, x_hexagon),
  y = list(y_star, y_cross, y_hexagon))

p['glyph'] <- gl

gl['showArea'] <- FALSE

```

---

l\_glyph\_add\_serialaxes

*Add a Serialaxes Glyph*

---

**Description**

Serialaxes glyph show either a star glyph or a parralel coordinate glyph for each point.

**Usage**

```

l_glyph_add_serialaxes(widget, data, sequence, linewidth = 1,
  scaling = "variable", axesLayout = "radial", showAxes = FALSE,
  axesColor = "gray70", showEnclosing = FALSE, bboxColor = "gray70",
  label = "", ...)

```



**Arguments**

widget	widget path as a string or as an object handle
data	a data frame with numerical data only
sequence	vector with variable names that defines the axes sequence
linewidth	linewidth of outline
scaling	one of 'variable', 'data', 'observation' or 'none' to specify how the data is scaled. See Details and Examples for more information.
axesLayout	either "radial" or "parallel"
showAxes	boolean to indicate whether axes should be shown or not
axesColor	color of axes
showEnclosing	boolean, circle (axesLayout=radial) or square (axesLayout=parallel) to show bounding box/circle of the glyph (or showing unit circle or rectangle with height 1 if scaling=none)
bboxColor	color of bounding box/circle
label	label of a glyph (currently shown only in the glyph inspector)
...	state arguments

**See Also**

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

**Examples**

```
p <- with(olive, l_plot(oleic, stearic, color=Area))
gs <- l_glyph_add_serialaxes(p, data=olive[, -c(1,2)], showArea=FALSE)
p['glyph'] <- gs
```

---

`l_glyph_add_text`      *Add a Text Glyph*

---

**Description**

Each text glyph can be a multiline string.

**Usage**

```
l_glyph_add_text(widget, text, label = "", ...)
```

**Arguments**

widget	widget path as a string or as an object handle
text	the text strings for each observation. If the object is a factor then the labels get extracted with <a href="#">as.character</a> .
label	label of a glyph (currently shown only in the glyph inspector)
...	state arguments

**See Also**

[l\\_glyph\\_add](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

**Examples**

```
p <- l_plot(iris, color = iris$Species)
g <- l_glyph_add_text(p, iris$Species, "test_label")
p['glyph'] <- g
```

---

<code>l_glyph_delete</code>	<i>Delete a Glyph</i>
-----------------------------	-----------------------

---

**Description**

Delete a glyph from the plot.

**Usage**

```
l_glyph_delete(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	glyph id

**See Also**

[l\\_glyph\\_add](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

---

l_glyph_getLabel	<i>Get Glyph Label</i>
------------------	------------------------

---

**Description**

Returns the label of a glyph

**Usage**

```
l_glyph_getLabel(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	glyph id

**See Also**

[l\\_glyph\\_add](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

---

l_glyph_getType	<i>Get Glyph Type</i>
-----------------	-----------------------

---

**Description**

Query the type of a glyph

**Usage**

```
l_glyph_getType(widget, id)
```

**Arguments**

widget	widget path as a string or as an object handle
id	glyph id

**See Also**

[l\\_glyph\\_add](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

---

<code>l_glyph_ids</code>	<i>List glyphs ids</i>
--------------------------	------------------------

---

**Description**

List all the non-primitive glyph ids attached to display.

**Usage**

```
l_glyph_ids(widget)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
---------------------	--

**See Also**

[l\\_glyph\\_add](#)

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_relabel](#), [l\\_primitiveGlyphs](#)

---

<code>l_glyph_relabel</code>	<i>Relabel Glyph</i>
------------------------------	----------------------

---

**Description**

Change the label of a glyph. Note that the label is only displayed in the glyph inspector.

**Usage**

```
l_glyph_relabel(widget, id, label)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>id</code>	glyph id
<code>label</code>	new label

**See Also**

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#), [l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_primitiveGlyphs](#)

**Examples**

```
p <- l_plot(iris, color = iris$Species)
g <- l_glyph_add_text(p, iris$Species, "test_label")
p['glyph'] <- g
l_glyph_relabel(p, g, "Species")
```

---

l_graph	<i>Generic function to create an interactive graph display</i>
---------	--

---

**Description**

Interactive graphs in loon are currently most often used for navigation graphs.

**Usage**

```
l_graph(nodes = NULL, ...)
```

**Arguments**

nodes	object for method dispatch
...	arguments passed on to methods

**Details**

For more information run: `l_help("learn_R_display_graph.html#graph")`

**Value**

graph handle

**See Also**

[l\\_graph.graph](#), [l\\_graph.loongraph](#), [l\\_graph.default](#)

---

l_graph.default	<i>Create a graph display based on node names and from-to edges list</i>
-----------------	--

---

**Description**

This default method uses the loongraph display states as arguments to create a graph display.

**Usage**

```
## Default S3 method:
l_graph(nodes = "", from = "", to = "",
        parent = NULL, ...)
```

**Arguments**

nodes	vector with nodenames
from	vector with node names of the from-to pairs for edges
to	vector with node names of the from-to pairs for edges
parent	parent widget of graph display
...	arguments to modify the graph display state

**Details**

For more information run: `l_help("learn_R_display_graph.html#graph")`

**Value**

graph handle

**See Also**

[loongraph](#), [l\\_graph](#), [l\\_info\\_states](#), [l\\_graph.graph](#)

---

<code>l_graph.graph</code>	<i>Create a graph display based on a graph object</i>
----------------------------	---

---

**Description**

Graph objects are defined in the graph R package.

**Usage**

```
## S3 method for class 'graph'
l_graph(nodes, ...)
```

**Arguments**

nodes	a graph object created with the functions in the graph R package.
...	arguments to modify the graph display state

**Details**

For more information run: `l_help("learn_R_display_graph.html#graph")`

**Value**

graph handle

**See Also**

[l\\_graph](#), [l\\_info\\_states](#), [l\\_graph.loongraph](#)

---

l_graph.loongraph	<i>Create a graph display based on a loongraph object</i>
-------------------	---

---

**Description**

Loongraphs can be created with the [loongraph](#) function.

**Usage**

```
## S3 method for class 'loongraph'
l_graph(nodes, ...)
```

**Arguments**

nodes	a loongraph object created with the <a href="#">loongraph</a> function.
...	arguments to modify the graph display state

**Details**

For more information run: `l_help("learn_R_display_graph.html#graph")`

**Value**

graph handle

**See Also**

[loongraph](#), [l\\_graph](#), [l\\_info\\_states](#), [l\\_graph.graph](#)

---

l_graphswitch	<i>Create a graphswitch widget</i>
---------------	------------------------------------

---

**Description**

The graphswitch provides a graphical user interface for changing the graph in a graph display interactively.

**Usage**

```
l_graphswitch(activewidget = "", parent = NULL, ...)
```

**Arguments**

activewidget	widget handle of a graph display
parent	parent widget path
...	widget states

**Details**

For more information run: `l_help("learn_R_display_graph.html#graph-switch-widget")`

**See Also**

[l\\_graphswitch\\_add](#), [l\\_graphswitch\\_ids](#), [l\\_graphswitch\\_delete](#), [l\\_graphswitch\\_relabel](#), [l\\_graphswitch\\_getLabel](#), [l\\_graphswitch\\_move](#), [l\\_graphswitch\\_reorder](#), [l\\_graphswitch\\_set](#), [l\\_graphswitch\\_get](#)

---

<code>l_graphswitch_add</code>	<i>Add a graph to a graphswitch widget</i>
--------------------------------	--

---

**Description**

This is a generic function to add a graph to a graphswitch widget.

**Usage**

```
l_graphswitch_add(widget, graph, ...)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>graph</code>	a graph or a loongraph object
<code>...</code>	arguments passed on to method

**Details**

For more information run: `l_help("learn_R_display_graph.html#graph-switch-widget")`

**Value**

id for graph in the graphswitch widget

**See Also**

[l\\_graphswitch](#)



---

`l_graphswitch_add.default`*Add a graph that is defined by node names and a from-to edges list*

---

## Description

This default method uses the loongraph display states as arguments to add a graph to the graphswitch widget.

## Usage

```
## Default S3 method:  
l_graphswitch_add(widget, graph, from, to, isDirected,  
  label = "", index = "end", ...)
```

## Arguments

<code>widget</code>	graphswitch widget handle (or widget path)
<code>graph</code>	a vector with the node names, i.e. this argument gets passed on as the nodes argument to create a <a href="#">loongraph</a> like object
<code>from</code>	vector with node names of the from-to pairs for edges
<code>to</code>	vector with node names of the from-to pairs for edges
<code>isDirected</code>	boolean to indicate whether the from-to-list defines directed or undirected edges
<code>label</code>	string with label for graph
<code>index</code>	position of graph in the graph list
<code>...</code>	additional arguments are not used for this method

## Value

id for graph in the graphswitch widget

## See Also

[l\\_graphswitch](#)

---

```
l_graphswitch_add.graph
```

*Add a graph to the graphswitch widget using a graph object*

---

### Description

Graph objects are defined in the graph R package.

### Usage

```
## S3 method for class 'graph'
l_graphswitch_add(widget, graph, label = "",
  index = "end", ...)
```

### Arguments

widget	graphswitch widget handle (or widget path)
graph	a graph object created with the functions in the graph R package.
label	string with label for graph
index	position of graph in the graph list
...	additional arguments are not used for this method

### Value

id for graph in the graphswitch widget

### See Also

[l\\_graphswitch](#)

---

```
l_graphswitch_add.loongraph
```

*Add a graph to the graphswitch widget using a loongraph object*

---

### Description

Loongraphs can be created with the [loongraph](#) function.

### Usage

```
## S3 method for class 'loongraph'
l_graphswitch_add(widget, graph, label = "",
  index = "end", ...)
```

**Arguments**

widget	graphswitch widget handle (or widget path)
graph	a loongraph object
label	string with label for graph
index	position of graph in the graph list
...	additional arguments are not used for this method

**Value**

id for graph in the graphswitch widget

**See Also**

[l\\_graphswitch](#)

---

*l\_graphswitch\_delete*    *Delete a graph from the graphswitch widget*

---

**Description**

Remove a a graph from the graphswitch widget

**Usage**

```
l_graphswitch_delete(widget, id)
```

**Arguments**

widget	graphswitch widget handle (or widget path)
id	of the graph

**See Also**

[l\\_graphswitch](#)

---

`l_graphswitch_get`      *Return a Graph as a loongraph Object*

---

**Description**

Graphs can be extracted from the graphswitch widget as loongraph objects.

**Usage**

```
l_graphswitch_get(widget, id)
```

**Arguments**

<code>widget</code>	graphswitch widget handle (or widget path)
<code>id</code>	of the graph

**See Also**

[l\\_graphswitch](#), [loongraph](#)

---

`l_graphswitch_getLabel`  
*Query Label of a Graph in the Graphswitch Widget*

---

**Description**

The graphs in the graphswitch widgets have labels. Use this function to query the label of a graph.

**Usage**

```
l_graphswitch_getLabel(widget, id)
```

**Arguments**

<code>widget</code>	graphswitch widget handle (or widget path)
<code>id</code>	of the graph

**See Also**

[l\\_graphswitch](#)

---

l_graphswitch_ids	<i>List the ids of the graphs in the graphswitch widget</i>
-------------------	---

---

**Description**

Every graph in the graphswitch widget has an id. This function returns these ids preserving the order of how the graphs are listed in the graphswitch.

**Usage**

```
l_graphswitch_ids(widget)
```

**Arguments**

widget	graphswitch widget handle (or widget path)
--------	--

---

l_graphswitch_move	<i>Move a Graph in the Graph List</i>
--------------------	---------------------------------------

---

**Description**

Change the position in of a graph in the graphswitch widget.

**Usage**

```
l_graphswitch_move(widget, id, index)
```

**Arguments**

widget	graphswitch widget handle (or widget path)
id	of the graph
index	position of the graph as a positive integer, "start" and "end" are also valid keywords.

**See Also**

[l\\_graphswitch](#)

---

`l_graphswitch_relabel` *Relabel a Graph in the Graphswitch Widget*

---

**Description**

The graphs in the graphswitch widgets have labels. Use this function the relabel a graph.

**Usage**

```
l_graphswitch_relabel(widget, id, label)
```

**Arguments**

<code>widget</code>	graphswitch widget handle (or widget path)
<code>id</code>	of the graph
<code>label</code>	string with label of graph

**See Also**

[l\\_graphswitch](#)

---

`l_graphswitch_reorder` *Reorder the Positions of the Graphs in the Graph List*

---

**Description**

Define a new graph order in the graph list.

**Usage**

```
l_graphswitch_reorder(widget, ids)
```

**Arguments**

<code>widget</code>	graphswitch widget handle (or widget path)
<code>ids</code>	vector with all graph ids from the graph widget. Use <a href="#">l_graphswitch_ids</a> to query the ids.

**See Also**

[l\\_graphswitch](#)

---

l_graphswitch_set	<i>Change the Graph shown in the Active Graph Widget</i>
-------------------	--

---

**Description**

The activewidget state holds the widget handle of a graph display. This function replaces the graph in the activewidget with one of the graphs in the graphswitch widget.

**Usage**

```
l_graphswitch_set(widget, id)
```

**Arguments**

widget	graphswitch widget handle (or widget path)
id	of the graph

**See Also**

[l\\_graphswitch](#)

---

l_graph_inspector	<i>Create a Graph Inspector</i>
-------------------	---------------------------------

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_graph_inspector(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_graph_inspector()
```

l\_graph\_inspector\_analysis

*Create a Graph Analysis Inspector*

---

### Description

Inspectors provide graphical user interfaces to oversee and modify plot states

### Usage

```
l_graph_inspector_analysis(parent = NULL, ...)
```

### Arguments

parent	parent widget path
...	state arguments

### Value

widget handle

### See Also

[l\\_create\\_handle](#)

### Examples

```
i <- l_graph_inspector_analysis()
```

---

l\_graph\_inspector\_navigators

*Create a Graph Navigator Inspector*

---

### Description

Inspectors provide graphical user interfaces to oversee and modify plot states

### Usage

```
l_graph_inspector_navigators(parent = NULL, ...)
```

### Arguments

parent	parent widget path
...	state arguments



**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_graph_inspector_navigators()
```

---

l_help	<i>Open a browser with loon's combined (TCL and R) documentation website</i>
--------	--

---

**Description**

l\_help opens a browser with the relevant page on the official combined loon documentation website at <http://great-northern-diver.github.io/loon/>.

**Usage**

```
l_help(page = "index", ...)
```

**Arguments**

page	relative path to a page, the .html part may be omitted
...	arguments forwarded to browseURL, e.g. to specify a browser

**See Also**

[help](#), [l\\_web](#) for R manual or web R manual

**Examples**

```
## Not run:  
l_help()  
l_help("learn_R_intro")  
l_help("learn_R_display_hist")  
l_help("learn_R_bind")  
# jump to a section  
l_help("learn_R_bind.html#list-reorder-delete-bindings")  
  
## End(Not run)
```

---

l_hexcolor	<i>Convert color names to their 12 digit hexadecimal color representation</i>
------------	---

---

**Description**

Color names in loon will be mapped to colors according to the Tk color specifications and are normalized to a 12 digit hexadecimal color representation.

**Usage**

```
l_hexcolor(color)
```

**Arguments**

color            a vector with color names

**Value**

a character vector with the 12 digit hexadecimal color strings.

**Examples**

```
p <- l_plot(1:2)
p['color'] <- 'red'
p['color']

l_hexcolor('red')
```

---

l_hist	<i>Create an interactive histogram</i>
--------	--

---

**Description**

l\_hist is a generic function for creating an interactive histogram display that can be linked with loon's other displays

**Usage**

```
l_hist(x, yshows = c("frequency", "density"), showStackedColors = TRUE,
       origin = NULL, binwidth = NULL, showBinHandle = FALSE,
       xlabel = NULL, parent = NULL, ...)
```

**Arguments**

x	vector with numerical data to perform the binning onx,
yshows	one of "frequency" (default) or "density"
showStackedColors	if TRUE (default) then bars will be coloured according to colours of the points; if FALSE, then the bars will be a uniform colour except for highlighted points.
origin	numeric scalar to define the binning origin
binwidth	a numeric scalar to specify the binwidth If NULL binwidth is set using David Scott's rule when x is numeric and and using the minumum numerical difference between factor levels when x is a factor or a character vector.
showBinHandle	If TRUE, then an interactive "bin handle" appears on the plot whose movement resets the origin and the binwidth. Default is FALSE
xlabel	label to be used on the horizontal axis. If NULL, an attempt at a meaningful label inferred from x will be made.
parent	parent widget path
...	named arguments to modify the histogram plot states

**Details**

Note that when changing the yshows state form 'frequency' to 'density' you might have to use [l\\_scaleto\\_world](#) to show the complete histogram in the plotting region.

For more information run: `l_help("learn_R_display_hist")`

**Value**

widget handle

**See Also**

[l\\_plot](#)

**Examples**

```
h <- l_hist(iris$Sepal.Length)

names(h)
h["xlabel"] <- "Sepal length"
h["showOutlines"] <- FALSE

h["yshows"]
h["yshows"] <- "density"
l_scaleto_plot(h)

h["showStackedColors"] <- TRUE
h['color'] <- iris$Species
h["showStackedColors"] <- FALSE
h["showOutlines"] <- TRUE
```

```
h["showGuides"] <- TRUE

# link another plot with the previous plot
h['linkingGroup'] <- "iris_data"
h2 <- with(iris, l_hist(Petal.Width,
                       linkingGroup="iris_data",
                       showStackedColors = TRUE))

# Get an R (grid) graphics plot of the current loon plot
plot(h)
# or with more control about grid parameters
grid.loon(h)
# or to save the grid data structure (grob) for later use
hg <- loonGrob(h)
```

---

l\_hist\_inspector      *Create a Histogram Inspector*

---

## Description

Inspectors provide graphical user interfaces to oversee and modify plot states

## Usage

```
l_hist_inspector(parent = NULL, ...)
```

## Arguments

parent	parent widget path
...	state arguments

## Value

widget handle

## See Also

[l\\_create\\_handle](#)

## Examples

```
i <- l_hist_inspector()
```

---

`l_hist_inspector_analysis`*Create a Histogram Analysis Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_hist_inspector_analysis(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_hist_inspector_analysis()
```

---

`l_imageviewer`*Display Tcl Images in a Simple Image Viewer*

---

**Description**

Loon provides a simple image viewer to browse through the specified tcl image objects.

The simple GUI supports either the use of the mouse or left and right arrow keys to switch the images to the previous or next image in the specified image vector.

The images are resized to fill the viewer window.

**Usage**

```
l_imageviewer(tclimages)
```

**Arguments**

tclimages	Vector of tcl image object names.
-----------	-----------------------------------

**Value**

the tclimages vector is returned

**Examples**

```
img2 <- tkimage.create('photo', width=200, height=150)
tcl(img2, 'put', 'yellow', '-to', 0, 0, 199, 149)
tcl(img2, 'put', 'green', '-to', 40, 20, 130, 40)
img3 <- tkimage.create('photo', width=500, height=100)
tcl(img3, 'put', 'orange', '-to', 0, 0, 499, 99)
tcl(img3, 'put', 'green', '-to', 40, 80, 350, 95)

l_imageviewer(c(tclvalue(img2), tclvalue(img3)))
```

---

`l_image_import_array` *Import Greyscale Images as Tcl images from an Array*

---

**Description**

Import image grayscale data (0-255) with each image saved as a row or column of an array.

**Usage**

```
l_image_import_array(array, width, height, img_in_row = TRUE,
  invert = FALSE, rotate = 0)
```

**Arguments**

<code>array</code>	of 0-255 grayscale value data.
<code>width</code>	of images in pixels.
<code>height</code>	of images in pixels.
<code>img_in_row</code>	logical, TRUE if every row of the array represents an image
<code>invert</code>	logical, for 'invert=FALSE' 0=white, for 'invert=TRUE' 0=black
<code>rotate</code>	the image: one of 0, 90, 180, or 270 degrees.

**Details**

Images in tcl are managed by the tcl interpreter and made accessible to the user via a handle, i.e. a function name of the form `image1`, `image2`, etc.

For more information run: `l_help("learn_R_display_plot.html#images")`

**Value**

vector of image object names

**Examples**

```
## Not run:  
# see  
demo("l_ng_images_frey_LLE")  
  
## End(Not run)
```

---

*l\_image\_import\_files* *Import Image Files as Tk Image Objects*

---

**Description**

Note that the supported image file formats depend on whether the Img Tk extension is installed.

**Usage**

```
l_image_import_files(paths)
```

**Arguments**

paths            vector with paths to image files that are supported

**Details**

For more information run: `l_help("learn_R_display_plot.html#load-images")`

**Value**

vector of image object names

**See Also**

[l\\_image\\_import\\_array](#), [l\\_imageviewer](#)

---

l\_info\_states

*Retrieve Information about the States of a Loon Widget*


---

### Description

Loon's built-in object documentation. Can be used with every loon object that has plot states including plots, layers, navigators, contexts. This is a generic function.

### Usage

```
l_info_states(target, states = "all")
```

### Arguments

target	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. '.l0.plot'), the remaining objects by their ids.
states	vector with names of states. 'all' is treated as a keyword and results in returning information on all plot states

### Value

a named nested list with one element per state. The list elements are also named lists with type, dimension, defaultvalue, and description elements containing the respective information.

### See Also

Other loon interactive states: [l\\_plot.default](#), [l\\_plot](#), [l\\_state\\_names](#), [names.loon](#)

### Examples

```
p <- l_plot(iris, linkingGroup="iris")
i <- l_info_states(p)
names(p)
names(i)
i$selectBy

l <- l_layer_rectangle(p, x=range(iris[,1]), y=range(iris[,2]), color="")
l_info_states(l)

h <- l_hist(iris$Sepal.Length, linkingGroup="iris")
l_info_states(h)
```



---

l_isLoonWidget	<i>Check if a widget path is a valid loon widget</i>
----------------	--

---

**Description**

This function can be useful to check whether a loon widget is has been closed by the user.

**Usage**

```
l_isLoonWidget(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

**Value**

boolean, TRUE if the argument is a valid loon widget path, FALSE otherwise

---

l_layer	<i>Loon layers</i>
---------	--------------------

---

**Description**

Loon supports layering of visuals and groups of visuals. The l\_layer function is a generic method.

**Usage**

```
l_layer(widget, x, ...)
```

**Arguments**

widget	widget path as a string or as an object handle
x	object that should be layered
...	additional arguments, often state definition for the basic layering function

**Details**

loon's displays that use the main graphics model (i.e. histogram, scatterplot and graph displays) support layering of visual information. The following table lists the layer types and functions for layering on a display.

Type	Description	Creator Function
group	a group can be a parent of other layers	<a href="#">l_layer_group</a>
polygon	one polygon	<a href="#">l_layer_polygon</a>
text	one text string	<a href="#">l_layer_text</a>

line	one line (i.e. connected line segments)	<a href="#">l_layer_line</a>
rectangle	one rectangle	<a href="#">l_layer_rectangle</a>
oval	one oval	<a href="#">l_layer_oval</a>
points	n points (filled) circle	<a href="#">l_layer_points</a>
texts	n text strings	<a href="#">l_layer_text</a>
polygons	n polygons	<a href="#">l_layer_polygons</a>
rectangles	n rectangles	<a href="#">l_layer_rectangles</a>
lines	n sets of connected line segments	<a href="#">l_layer_lines</a>

Every layer within a display has a unique id. The visuals of the data in a display present the default layer of that display and has the layer id 'model'. For example, the 'model' layer of a scatterplot display visualizes the scatterplot glyphs. Functions useful to query layers are

Function	Description
<a href="#">l_layer_ids</a>	List layer ids
<a href="#">l_layer_getType</a>	Get layer type

Layers are arranged in a tree structure with the tree root having the layer id 'root'. The rendering order of the layers is according to a depth-first traversal of the layer tree. This tree also maintains a label and a visibility flag for each layer. The layer tree, layer ids, layer labels and the visibility of each layer are visualized in the layers inspector. If a layer is set to be invisible then it is not rendered on the display. If a group layer is set to be invisible then all its children are not rendered; however, the visibility flag of the children layers remain unchanged. Relevant functions are:

Function	Description
<a href="#">l_layer_getParent</a>	Get parent layer id of a layer
<a href="#">l_layer_getChildren</a>	Get children of a group layer
<a href="#">l_layer_index</a>	Get the order index of a layer among its siblings
<a href="#">l_layer_printTree</a>	Print out the layer tree
<a href="#">l_layer_move</a>	Move a layer
<a href="#">l_layer_lower</a>	Switch the layer place with its sibling to the right
<a href="#">l_layer_raise</a>	Switch the layer place with its sibling to the left
<a href="#">l_layer_demote</a>	Moves the layer up to be a left sibling of its parent
<a href="#">l_layer_promote</a>	Moves the layer to be a child of its right group layer sibling
<a href="#">l_layer_hide</a>	Set the layers visibility flag to FALSE
<a href="#">l_layer_show</a>	Set the layers visibility flag to TRUE
<a href="#">l_layer_isVisible</a>	Return visibility flag of layer
<a href="#">l_layer_layerVisibility</a>	Returns logical value for whether layer is actually seen
<a href="#">l_layer_groupVisibility</a>	Returns all, part or none for expressing which part of the layers children are visible.
<a href="#">l_layer_delete</a>	Delete a layer. If the layer is a group move all its children layers to the layers parent.
<a href="#">l_layer_expunge</a>	Delete layer and all its children layer.
<a href="#">l_layer_getLabel</a>	Get layer label.
<a href="#">l_layer_relabel</a>	Change layer label.
<a href="#">l_layer_bbox</a>	Get the bounding box of a layer.

All layers have states that can be queried and modified using the same functions as the ones used for displays (i.e. [l\\_cget](#), [l\\_configure](#), ``[`` and ``[<-``). The last group of layer types in the above

table have n-dimensional states, where the actual value of n can be different for every layer in a display.

The difference between the model layer and the other layers is that the model layer has a *selected* state, responds to selection gestures and supports linking.

For more information run: `l_help("learn_R_layer")`

### Value

layer object handle, layer id

### See Also

[l\\_info\\_states](#), [l\\_scaletto\\_layer](#), [l\\_scaletto\\_world](#)

### Examples

```
# l_layer is a generic method
newFoo <- function(x, y, ...) {
  r <- list(x=x, y=y, ...)
  class(r) <- 'foo'
  return(r)
}

l_layer.foo <- function(widget, x) {
  x$widget <- widget
  id <- do.call('l_layer_polygon', x)
  return(id)
}

p <- l_plot()

obj <- newFoo(x=c(1:6,6:2), y=c(3,1,0,0,1,3,3,5,6,6,5), color='yellow')

id <- l_layer(p, obj)

l_scaletto_world(p)
```

---

`l_layer.density`

*Layer Method for Kernel Density Estimation*

---

### Description

Layer a line that represents a kernel density estimate.

### Usage

```
## S3 method for class 'density'
l_layer(widget, x, ...)
```

**Arguments**

widget            widget path as a string or as an object handle  
 x                    object from [density](#) of class "density"  
 ...                  additional arguments, often state definition for the basic layering function

**Value**

layer object handle, layer id

**See Also**

[density](#), [l\\_layer](#)

**Examples**

```

d <- density(faithful$eruptions, bw = "sj")
h <- l_hist(x = faithful$eruptions, yshows="density")
l <- l_layer.density(h, d, color="steelblue", linewidth=3)

```

---

<code>l_layer.Line</code>	<i>Layer line in Line object</i>
---------------------------	----------------------------------

---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```

## S3 method for class 'Line'
l_layer(widget, x, ...)

```

**Arguments**

widget            widget widget path as a string or as an object handle  
 x                    an object defined in the class [sp](#)  
 ...                  arguments forwarded to the relative [l\\_layer](#) function

**Details**

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {
  world <- rworldmap::getMap(resolution = "coarse")
  p <- l_plot()
  lmap <- l_layer(p, world, asSingleLayer=TRUE)
  l_scaleto_world(p)
  attr(lmap, 'hole')
  attr(lmap, 'NAME')
}
```

---

l\_layer.Lines

*Layer lines in Lines object*


---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```
## S3 method for class 'Lines'
l_layer(widget, x, asSingleLayer = TRUE, ...)
```

**Arguments**

widget	widget widget path as a string or as an object handle
x	an object defined in the class <a href="#">sp</a>
asSingleLayer	If TRUE then prefer a single layer over groups with nested 1-dimensinal layers
...	arguments forwarded to the relative <a href="#">l_layer</a> function

**Details**

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

## References

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

## See Also

[sp](#), [l\\_layer](#)

## Examples

```
if (requireNamespace("rworldmap", quietly = TRUE)) {
  world <- rworldmap::getMap(resolution = "coarse")
  p <- l_plot()
  lmap <- l_layer(p, world, asSingleLayer=TRUE)
  l_scaletoworld(p)
  attr(lmap, 'hole')
  attr(lmap, 'NAME')
}
```

---

`l_layer.map`

*Add a Map of class map as Drawings to Loon plot*

---

## Description

The maps library provides some map data in polygon which can be added as drawings (currently with polygons) to Loon plots. This function adds map objects with class map from the maps library as background drawings.

## Usage

```
## S3 method for class 'map'
l_layer(widget, x, color = "", linecolor = "black",
  linewidth = 1, label, parent = "root", index = 0,
  asSingleLayer = TRUE, ...)
```

## Arguments

<code>widget</code>	widget path as a string or as an object handle
<code>x</code>	a map object of class <code>map</code> as defined in the maps R package
<code>color</code>	fill color, if empty string "", then the fill is transparent
<code>linecolor</code>	outline color
<code>linewidth</code>	linewidth of outline
<code>label</code>	label used in the layers inspector
<code>parent</code>	parent widget path
<code>index</code>	position among its siblings. valid values are 0, 1, 2, ..., 'end'

asSingleLayer if TRUE then all the polygons get placed in a n-dimension layer of type polygons. Otherwise, if FALSE, each polygon gets its own layer.  
 ... additional arguments are not used for this method

### Value

If asSingleLayer=TRUE then returns layer id of polygons layer, otherwise group layer that contains polygon children layers.

### Examples

```
if (requireNamespace("maps", quietly = TRUE)) {
  canada <- maps::map("world", "Canada",
                    fill=TRUE, plot=FALSE)
  p <- l_plot()
  l_map <- l_layer(p, canada,
                asSingleLayer=TRUE, color = "cornsilk")
  l_map['color'] <- ifelse(grepl("lake", canada$names, TRUE),
                        "lightblue", "cornsilk")
  l_scaleto_layer(p, l_map)
  l_map['active'] <- FALSE
  l_map['active'] <- TRUE
  l_map['tag']
}
```

---

<code>l_layer.Polygon</code>	<i>Layer polygon in Polygon object</i>
------------------------------	--

---

### Description

Methods to plot map data defined in the [sp](#) package

### Usage

```
## S3 method for class 'Polygon'
l_layer(widget, x, ...)
```

### Arguments

widget widget widget path as a string or as an object handle  
 x an object defined in the class [sp](#)  
 ... arguments forwarded to the relative [l\\_layer](#) function

### Details

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {
  world <- rworldmap::getMap(resolution = "coarse")
  p <- l_plot()
  lmap <- l_layer(p, world, asSingleLayer=TRUE)
  l_scaleto_world(p)
  attr(lmap, 'hole')
  attr(lmap, 'NAME')
}
```

---

`l_layer.Polygons`

*Layer polygons in Polygons object*

---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```
## S3 method for class 'Polygons'
l_layer(widget, x, asSingleLayer = TRUE, ...)
```

**Arguments**

<code>widget</code>	widget widget path as a string or as an object handle
<code>x</code>	an object defined in the class <a href="#">sp</a>
<code>asSingleLayer</code>	If TRUE then prefer a single layer over groups with nested 1-dimensional layers
<code>...</code>	arguments forwarded to the relative <a href="#">l_layer</a> function

**Details**

Note that currently loon does neither support holes and ring directions.



**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {  
  world <- rworldmap::getMap(resolution = "coarse")  
  p <- l_plot()  
  lmap <- l_layer(p, world, asSingleLayer=TRUE)  
  l_scaleto_world(p)  
  attr(lmap, 'hole')  
  attr(lmap, 'NAME')  
}
```

---

`l_layer.SpatialLines` *Layer lines in SpatialLines object*

---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```
## S3 method for class 'SpatialLines'  
l_layer(widget, x, asSingleLayer = TRUE, ...)
```

**Arguments**

<code>widget</code>	widget widget path as a string or as an object handle
<code>x</code>	an object defined in the class <a href="#">sp</a>
<code>asSingleLayer</code>	If TRUE then prefer a single layer over groups with nested 1-dimensional layers
<code>...</code>	arguments forwarded to the relative <a href="#">l_layer</a> function

**Details**

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {
  world <- rworldmap::getMap(resolution = "coarse")
  p <- l_plot()
  lmap <- l_layer(p, world, asSingleLayer=TRUE)
  l_scaletoworld(p)
  attr(lmap, 'hole')
  attr(lmap, 'NAME')
}
```

---

`l_layer.SpatialLinesDataFrame`

*Layer lines in SpatialLinesDataFrame object*

---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```
## S3 method for class 'SpatialLinesDataFrame'
l_layer(widget, x, asSingleLayer = TRUE,
  ...)
```

**Arguments**

<code>widget</code>	widget widget path as a string or as an object handle
<code>x</code>	an object defined in the class <a href="#">sp</a>
<code>asSingleLayer</code>	If TRUE then prefer a single layer over groups with nested 1-dimensional layers
<code>...</code>	arguments forwarded to the relative <a href="#">l_layer</a> function

**Details**

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {
  world <- rworldmap::getMap(resolution = "coarse")
  p <- l_plot()
  lmap <- l_layer(p, world, asSingleLayer=TRUE)
  l_scaleto_world(p)
  attr(lmap, 'hole')
  attr(lmap, 'NAME')
}
```

---

`l_layer.SpatialPoints` *Layer points in SpatialPoints object*

---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```
## S3 method for class 'SpatialPoints'
l_layer(widget, x, asMainLayer = FALSE, ...)
```

**Arguments**

<code>widget</code>	widget widget path as a string or as an object handle
<code>x</code>	an object defined in the class <a href="#">sp</a>
<code>asMainLayer</code>	if TRUE and the widget is a scatterplot widget, then points can be chosen to be added to the 'model' layer
<code>...</code>	arguments forwarded to the relative <a href="#">l_layer</a> function

**Details**

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {
  world <- rworldmap::getMap(resolution = "coarse")
  p <- l_plot()
  lmap <- l_layer(p, world, asSingleLayer=TRUE)
  l_scaleto_world(p)
  attr(lmap, 'hole')
  attr(lmap, 'NAME')
}
```

---

`l_layer.SpatialPointsDataFrame`

*Layer points in SpatialPointsDataFrame object*

---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```
## S3 method for class 'SpatialPointsDataFrame'
l_layer(widget, x, asMainLayer = FALSE,
  ...)
```

**Arguments**

<code>widget</code>	widget widget path as a string or as an object handle
<code>x</code>	an object defined in the class <a href="#">sp</a>
<code>asMainLayer</code>	if TRUE and the widget is a scatterplot widget, then points can be chosen to be added to the 'model' layer
<code>...</code>	arguments forwarded to the relative <a href="#">l_layer</a> function

**Details**

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {
  world <- rworldmap::getMap(resolution = "coarse")
  p <- l_plot()
  lmap <- l_layer(p, world, asSingleLayer=TRUE)
  l_scaleto_world(p)
  attr(lmap, 'hole')
  attr(lmap, 'NAME')
}
```

---

`l_layer.SpatialPolygons`

*Layer polygons in SpatialPolygons object*

---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```
## S3 method for class 'SpatialPolygons'
l_layer(widget, x, asSingleLayer = TRUE, ...)
```

**Arguments**

<code>widget</code>	widget widget path as a string or as an object handle
<code>x</code>	an object defined in the class <a href="#">sp</a>
<code>asSingleLayer</code>	If TRUE then prefer a single layer over groups with nested 1-dimensional layers
<code>...</code>	arguments forwarded to the relative <a href="#">l_layer</a> function

**Details**

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {
  world <- rworldmap::getMap(resolution = "coarse")
  p <- l_plot()
  lmap <- l_layer(p, world, asSingleLayer=TRUE)
  l_scaleto_world(p)
  attr(lmap, 'hole')
  attr(lmap, 'NAME')
}
```

---

`l_layer.SpatialPolygonsDataFrame`

*Layer polygons in SpatialPolygonDataFrame*

---

**Description**

Methods to plot map data defined in the [sp](#) package

**Usage**

```
## S3 method for class 'SpatialPolygonsDataFrame'
l_layer(widget, x,
        asSingleLayer = TRUE, ...)
```

**Arguments**

<code>widget</code>	widget widget path as a string or as an object handle
<code>x</code>	an object defined in the class <a href="#">sp</a>
<code>asSingleLayer</code>	If TRUE then prefer a single layer over groups with nested 1-dimensional layers
<code>...</code>	arguments forwarded to the relative <a href="#">l_layer</a> function

**Details**

Note that currently loon does neither support holes and ring directions.

**Value**

layer id

**References**

Applied Spatial Data Analysis with R by Bivand, Roger S. and Pebesma, Edzer and Gomez-Rubio and Virgilio <http://www.springer.com/us/book/9781461476177>

**See Also**

[sp](#), [l\\_layer](#)

**Examples**

```
if (requireNamespace("rworldmap", quietly = TRUE)) {  
  world <- rworldmap::getMap(resolution = "coarse")  
  p <- l_plot()  
  lmap <- l_layer(p, world, asSingleLayer=TRUE)  
  l_scaleto_world(p)  
  attr(lmap, 'hole')  
  attr(lmap, 'NAME')  
}
```

---

`l_layers_inspector`      *Create a Layers Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_layers_inspector(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**[l\\_create\\_handle](#)**Examples**

```
i <- l_layers_inspector()
```

---

l_layer_bbox	<i>Get the bounding box of a layer.</i>
--------------	---

---

**Description**

The bounding box of a layer returns the coordinates of the smallest rectangle that encloses all the elements of the layer.

**Usage**

```
l_layer_bbox(widget, layer = "root")
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Value**

Numeric vector of length 4 with (xmin, ymin, xmax, ymax) of the bounding box

**Examples**

```
p <- with(iris, l_plot(Sepal.Length ~ Sepal.Width, color=Species))
l_layer_bbox(p, layer='model')

l <- l_layer_rectangle(p, x=0:1, y=30:31)
l_layer_bbox(p, l)

l_layer_bbox(p, 'root')
```



---

 l\_layer\_contourLines *Layer Contour Lines*


---

### Description

This function is a wrapper around [contourLines](#) that adds the countourlines to a loon plot which is based on the cartesian coordinate system.

### Usage

```
l_layer_contourLines(widget, x = seq(0, 1, length.out = nrow(z)),
  y = seq(0, 1, length.out = ncol(z)), z, nlevels = 10,
  levels = pretty(range(z, na.rm = TRUE), nlevels),
  asSingleLayer = TRUE, parent = "root", index = "end", ...)
```

### Arguments

widget	widget path as a string or as an object handle
x	locations of grid lines at which the values in z are measured. These must be in ascending order. By default, equally spaced values from 0 to 1 are used. If x is a list, its components x\$x and x\$y are used for x and y, respectively. If the list has component z this is used for z.
y	see description for the x argument
z	a matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.
nlevels	number of contour levels desired <b>iff</b> levels is not supplied.
levels	numeric vector of levels at which to draw contour lines.
asSingleLayer	if TRUE a lines layer is used for the line, otherwise if FALSE a group with nested line layers for each line is created
parent	parent widget path
index	position among its siblings. valid values are 0, 1, 2, ..., 'end'
...	argumnets forwarded to <a href="#">l_layer_line</a>

### Details

For more information run: `l_help("learn_R_layer.html#countourlines-heatimage-rasterimage")`

### Value

layer id of group or lines layer

**Examples**

```

p <- l_plot()
x <- 10*1:nrow(volcano)
y <- 10*1:ncol(volcano)
lcl <- l_layer_contourLines(p, x, y, volcano)
l_scaleto_world(p)

if (requireNamespace("MASS", quietly = TRUE)) {

  p1 <- with(iris, l_plot(Sepal.Length~Sepal.Width, color=Species))
  lcl <- with(iris, l_layer_contourLines(p1, MASS::kde2d(Sepal.Width,Sepal.Length)))

  p2 <- with(iris, l_plot(Sepal.Length~Sepal.Width, color=Species))
  layers <- sapply(split(cbind(iris, color=p2['color']), iris$Species), function(dat) {
    kest <- with(dat, MASS::kde2d(Sepal.Width,Sepal.Length))
    l_layer_contourLines(p2, kest, color=as.character(dat$color[1]), linewidth=2,
      label=paste0(as.character(dat$Species[1]), " contours"))
  })
}

```

---

`l_layer_delete`*Delete a layer*

---

**Description**

All but the 'model' and the 'root' layer can be dynamically deleted. If a group layer gets deleted with `l_layer_delete` then all its children layers get moved into their grandparent group layer.

**Usage**

```
l_layer_delete(widget, layer)
```

**Arguments**

<code>widget</code>	widget path or layer object of class 'l_layer'
<code>layer</code>	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Value**

0 if success otherwise the function throws an error

**See Also**

[l\\_layer](#), [l\\_info\\_states](#)

**Examples**

```
p <- l_plot()
l1 <- l_layer_rectangle(p, x = 0:1, y = 0:1, color='red')
l_layer_delete(l1)

l2 <- l_layer_rectangle(p, x = 0:1, y = 0:1, color='yellow')
l_layer_delete(p,l2)
```

---

l_layer_demote	<i>Moves the layer to be a child of its right group layer sibling</i>
----------------	---

---

**Description**

Moves the layer up the layer tree (away from the root layer) if there is a sibling group layer to the right of the layer.

**Usage**

```
l_layer_demote(widget, layer)
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Value**

0 if success otherwise the function throws an error

**Examples**

```
p <- l_plot()

g1 <- l_layer_group(p)
g2 <- l_layer_group(p, parent=g1)
l1 <- l_layer_oval(p, x=0:1, y=0:1)

l_layer_printTree(p)
l_layer_demote(p, l1)
l_layer_printTree(p)
l_layer_demote(p, l1)
l_layer_printTree(p)
```

---

l_layer_expunge	<i>Delete a layer and all its descendants</i>
-----------------	---

---

**Description**

Delete a group layer and all its descendants. Note that the 'model' layer cannot be deleted.

**Usage**

```
l_layer_expunge(widget, layer)
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Value**

0 if success otherwise the function throws an error

**See Also**

[l\\_layer](#), [l\\_layer\\_delete](#)

**Examples**

```
p <- l_plot()
g <- l_layer_group(p)
l1 <- l_layer_rectangle(p, x=0:1, y=0:1, parent=g, color="", linecolor="orange", linewidth=2)
l2 <- l_layer_line(p, x=c(0,.5,1), y=c(0,1,0), parent=g, color="blue")

l_layer_expunge(p, g)

# or l_layer_expunge(g)
```

---

l_layer_getChildren	<i>Get children of a group layer</i>
---------------------	--------------------------------------

---

**Description**

Returns the ids of a group layer's children.

**Usage**

```
l_layer_getChildren(widget, layer = "root")
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Value**

Character vector with ids of the childrens. To create layer handles (i.e. objects of class 'l\_layer') use the [l\\_create\\_handle](#) function.

**See Also**

[l\\_layer](#), [l\\_layer\\_getParent](#)

**Examples**

```
p <- l_plot()

g <- l_layer_group(p)
l1 <- l_layer_rectangle(p, x=0:1, y=0:1, parent=g)
l2 <- l_layer_oval(p, x=0:1, y=0:1, color='thistle', parent=g)

l_layer_getChildren(p, g)
```

---

<code>l_layer_getLabel</code>	<i>Get layer label.</i>
-------------------------------	-------------------------

---

**Description**

Layer labels are useful to identify layer in the layer inspector. The layer label can be initially set at layer creation with the label argument.

**Usage**

```
l_layer_getLabel(widget, layer)
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Details**

Note that the layer label is not a state of the layer itself, instead is information that is part of the layer collection (i.e. its parent widget).

**Value**

Named vector of length 1 with layer label as value and layer id as name.

**See Also**

[l\\_layer](#), [l\\_layer\\_relabel](#)

**Examples**

```
p <- l_plot()
l1 <- l_layer_rectangle(p, x=0:1, y=0:1, label="a rectangle")
l_layer_getLabel(p, 'model')
l_layer_getLabel(p, l1)
```

---

<code>l_layer_getParent</code>	<i>Get parent layer id of a layer</i>
--------------------------------	---------------------------------------

---

**Description**

The toplevel parent is the 'root' layer.

**Usage**

```
l_layer_getParent(widget, layer)
```

**Arguments**

<code>widget</code>	widget path or layer object of class 'l_layer'
<code>layer</code>	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**See Also**

[l\\_layer](#), [l\\_layer\\_getChildren](#)

**Examples**

```
p <- with(iris, l_plot(Sepal.Length ~ Sepal.Width, color=Species))
l_layer_getParent(p, 'model')
```

---

l_layer_getType	<i>Get layer type</i>
-----------------	-----------------------

---

### Description

To see the manual page of [l\\_layer](#) for all the primitive layer types.

### Usage

```
l_layer_getType(widget, layer)
```

### Arguments

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

### Details

For more information run: `l_help("learn_R_layer")`

### Value

One of: 'group', 'polygon', 'text', 'line', 'rectangle', 'oval', 'points', 'texts', 'polygons', 'rectangles', 'lines' and 'scatterplot', 'histogram', 'serialaxes' and 'graph'.

### See Also

[l\\_layer](#)

### Examples

```
p <- l_plot()
l <- l_layer_rectangle(p, x=0:1, y=0:1)
l_layer_getType(p, l)
l_layer_getType(p, 'model')
```

---

l_layer_group	<i>layer a group node</i>
---------------	---------------------------

---

### Description

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

A group layer can contain other layers. If the group layer is invisible, then so are all its children.

### Usage

```
l_layer_group(widget, label = "group", parent = "root", index = 0)
```

### Arguments

widget	widget path name as a string
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group

### Details

For more information run: `l_help("learn_R_layer")`

### Value

layer object handle, layer id

### See Also

[l\\_layer](#), [l\\_info\\_states](#)

### Examples

```
p <- l_plot(x=c(1,10,1.5,7,4.3,9,5,2,8),
           y=c(1,10,7,3,4,3.3,8,3,4),
           title="Demo Layers")

id.g <- l_layer_group(p, "A Layer Group")
id.pts <- l_layer_points(p, x=c(3,6), y=c(4,7), color="red", parent=id.g)
l_scaleto_layer(p, id.pts)
l_configure(id.pts, x=c(-5,5,12), y=c(-2,-5,18), color="lightgray")
```



---

**l\_layer\_groupVisibility***Queries visibility status of descendants*

---

**Description**

Query whether all, part or none of the group layers descendants are visible.

**Usage**

```
l_layer_groupVisibility(widget, layer)
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Details**

Visible layers are rendered, invisible ones are not. If any ancestor of a layer is set to be invisible then the layer is not rendered either. The layer visibility flag can be checked with [l\\_layer\\_isVisible](#) and the actual visibility (i.e. are all the ancestors visible too) can be checked with [l\\_layer\\_layerVisibility](#).

Note that layer visibility is not a state of the layer itself, instead is information that is part of the layer collection (i.e. its parent widget).

**Value**

'all', 'part' or 'none' depending on the visibility status of the descendants.

**See Also**

[l\\_layer](#), [l\\_layer\\_show](#), [l\\_layer\\_hide](#), [l\\_layer\\_isVisible](#), [l\\_layer\\_layerVisibility](#)

**Examples**

```
p <- l_plot()

g <- l_layer_group(p)
l1 <- l_layer_rectangle(p, x=0:1, y=0:1, parent=g)
l2 <- l_layer_oval(p, x=0:1, y=0:1, parent=g)

l_layer_groupVisibility(p, g)
l_layer_hide(p, l2)
l_layer_groupVisibility(p, g)
l_layer_hide(p, l1)
l_layer_groupVisibility(p, g)
```

```
l_layer_hide(p, g)
l_layer_groupVisibility(p, g)
```

---

`l_layer_heatImage`      *Display a Heat Image*

---

### Description

This function is very similar to the `image` function. It works with every loon plot which is based on the cartesian coordinate system.

### Usage

```
l_layer_heatImage(widget, x = seq(0, 1, length.out = nrow(z)),
  y = seq(0, 1, length.out = ncol(z)), z,
  zlim = range(z[is.finite(z)]), xlim = range(x), ylim = range(y),
  col = grDevices::heat.colors(12), breaks, oldstyle = FALSE,
  useRaster, index = "end", parent = "root", ...)
```

### Arguments

<code>widget</code>	widget path as a string or as an object handle
<code>x</code>	locations of grid lines at which the values in <code>z</code> are measured. These must be finite, non-missing and in (strictly) ascending order. By default, equally spaced values from 0 to 1 are used. If <code>x</code> is a list, its components <code>x\$x</code> and <code>x\$y</code> are used for <code>x</code> and <code>y</code> , respectively. If the list has component <code>z</code> this is used for <code>z</code> .
<code>y</code>	see description for the <code>x</code> argument above
<code>z</code>	a numeric or logical matrix containing the values to be plotted (NAs are allowed). Note that <code>x</code> can be used instead of <code>z</code> for convenience.
<code>zlim</code>	the minimum and maximum <code>z</code> values for which colors should be plotted, defaulting to the range of the finite values of <code>z</code> . Each of the given colors will be used to color an equispaced interval of this range. The <i>midpoints</i> of the intervals cover the range, so that values just outside the range will be plotted.
<code>xlim</code>	range for the plotted <code>x</code> values, defaulting to the range of <code>x</code>
<code>ylim</code>	range for the plotted <code>y</code> values, defaulting to the range of <code>y</code>
<code>col</code>	a list of colors such as that generated by <code>rainbow</code> , <code>heat.colors</code> , <code>topo.colors</code> , <code>terrain.colors</code> or similar functions.
<code>breaks</code>	a set of finite numeric breakpoints for the colours: must have one more breakpoint than colour and be in increasing order. Unsorted vectors will be sorted, with a warning.
<code>oldstyle</code>	logical. If true the midpoints of the colour intervals are equally spaced, and <code>zlim[1]</code> and <code>zlim[2]</code> were taken to be midpoints. The default is to have colour intervals of equal lengths between the limits.

useRaster	logical; if TRUE a bitmap raster is used to plot the image instead of polygons. The grid must be regular in that case, otherwise an error is raised. For the behaviour when this is not specified, see 'Details'.
index	position among its siblings. valid values are 0, 1, 2, ..., 'end'
parent	parent widget path
...	arguments forwarded to <a href="#">l_layer_line</a>

### Details

For more information run: `l_help("learn_R_layer.html#countourlines-heatmap-rasterimage")`

### Value

layer id of group or rectangles layer

### Examples

```
if (requireNamespace("MASS", quietly = TRUE)) {
  kest <- with(iris, MASS::kde2d(Sepal.Width, Sepal.Length))
  image(kest)
  contour(kest, add=TRUE)

  p <- l_plot()
  lcl <- l_layer_contourLines(p, kest, label='contour lines')
  lim <- l_layer_heatImage(p, kest, label='heatmap')
  l_scaleto_world(p)
}

# from examples(image)
x <- y <- seq(-4*pi, 4*pi, len = 27)
r <- sqrt(outer(x^2, y^2, "+"))
p1 <- l_plot()
l_layer_heatImage(p1, z = z <- cos(r^2)*exp(-r/6), col = gray((0:32)/32))
l_scaleto_world(p1)

image(z = z <- cos(r^2)*exp(-r/6), col = gray((0:32)/32))
```

---

l\_layer\_hide

*Hide a Layer*

---

### Description

A hidden layer is not rendered. If a group layer is set to be hidden then all its descendants are not rendered either.

### Usage

```
l_layer_hide(widget, layer)
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Details**

Visible layers are rendered, invisible ones are not. If any ancestor of a layer is set to be invisible then the layer is not rendered either. The layer visibility flag can be checked with [l\\_layer\\_isVisible](#) and the actual visibility (i.e. are all the ancestors visible too) can be checked with [l\\_layer\\_layerVisibility](#).

Note that layer visibility is not a state of the layer itself, instead is information that is part of the layer collection (i.e. its parent widget).

**Value**

0 if success otherwise the function throws an error

**See Also**

[l\\_layer](#), [l\\_layer\\_show](#), [l\\_layer\\_isVisible](#), [l\\_layer\\_layerVisibility](#), [l\\_layer\\_groupVisibility](#)

**Examples**

```
p <- l_plot()

l <- l_layer_rectangle(p, x=0:1, y=0:1, color="steelblue")
l_layer_hide(p, l)
```

---

<code>l_layer_ids</code>	<i>List ids of layers in Plot</i>
--------------------------	-----------------------------------

---

**Description**

Every layer within a display has a unique id. This function returns a list of all the layer ids for a widget.

**Usage**

```
l_layer_ids(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

**Details**

For more information run: `l_help("learn_R_layer.html#add-move-delete-layers")`

**Value**

vector with layer ids in rendering order. To create a layer handle object use [l\\_create\\_handle](#).

**See Also**

[l\\_layer](#), [l\\_info\\_states](#)

**Examples**

```
set.seed(500)
x <- rnorm(30)
y <- 4 + 3*x + rnorm(30)
fit <- lm(y~x)
xseq <- seq(min(x)-1, max(x)+1, length.out = 50)
fit_line <- predict(fit, data.frame(x=xseq))
ci <- predict(fit, data.frame(x=xseq),
              interval="confidence", level=0.95)
pi <- predict(fit, data.frame(x=xseq),
              interval="prediction", level=0.95)

p <- l_plot(y~x, color='black', showScales=TRUE, showGuides=TRUE)
gLayer <- l_layer_group(
  p, label="simple linear regression",
  parent="root", index="end"
)
fitLayer <- l_layer_line(
  p, x=range(xseq), y=fit_line, color="#04327F",
  linewidth=4, label="fit", parent=gLayer
)
ciLayer <- l_layer_polygon(
  p,
  x = c(xseq, rev(xseq)),
  y = c(ci[, 'lwr'], rev(ci[, 'upr'])),
  color = "#96BDFE", linecolor="",
  label = "95 % confidence interval",
  parent = gLayer, index='end'
)
piLayer <- l_layer_polygon(
  p,
  x = c(xseq, rev(xseq)),
  y = c(pi[, 'lwr'], rev(pi[, 'upr'])),
  color = "#E2EDFE", linecolor="",
  label = "95 % prediction interval",
  parent = gLayer, index='end'
)

l_info_states(piLayer)
```

---

<code>l_layer_index</code>	<i>Get the order index of a layer among its siblings</i>
----------------------------	--

---

**Description**

The index determines the rendering order of the children layers of a parent. The layer with index=0 is rendered first.

**Usage**

```
l_layer_index(widget, layer)
```

**Arguments**

<code>widget</code>	widget path or layer object of class 'l_layer'
<code>layer</code>	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Details**

Note that the index for layers is 0 based.

**Value**

numeric value

**See Also**

[l\\_layer](#), [l\\_layer\\_move](#)

---

<code>l_layer_isVisible</code>	<i>Return visibility flag of layer</i>
--------------------------------	--

---

**Description**

Hidden or invisible layers are not rendered. This function queries whether a layer is visible/rendered or not.

**Usage**

```
l_layer_isVisible(widget, layer)
```

**Arguments**

<code>widget</code>	widget path or layer object of class 'l_layer'
<code>layer</code>	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

## Details

Visible layers are rendered, invisible ones are not. If any ancestor of a layer is set to be invisible then the layer is not rendered either. The layer visibility flag can be checked with `l_layer_isVisible` and the actual visibility (i.e. are all the ancestors visible too) can be checked with `l_layer_layerVisibility`.

Note that layer visibility is not a state of the layer itself, instead is information that is part of the layer collection (i.e. its parent widget).

## Value

TRUE or FALSE depending whether the layer is visible or not.

## See Also

`l_layer`, `l_layer_show`, `l_layer_hide`, `l_layer_layerVisibility`, `l_layer_groupVisibility`

## Examples

```
p <- l_plot()
l <- l_layer_rectangle(p, x=0:1, y=0:1)
l_layer_isVisible(p, l)
l_layer_hide(p, l)
l_layer_isVisible(p, l)
```

---

`l_layer_layerVisibility`

*Returns logical value for whether layer is actually seen*

---

## Description

Although the visibility flag for a layer might be set to TRUE it won't be rendered as long as its ancestor group layer is set to be invisible. The `l_layer_visibility` returns TRUE if the layer and all its ancestor layers have their visibility flag set to true and the layer is actually rendered.

## Usage

```
l_layer_layerVisibility(widget, layer)
```

## Arguments

<code>widget</code>	widget path or layer object of class 'l_layer'
<code>layer</code>	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Details**

Visible layers are rendered, invisible ones are not. If any ancestor of a layer is set to be invisible then the layer is not rendered either. The layer visibility flag can be checked with `l_layer_isVisible` and the actual visibility (i.e. are all the ancestors visible too) can be checked with `l_layer_layerVisibility`.

Note that layer visibility is not a state of the layer itself, instead is information that is part of the layer collection (i.e. its parent widget).

**Value**

TRUE if the layer and all its ancestor layers have their visibility flag set to true and the layer is actually rendered, otherwise FALSE.

**See Also**

`l_layer`, `l_layer_show`, `l_layer_hide`, `l_layer_isVisible`, `l_layer_groupVisibility`

---

<code>l_layer_line</code>	<i>Layer a line</i>
---------------------------	---------------------

---

**Description**

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

**Usage**

```
l_layer_line(widget, x, y = NULL, color = "black", linewidth = 1,
             dash = "", label = "line", parent = "root", index = 0, ...)
```

**Arguments**

<code>widget</code>	widget path name as a string
<code>x</code>	the coordinates of line. Alternatively, a single plotting structure, function or any <i>R</i> object with a plot method can be provided as <code>x</code> and <code>y</code> are passed on to <code>xy.coords</code>
<code>y</code>	the y coordinates of the line, optional if <code>x</code> is an appropriate structure.
<code>color</code>	color of line
<code>linewidth</code>	linewidth of outline
<code>dash</code>	dash pattern of line, see <a href="https://www.tcl.tk/man/tcl8.6/TkCmd/canvas.htm#M26">https://www.tcl.tk/man/tcl8.6/TkCmd/canvas.htm#M26</a>
<code>label</code>	label used in the layers inspector
<code>parent</code>	group layer
<code>index</code>	of the newly added layer in its parent group
<code>...</code>	additional state initialization arguments, see <code>l_info_states</code>



**Details**

For more information run: `l_help("learn_R_layer")`

**Value**

layer object handle, layer id

**See Also**

[l\\_layer](#), [l\\_info\\_states](#)

**Examples**

```
p <- l_plot()
l <- l_layer_line(p, x=c(1,2,3,4), y=c(1,3,2,4), color='red', linewidth=2)
l_scaleto_world(p)

# object
p <- l_plot()
l <- l_layer_line(p, x=nhtemp)
l_scaleto_layer(l)
```

---

l\_layer\_lines

*Layer a lines*


---

**Description**

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

**Usage**

```
l_layer_lines(widget, x, y, color = "black", linewidth = 1,
  label = "lines", parent = "root", index = 0, ...)
```

**Arguments**

widget	widget path name as a string
x	list with vectors with x coordinates
y	list with vectors with y coordinates
color	color of lines
linewidth	vector with line widths
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

**Details**

For more information run: `l_help("learn_R_layer")`

**Value**

layer object handle, layer id

**See Also**

[l\\_layer](#), [l\\_info\\_states](#)

**Examples**

```
s <- Filter(function(df)nrow(df) > 1, split(UsAndThem, UsAndThem$Country))
sUaT <- Map(function(country){country[order(country$Year),]} , s)
xcoords <- Map(function(x)x$Year, sUaT)
ycoords <- Map(function(x)x$LifeExpectancy, sUaT)
region <- sapply(sUaT, function(x)as.character(x$Geographic.Region[1]))

p <- l_plot(showItemLabels=TRUE)
l <- l_layer_lines(p, xcoords, ycoords, itemLabel=names(sUaT), color=region)
l_scaleto_layer(l)
```

---

`l_layer_lower`

*Switch the layer place with its sibling to the right*

---

**Description**

Change the layers position within its parent layer group by increasing the index of the layer by one if possible. This means that the raised layer will be rendered before (or on below) of its sibling layer to the right.

**Usage**

```
l_layer_lower(widget, layer)
```

**Arguments**

<code>widget</code>	widget path or layer object of class 'l_layer'
<code>layer</code>	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Value**

0 if success otherwise the function throws an error

**See Also**

[l\\_layer](#), [l\\_layer\\_raise](#), [l\\_layer\\_move](#)

**Examples**

```

p <- l_plot()

l1 <- l_layer_rectangle(p, x=0:1, y=0:1)
l2 <- l_layer_oval(p, x=0:1, y=0:1, color='thistle')

l_aspect(p) <- 1

l_layer_lower(p, l2)

```

---

l_layer_move	<i>Move a layer</i>
--------------	---------------------

---

**Description**

The position of a layer in the layer tree determines the rendering order. That is, the non-group layers are rendered in order of a Depth-first traversal of the layer tree. The toplevel group layer is called 'root'.

**Usage**

```
l_layer_move(widget, layer, parent, index = "0")
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used
parent	if parent layer is not specified it is set to the current parent layer of the layer
index	position among its siblings. valid values are 0, 1, 2, ..., 'end'

**Value**

0 if success otherwise the function throws an error

**See Also**

[l\\_layer](#), [l\\_layer\\_printTree](#), [l\\_layer\\_index](#)

**Examples**

```

p <- l_plot()

l <- l_layer_rectangle(p, x=0:1, y=0:1, color="steelblue")
g <- l_layer_group(p)
l_layer_printTree(p)

l_layer_move(l, parent=g)

```

```
l_layer_printTree(p)

l_layer_move(p, 'model', parent=g)
l_layer_printTree(p)
```

---

l_layer_oval	<i>Layer a oval</i>
--------------	---------------------

---

### Description

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

### Usage

```
l_layer_oval(widget, x, y, color = "gray80", linecolor = "black",
  linewidth = 1, label = "oval", parent = "root", index = 0, ...)
```

### Arguments

widget	widget path name as a string
x	x coordinates
y	y coordinates
color	fill color, if empty string "", then the fill is transparent
linecolor	outline color
linewidth	linewidth of outline
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

### Details

For more information run: `l_help("learn_R_layer")`

### Value

layer object handle, layer id

### See Also

[l\\_layer](#), [l\\_info\\_states](#)

### Examples

```
p <- l_plot()
l <- l_layer_oval(p, c(1,5), c(2,12), color='steelblue')
l_configure(p, panX=0, panY=0, deltaX=20, deltaY=20)
```

---

l_layer_points	<i>Layer a points</i>
----------------	-----------------------

---

**Description**

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

Scatter points layer

**Usage**

```
l_layer_points(widget, x, y = NULL, color = "gray60", size = 6,
  label = "points", parent = "root", index = 0, ...)
```

**Arguments**

widget	widget path name as a string
x	the coordinates of line. Alternatively, a single plotting structure, function or any <i>R</i> object with a plot method can be provided as x and y are passed on to <a href="#">xy.coords</a>
y	the y coordinates of the line, optional if x is an appropriate structure.
color	color of points
size	size point, as for scatterplot model layer
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

**Details**

For more information run: `l_help("learn_R_layer")`

**Value**

layer object handle, layer id

**See Also**

[l\\_layer](#), [l\\_info\\_states](#)

---

l_layer_polygon	<i>Layer a polygon</i>
-----------------	------------------------

---

### Description

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

### Usage

```
l_layer_polygon(widget, x, y, color = "gray80", linecolor = "black",
  linewidth = 1, label = "polygon", parent = "root", index = 0,
  ...)
```

### Arguments

widget	widget path name as a string
x	x coordinates
y	y coordinates
color	fill color, if empty string "", then the fill is transparent
linecolor	outline color
linewidth	linewidth of outline
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

### Details

For more information run: `l_help("learn_R_layer")`

### Value

layer object handle, layer id

### See Also

[l\\_layer](#), [l\\_info\\_states](#)

**Examples**

```

set.seed(500)
x <- rnorm(30)
y <- 4 + 3*x + rnorm(30)
fit <- lm(y~x)
xseq <- seq(min(x)-1, max(x)+1, length.out = 50)
fit_line <- predict(fit, data.frame(x=range(xseq)))
ci <- predict(fit, data.frame(x=xseq),
              interval="confidence", level=0.95)
pi <- predict(fit, data.frame(x=xseq),
              interval="prediction", level=0.95)

p <- l_plot(y~x, color='black', showScales=TRUE, showGuides=TRUE)
gLayer <- l_layer_group(
  p, label="simple linear regression",
  parent="root", index="end"
)
fitLayer <- l_layer_line(
  p, x=range(xseq), y=fit_line, color="#04327F",
  linewidth=4, label="fit", parent=gLayer
)
ciLayer <- l_layer_polygon(
  p,
  x = c(xseq, rev(xseq)),
  y = c(ci[, 'lwr'], rev(ci[, 'upr'])),
  color = "#96BDFE", linecolor="",
  label = "95 % confidence interval",
  parent = gLayer, index='end'
)
piLayer <- l_layer_polygon(
  p,
  x = c(xseq, rev(xseq)),
  y = c(pi[, 'lwr'], rev(pi[, 'upr'])),
  color = "#E2EDFE", linecolor="",
  label = "95 % prediction interval",
  parent = gLayer, index='end'
)

l_info_states(piLayer)

```

---

l\_layer\_polygons

*Layer a polygons*


---

**Description**

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

**Usage**

```
l_layer_polygons(widget, x, y, color = "gray80", linecolor = "black",
  linewidth = 1, label = "polygons", parent = "root", index = 0,
  ...)
```

**Arguments**

widget	widget path name as a string
x	list with vectors with x coordinates
y	list with vectors with y coordinates
color	vector with fill colors, if empty string "", then the fill is transparent
linecolor	vector with outline colors
linewidth	vector with line widths
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

**Details**

For more information run: `l_help("learn_R_layer")`

**Value**

layer object handle, layer id

**See Also**

[l\\_layer](#), [l\\_info\\_states](#)

**Examples**

```
p <- l_plot()

l <- l_layer_polygons(
  p,
  x = list(c(1,2,1.5), c(3,4,6,5,2), c(1,3,5,3)),
  y = list(c(1,1,2), c(1,1.5,1,4,2), c(3,5,6,4)),
  color = c('red', 'green', 'blue'),
  linecolor = ""
)
l_scaleto_world(p)

l_info_states(l, "color")
```



---

l_layer_printTree	<i>Print the layer tree</i>
-------------------	-----------------------------

---

**Description**

Prints the layer tree (i.e. the layer ids) to the prompt. Group layers are prefixed with a '+'. The 'root' layer is not listed.

**Usage**

```
l_layer_printTree(widget)
```

**Arguments**

widget            widget path as a string or as an object handle

**Value**

empty string

**See Also**

[l\\_layer](#), [l\\_layer\\_getChildren](#), [l\\_layer\\_getParent](#)

**Examples**

```
p <- l_plot()
l_layer_rectangle(p, x=0:1, y=0:1)
g <- l_layer_group(p)
l_layer_oval(p, x=0:1, y=0:1, parent=g)
l_layer_line(p, x=0:1, y=0:1, parent=g)
l_layer_printTree(p)
```

---

l_layer_promote	<i>Moves the layer up to be a left sibling of its parent</i>
-----------------	--

---

**Description**

Moves the layer down the layer tree (towards the root layer) if the parent layer is not the root layer.

**Usage**

```
l_layer_promote(widget, layer)
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Value**

0 if success otherwise the function throws an error

**Examples**

```
p <- l_plot()

g1 <- l_layer_group(p)
g2 <- l_layer_group(p, parent=g1)
l1 <- l_layer_oval(p, x=0:1, y=0:1, parent=g2)

l_layer_printTree(p)
l_layer_promote(p, l1)
l_layer_printTree(p)
l_layer_promote(p, l1)
l_layer_printTree(p)
```

---

l_layer_raise	<i>Switch the layer place with its sibling to the left</i>
---------------	--

---

**Description**

Change the layers position within its parent layer group by decreasing the index of the layer by one if possible. This means that the raised layer will be rendered after (or on top) of its sibling layer to the left.

**Usage**

```
l_layer_raise(widget, layer)
```

**Arguments**

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

**Value**

0 if success otherwise the function throws an error

**See Also**

[l\\_layer](#), [l\\_layer\\_lower](#), [l\\_layer\\_move](#)

**Examples**

```
p <- l_plot()

l1 <- l_layer_rectangle(p, x=0:1, y=0:1)
l2 <- l_layer_oval(p, x=0:1, y=0:1, color='thistle')

l_aspect(p) <- 1

l_layer_raise(p, l1)
```

---

`l_layer_rasterImage`    *Layer a Raster Image*

---

**Description**

This function is very similar to the [rasterImage](#) function. It works with every loon plot which is based on the cartesian coordinate system.

**Usage**

```
l_layer_rasterImage(widget, image, xleft, ybottom, xright, ytop,
  angle = 0, interpolate = FALSE, parent = "root", index = "end",
  ...)
```

**Arguments**

<code>widget</code>	widget path as a string or as an object handle
<code>image</code>	a raster object, or an object that can be coerced to one by <a href="#">as.raster</a> .
<code>xleft</code>	a vector (or scalar) of left x positions.
<code>ybottom</code>	a vector (or scalar) of bottom y positions.
<code>xright</code>	a vector (or scalar) of right x positions.
<code>ytop</code>	a vector (or scalar) of top y positions.
<code>angle</code>	angle of rotation (in degrees, anti-clockwise from positive x-axis, about the bottom-left corner).
<code>interpolate</code>	a logical vector (or scalar) indicating whether to apply linear interpolation to the image when drawing.
<code>parent</code>	parent widget path
<code>index</code>	position among its siblings. valid values are 0, 1, 2, ..., 'end'
<code>...</code>	arguments forwarded to <a href="#">l_layer_line</a>

**Details**

For more information run: `l_help("learn_R_layer.html#countourlines-heatimage-rasterimage")`

**Value**

layer id of group or rectangles layer

**Examples**

```
plot(1,1, xlim = c(0,1), ylim=c(0,1))
mat <- matrix(c(0,0,0,0, 1,1), ncol=2)
rasterImage(mat, 0,0,1,1, interpolate = FALSE)

p <- l_plot()
l_layer_rasterImage(p, mat, 0,0,1,1)
l_scaleto_world(p)

image <- as.raster(matrix(0:1, ncol = 5, nrow = 3))
p <- l_plot(showScales=TRUE, background="thistle", useLoonInspector=FALSE)
l_layer_rasterImage(p, image, 100, 300, 150, 350, interpolate = FALSE)
l_layer_rasterImage(p, image, 100, 400, 150, 450)
l_layer_rasterImage(p, image, 200, 300, 200 + 10, 300 + 10,
  interpolate = FALSE)
l_scaleto_world(p)

# from examples(rasterImage)

# set up the plot region:
op <- par(bg = "thistle")
plot(c(100, 250), c(300, 450), type = "n", xlab = "", ylab = "")
rasterImage(image, 100, 300, 150, 350, interpolate = FALSE)
rasterImage(image, 100, 400, 150, 450)
rasterImage(image, 200, 300, 200 + 10, 300 + 10,
  interpolate = FALSE)
```

---

`l_layer_rectangle`      *Layer a rectangle*

---

**Description**

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

**Usage**

```
l_layer_rectangle(widget, x, y, color = "gray80", linecolor = "black",
  linewidth = 1, label = "rectangle", parent = "root", index = 0,
  ...)
```

**Arguments**

widget	widget path name as a string
x	x coordinates
y	y coordinates
color	fill color, if empty string "", then the fill is transparent
linecolor	outline color
linewidth	linewidth of outline
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

**Details**

For more information run: `l_help("learn_R_layer")`

**Value**

layer object handle, layer id

**See Also**

[l\\_layer](#), [l\\_info\\_states](#)

**Examples**

```
p <- l_plot()
l <- l_layer_rectangle(p, x=c(2,3), y=c(1,10), color='steelblue')
l_scaleto_layer(l)
```

---

`l_layer_rectangles`      *Layer a rectangles*

---

**Description**

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

**Usage**

```
l_layer_rectangles(widget, x, y, color = "gray80", linecolor = "black",
  linewidth = 1, label = "rectangles", parent = "root", index = 0,
  ...)
```

**Arguments**

widget	widget path name as a string
x	list with vectors with x coordinates
y	list with vectors with y coordinates
color	vector with fill colors, if empty string "", then the fill is transparent
linecolor	vector with outline colors
linewidth	vector with line widths
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

**Details**

For more information run: `l_help("learn_R_layer")`

**Value**

layer object handle, layer id

**See Also**

[l\\_layer](#), [l\\_info\\_states](#)

**Examples**

```
p <- l_plot()

l <- l_layer_rectangles(
  p,
  x = list(c(0,1), c(1,2), c(2,3), c(5,6)),
  y = list(c(0,1), c(1,2), c(0,1), c(3,4)),
  color = c('red', 'blue', 'green', 'orange'),
  linecolor = "black"
)
l_scaleto_world(p)

l_info_states(l)
```

---

l_layer_relabel	<i>Change layer label</i>
-----------------	---------------------------

---

### Description

Layer labels are useful to identify layer in the layer inspector. The layer label can be initially set at layer creation with the label argument.

### Usage

```
l_layer_relabel(widget, layer, label)
```

### Arguments

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used
label	new label of layer

### Details

Note that the layer label is not a state of the layer itself, instead is information that is part of the layer collection (i.e. its parent widget).

### Value

0 if success otherwise the function throws an error

### See Also

[l\\_layer](#), [l\\_layer\\_getLabel](#)

### Examples

```
p <- l_plot()

l <- l_layer_rectangle(p, x=0:1, y=0:1, label="A rectangle")
l_layer_getLabel(p, l)

l_layer_relabel(p, l, label="A relabelled rectangle")
l_layer_getLabel(p, l)
```

---

l_layer_show	<i>Show or unhide a Layer</i>
--------------	-------------------------------

---

### Description

Hidden or invisible layers are not rendered. This function unhides invisible layer so that they are rendered again.

### Usage

```
l_layer_show(widget, layer)
```

### Arguments

widget	widget path or layer object of class 'l_layer'
layer	layer id. If the widget argument is of class 'l_layer' then the layer argument is not used

### Details

Visible layers are rendered, invisible ones are not. If any ancestor of a layer is set to be invisible then the layer is not rendered either. The layer visibility flag can be checked with [l\\_layer\\_isVisible](#) and the actual visibility (i.e. are all the ancestors visible too) can be checked with [l\\_layer\\_layerVisibility](#).

Note that layer visibility is not a state of the layer itself, instead is information that is part of the layer collection (i.e. its parent widget).

### Value

0 if success otherwise the function throws an error

### See Also

[l\\_layer](#), [l\\_layer\\_hide](#), [l\\_layer\\_isVisible](#), [l\\_layer\\_layerVisibility](#), [l\\_layer\\_groupVisibility](#)

### Examples

```
p <- l_plot()

l <- l_layer_rectangle(p, x=0:1, y=0:1, color="steelblue")
l_layer_hide(p, l)

l_layer_show(p, l)
```



---

l_layer_text	<i>Layer a text</i>
--------------	---------------------

---

### Description

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

layer a single character string

### Usage

```
l_layer_text(widget, x, y, text, color = "gray60", size = 6,  
            angle = 0, label = "text", parent = "root", index = 0, ...)
```

### Arguments

widget	widget path name as a string
x	coordinate
y	coordinate
text	character string
color	color of text
size	size of the font
angle	rotation of text
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

### Details

As a side effect of Tcl's text-based design, it is best to use `l_layer_text` if one would like to layer a single character string (and not `l_layer_texts` with `n=1`).

For more information run: `l_help("learn_R_layer")`

### Value

layer object handle, layer id

### See Also

[l\\_layer](#), [l\\_info\\_states](#)

### Examples

```
p <- l_plot()  
l <- l_layer_text(p, 0, 0, "Hello World")
```

---

l_layer_texts	<i>Layer a texts</i>
---------------	----------------------

---

### Description

Loon's displays that are based on Cartesian coordinates (i.e. scatterplot, histogram and graph display) allow for layering visual information including polygons, text and rectangles.

### Usage

```
l_layer_texts(widget, x, y, text, color = "gray60", size = 6,
             angle = 0, label = "texts", parent = "root", index = 0, ...)
```

### Arguments

widget	widget path name as a string
x	the coordinates of line. Alternatively, a single plotting structure, function or any <i>R</i> object with a plot method can be provided as x and y are passed on to <a href="#">xy.coords</a>
y	the y coordinates of the line, optional if x is an appropriate structure.
text	vector with text strings
color	color of line
size	font size
angle	text rotation
label	label used in the layers inspector
parent	group layer
index	of the newly added layer in its parent group
...	additional state initialization arguments, see <a href="#">l_info_states</a>

### Details

For more information run: `l_help("learn_R_layer")`

### Value

layer object handle, layer id

### See Also

[l\\_layer](#), [l\\_info\\_states](#)

### Examples

```
p <- l_plot()
l <- l_layer_texts(p, x=1:3, y=3:1, text=c("This is", "a", "test"), size=20)
l_scaleto_world(p)
```

---

l_loon_inspector	<i>Create a loon inspector</i>
------------------	--------------------------------

---

### Description

The loon inspector is a singleton widget that provides an overview to view and modify the active plot.

### Usage

```
l_loon_inspector(parent = NULL, ...)
```

### Arguments

parent	parent widget path
...	state arguments, see <a href="#">l_info_states</a> .

### Details

For more information run: `l_help("learn_R_display_inspectors")`

### Value

widget handle

### Examples

```
i <- l_loon_inspector()
```

---

l_make_glyphs	<i>Make arbitrary glyphs with R graphic devices</i>
---------------	---

---

### Description

Loon's primitive glyph types are limited in terms of compound shapes. With this function you can create each point glyph as a png and re-import it as a tk img object to be used as point glyphs in loon. See the examples.

### Usage

```
l_make_glyphs(data, draw_fun, width = 50, height = 50, ...)
```

**Arguments**

data	list where each element contains a data object used for the draw_fun
draw_fun	function that draws a glyph using R base graphics or the grid (including ggplot2 and lattice) engine
width	width of each glyph in pixel
height	height of each glyph in pixel
...	additional arguments passed on to the png function

**Value**

vector with tk img object references

**Examples**

```
## Not run:
if (requireNamespace("maps", quietly = TRUE)) {
  data(minority)
  p <- l_plot(minority$long, minority$lat)

  canada <- maps::map("world", "Canada", fill=TRUE, plot=FALSE)
  l_map <- l_layer(p, canada, asSingleLayer=TRUE)
  l_scaleto_world(p)

  img <- l_make_glyphs(lapply(1:nrow(minority), function(i)minority[i,]), function(m) {
    par(mar=c(1,1,1,1)*.5)
    mat <- as.matrix(m[1,1:10]/max(m[1:10]))
    barplot(height = mat,
            beside = FALSE,
            ylim = c(0,1),
            axes= FALSE,
            axisnames=FALSE)
  }, width=120, height=120)

  l_imageviewer(img)

  g <- l_glyph_add_image(p, img, "barplot")
  p['glyph'] <- g
}

## with grid
if (requireNamespace("grid", quietly = TRUE)) {
  li <- l_make_glyphs(runif(6), function(x) {
    if(any(x>1 | x<0))
      stop("out of range")
    grid::pushViewport(grid::plotViewport(grid::unit(c(1,1,1,1)*0, "points")))
    grid::grid.rect(gp=grid::gpar(fill=NA))
    grid::grid.rect(0, 0, height = grid::unit(x, "npc"), just = c("left", "bottom"),
                    gp=grid::gpar(col=NA, fill="steelblue"))
  })
}
```

```

    })

    l_imageviewer(li)

    p <- l_plot(1:6)
    g <- l_glyph_add_image(p, li, "bars")
    p['glyph'] <- g
  }

  ## End(Not run)

```

---

l\_move\_grid

*Arrange Points or Nodes on a Grid*


---

### Description

Scatterplot and graph displays support interactive temporary relocation of single points (nodes for graphs).

### Usage

```
l_move_grid(widget, which = "selected")
```

### Arguments

widget	plot or graph widget handle or widget path name
which	either one of 'selected', 'active', 'all', or a boolean vector with a value for each point.

### Details

Moving the points temporarily saves the new point coordinates to the states `xTemp` and `yTemp`. The dimension of `xTemp` and `yTemp` is either 0 or `n`. If `xTemp` or `yTemp` are not of length 0 then they are required to be of length `n`, and the scatterplot will display those coordinates instead of the coordinates in `x` or `y`.

Note that the points can also be temporally relocated using mouse and keyboard gestures. That is, to move a single point or node press the CTRL key while dragging a the point. To move the selected points press down the CTRL and Shift keys while dragging one of the selected points.

When distributing points horizontally or vertically, their order remains the same. When distributing points horizontally or vertically, their order remains the same. For example, when you distribute the point both horizontally and vertically, then the resulting scatterplot will be a plot of the `y` ranks versus the `x` ranks. The correlation on that plot will be Spearman's rho. When arranging points on a grid, some of the spatial ordering is preserved by first determining a grid size (i.e. `a x b` where `a` and `b` are the same or close numbers) and then by taking the `a` smallest values in the `y` direction and arrange them by their `x` order in the first row, then repeat for the remaining points.

Also note the the loon inspector also has buttons for these temporary points/nodes movements.

**See Also**

[l\\_move\\_valign](#), [l\\_move\\_halign](#), [l\\_move\\_vdist](#), [l\\_move\\_hdist](#), [l\\_move\\_grid](#), [l\\_move\\_jitter](#), [l\\_move\\_reset](#)

---

l\_move\_halign

*Horizontally Align Points or Nodes*


---

**Description**

Scatterplot and graph displays support interactive temporary relocation of single points (nodes for graphs).

**Usage**

```
l_move_halign(widget, which = "selected")
```

**Arguments**

widget	plot or graph widget handle or widget path name
which	either one of 'selected', 'active', 'all', or a boolean vector with a value for each point.

**Details**

Moving the points temporarily saves the new point coordinates to the states `xTemp` and `yTemp`. The dimension of `xTemp` and `yTemp` is either 0 or `n`. If `xTemp` or `yTemp` are not of length 0 then they are required to be of length `n`, and the scatterplot will display those coordinates instead of the coordinates in `x` or `y`.

Note that the points can also be temporally relocated using mouse and keyboard gestures. That is, to move a single point or node press the CTRL key while dragging a the point. To move the selected points press down the CTRL and Shift keys while dragging one of the selected points.

When distributing points horizontally or vertically, their order remains the same. When distributing points horizontally or vertically, their order remains the same. For example, when you distribute the point both horizontally and vertically, then the resulting scatterplot will be a plot of the `y` ranks versus the `x` ranks. The correlation on that plot will be Spearman's rho. When arranging points on a grid, some of the spatial ordering is preserved by first determining a grid size (i.e. `a x b` where `a` and `b` are the same or close numbers) and then by taking the `a` smallest values in the `y` direction and arrange them by their `x` order in the first row, then repeat for the remaining points.

Also note the the loon inspector also has buttons for these temporary points/nodes movements.

**See Also**

[l\\_move\\_valign](#), [l\\_move\\_halign](#), [l\\_move\\_vdist](#), [l\\_move\\_hdist](#), [l\\_move\\_grid](#), [l\\_move\\_jitter](#), [l\\_move\\_reset](#)

---

`l_move_hdist`*Horizontally Distribute Points or Nodes*

---

**Description**

Scatterplot and graph displays support interactive temporary relocation of single points (nodes for graphs).

**Usage**

```
l_move_hdist(widget, which = "selected")
```

**Arguments**

<code>widget</code>	plot or graph widget handle or widget path name
<code>which</code>	either one of 'selected', 'active', 'all', or a boolean vector with a value for each point.

**Details**

Moving the points temporarily saves the new point coordinates to the states `xTemp` and `yTemp`. The dimension of `xTemp` and `yTemp` is either 0 or `n`. If `xTemp` or `yTemp` are not of length 0 then they are required to be of length `n`, and the scatterplot will display those coordinates instead of the coordinates in `x` or `y`.

Note that the points can also be temporally relocated using mouse and keyboard gestures. That is, to move a single point or node press the CTRL key while dragging a the point. To move the selected points press down the CTRL and Shift keys while dragging one of the selected points.

When distributing points horizontally or vertically, their order remains the same. When distributing points horizontally or vertically, their order remains the same. For example, when you distribute the point both horizontally and vertically, then the resulting scatterplot will be a plot of the `y` ranks versus the `x` ranks. The correlation on that plot will be Spearman's rho. When arranging points on a grid, some of the spatial ordering is preserved by first determining a grid size (i.e. `a x b` where `a` and `b` are the same or close numbers) and then by taking the `a` smallest values in the `y` direction and arrange them by their `x` order in the first row, then repeat for the remaining points.

Also note the the loon inspector also has buttons for these temporary points/nodes movements.

**See Also**

[l\\_move\\_valign](#), [l\\_move\\_halign](#), [l\\_move\\_vdist](#), [l\\_move\\_hdist](#), [l\\_move\\_grid](#), [l\\_move\\_jitter](#), [l\\_move\\_reset](#)

---

l_move_jitter	<i>Jitter Points Or Nodes</i>
---------------	-------------------------------

---

### Description

Scatterplot and graph displays support interactive temporary relocation of single points (nodes for graphs).

### Usage

```
l_move_jitter(widget, which = "selected", factor = 1, amount = "")
```

### Arguments

widget	plot or graph widget handle or widget path name
which	either one of 'selected', 'active', 'all', or a boolean vector with a value for each point.
factor	numeric.
amount	numeric; if positive, used as <i>amount</i> (see below), otherwise, if = 0 the default is factor * z/50. Default (NULL): factor * d/5 where d is about the smallest difference between x values.

### Details

Moving the points temporarily saves the new point coordinates to the states xTemp and yTemp. The dimension of xTemp and yTemp is either 0 or n. If xTemp or yTemp are not of length 0 then they are required to be of length n, and the scatterplot will display those coordinates instead of the coordinates in x or y.

Note that the points can also be temporally relocated using mouse and keyboard gestures. That is, to move a single point or node press the CTRL key while dragging a the point. To move the selected points press down the CTRL and Shift keys while dragging one of the selected points.

When distributing points horizontally or vertically, their order remains the same. When distributing points horizontally or vertically, their order remains the same. For example, when you distribute the point both horizontally and vertically, then the resulting scatterplot will be a plot of the y ranks versus the x ranks. The correlation on that plot will be Spearman's rho. When arranging points on a grid, some of the spatial ordering is preserved by first determining a grid size (i.e. a x b where a and b are the same or close numbers) and then by taking the a smallest values in the y direction and arrange them by their x order in the first row, then repeat for the remaining points.

Also note the the loon inspector also has buttons for these temporary points/nodes movements.

### See Also

[l\\_move\\_valign](#), [l\\_move\\_halign](#), [l\\_move\\_vdist](#), [l\\_move\\_hdist](#), [l\\_move\\_grid](#), [l\\_move\\_jitter](#), [l\\_move\\_reset](#)



---

`l_move_reset`*Reset Temporary Point or Node Locations to the x and y states*

---

### Description

Scatterplot and graph displays support interactive temporary relocation of single points (nodes for graphs).

### Usage

```
l_move_reset(widget, which = "selected")
```

### Arguments

<code>widget</code>	plot or graph widget handle or widget path name
<code>which</code>	either one of 'selected', 'active', 'all', or a boolean vector with a value for each point.

### Details

Moving the points temporarily saves the new point coordinates to the states `xTemp` and `yTemp`. The dimension of `xTemp` and `yTemp` is either 0 or `n`. If `xTemp` or `yTemp` are not of length 0 then they are required to be of length `n`, and the scatterplot will display those coordinates instead of the coordinates in `x` or `y`.

Note that the points can also be temporally relocated using mouse and keyboard gestures. That is, to move a single point or node press the CTRL key while dragging a the point. To move the selected points press down the CTRL and Shift keys while dragging one of the selected points.

When distributing points horizontally or vertically, their order remains the same. When distributing points horizontally or vertically, their order remains the same. For example, when you distribute the point both horizontally and vertically, then the resulting scatterplot will be a plot of the `y` ranks versus the `x` ranks. The correlation on that plot will be Spearman's rho. When arranging points on a grid, some of the spatial ordering is preserved by first determining a grid size (i.e. `a x b` where `a` and `b` are the same or close numbers) and then by taking the `a` smallest values in the `y` direction and arrange them by their `x` order in the first row, then repeat for the remaining points.

Also note the the loon inspector also has buttons for these temporary points/nodes movements.

### See Also

[l\\_move\\_valign](#), [l\\_move\\_halign](#), [l\\_move\\_vdist](#), [l\\_move\\_hdist](#), [l\\_move\\_grid](#), [l\\_move\\_jitter](#), [l\\_move\\_reset](#)

---

l_move_valign	<i>Vertically Align Points or Nodes</i>
---------------	---

---

### Description

Scatterplot and graph displays support interactive temporary relocation of single points (nodes for graphs).

### Usage

```
l_move_valign(widget, which = "selected")
```

### Arguments

widget	plot or graph widget handle or widget path name
which	either one of 'selected', 'active', 'all', or a boolean vector with a value for each point.

### Details

Moving the points temporarily saves the new point coordinates to the states `xTemp` and `yTemp`. The dimension of `xTemp` and `yTemp` is either 0 or `n`. If `xTemp` or `yTemp` are not of length 0 then they are required to be of length `n`, and the scatterplot will display those coordinates instead of the coordinates in `x` or `y`.

Note that the points can also be temporally relocated using mouse and keyboard gestures. That is, to move a single point or node press the CTRL key while dragging a the point. To move the selected points press down the CTRL and Shift keys while dragging one of the selected points.

When distributing points horizontally or vertically, their order remains the same. When distributing points horizontally or vertically, their order remains the same. For example, when you distribute the point both horizontally and vertically, then the resulting scatterplot will be a plot of the `y` ranks versus the `x` ranks. The correlation on that plot will be Spearman's rho. When arranging points on a grid, some of the spatial ordering is preserved by first determining a grid size (i.e. `a x b` where `a` and `b` are the same or close numbers) and then by taking the `a` smallest values in the `y` direction and arrange them by their `x` order in the first row, then repeat for the remaining points.

Also note the the loon inspector also has buttons for these temporary points/nodes movements.

### See Also

[l\\_move\\_valign](#), [l\\_move\\_halign](#), [l\\_move\\_vdist](#), [l\\_move\\_hdist](#), [l\\_move\\_grid](#), [l\\_move\\_jitter](#), [l\\_move\\_reset](#)

---

`l_move_vdist`*Vertically Distribute Points or Nodes*

---

**Description**

Scatterplot and graph displays support interactive temporary relocation of single points (nodes for graphs).

**Usage**

```
l_move_vdist(widget, which = "selected")
```

**Arguments**

<code>widget</code>	plot or graph widget handle or widget path name
<code>which</code>	either one of 'selected', 'active', 'all', or a boolean vector with a value for each point.

**Details**

Moving the points temporarily saves the new point coordinates to the states `xTemp` and `yTemp`. The dimension of `xTemp` and `yTemp` is either 0 or `n`. If `xTemp` or `yTemp` are not of length 0 then they are required to be of length `n`, and the scatterplot will display those coordinates instead of the coordinates in `x` or `y`.

Note that the points can also be temporally relocated using mouse and keyboard gestures. That is, to move a single point or node press the CTRL key while dragging a the point. To move the selected points press down the CTRL and Shift keys while dragging one of the selected points.

When distributing points horizontally or vertically, their order remains the same. When distributing points horizontally or vertically, their order remains the same. For example, when you distribute the point both horizontally and vertically, then the resulting scatterplot will be a plot of the `y` ranks versus the `x` ranks. The correlation on that plot will be Spearman's rho. When arranging points on a grid, some of the spatial ordering is preserved by first determining a grid size (i.e. `a x b` where `a` and `b` are the same or close numbers) and then by taking the `a` smallest values in the `y` direction and arrange them by their `x` order in the first row, then repeat for the remaining points.

Also note the the loon inspector also has buttons for these temporary points/nodes movements.

**See Also**

[l\\_move\\_valign](#), [l\\_move\\_halign](#), [l\\_move\\_vdist](#), [l\\_move\\_hdist](#), [l\\_move\\_grid](#), [l\\_move\\_jitter](#), [l\\_move\\_reset](#)

---

l_navgraph	<i>Explore a dataset with the canonical 2d navigation graph setting</i>
------------	---

---

**Description**

Creates a navigation graph, a graphswitch, a navigator and a geodesic2d context added, and a scatterplot.

**Usage**

```
l_navgraph(data, separator = ":", graph = NULL, ...)
```

**Arguments**

data	a data.frame with numeric variables only
separator	string the separates variable names in 2d graph nodes
graph	optional, graph or loongraph object with navigation graph. If the graph argument is not used then a 3d and 4d transition graph and a complete transition graph is added.
...	arguments passed on to modify the scatterplot plot states

**Details**

For more information run: `l_help("learn_R_display_graph.html#l_navgraph")`

**Value**

named list with graph handle, plot handle, graphswitch handle, navigator handle, and context handle.

**Examples**

```
ng <- l_navgraph(oliveAcids, color=olive$Area)
ng2 <- l_navgraph(oliveAcids, separator='-', color=olive$Area)
```

---

l_navigator_add	<i>Add a Navigator to a Graph</i>
-----------------	-----------------------------------

---

**Description**

To turn a graph into a navigation graph you need to add one or more navigators. Navigator have their own set of states that can be queried and modified.

**Usage**

```
l_navigator_add(widget, from = "", to = "", proportion = 0,
               color = "orange", ...)
```

**Arguments**

widget	graph widget
from	The position of the navigator on the graph is defined by the states from, to and proportion. The states from and to hold vectors of node names of the graph. The proportion state is a number between and including 0 and 1 and defines how far the navigator is between the last element of from and the first element of to. The to state can also be an empty string '' if there is no further node to go to. Hence, the concatenation of from and to define a path on the graph.
to	see descriptoin above for from
proportion	see descriptoin above for from
color	of navigator
...	named arguments passed on to modify navigator states

**Details**

For more information run: `l_help("learn_R_display_graph.html#navigators")`

**Value**

navigator handle with navigator id

**See Also**

[l\\_navigator\\_delete](#), [l\\_navigator\\_ids](#), [l\\_navigator\\_walk\\_path](#), [l\\_navigator\\_walk\\_forward](#), [l\\_navigator\\_walk\\_backward](#), [l\\_navigator\\_relabel](#), [l\\_navigator\\_getLabel](#)

---

`l_navigator_delete`     *Delete a Navigator*

---

**Description**

Removes a navigator from a graph widget

**Usage**

```
l_navigator_delete(widget, id)
```

**Arguments**

widget	graph widget
id	navigator handle or navigator id

**See Also**[l\\_navigator\\_add](#)

---

`l_navigator_getLabel`    *Query the Label of a Navigator*

---

**Description**

Returns the label of a navigator

**Usage**

```
l_navigator_getLabel(widget, id)
```

**Arguments**

widget	graph widget handle
id	navigator id

**See Also**[l\\_navigator\\_add](#)

---

`l_navigator_getPath`    *Get the sequence of nodes of a navigator's current path*

---

**Description**

Determines and returns the current path of the navigator.

**Usage**

```
l_navigator_getPath(navigator)
```

**Arguments**

navigator	navigator handle
-----------	------------------

**Value**

a vector of node names for the current path of the navigator

---

l_navigator_ids	<i>List Navigators</i>
-----------------	------------------------

---

**Description**

Lists all navigators that belong to a graph

**Usage**

```
l_navigator_ids(widget)
```

**Arguments**

widget	graph widget
--------	--------------

**See Also**

[l\\_navigator\\_add](#)

---

l_navigator_relabel	<i>Modify the Label of a Navigator</i>
---------------------	--

---

**Description**

Change the navigator label

**Usage**

```
l_navigator_relabel(widget, id, label)
```

**Arguments**

widget	graph widget handle
id	navigator id
label	new label of navigator

**See Also**

[l\\_navigator\\_add](#)

---

`l_navigator_walk_backward`*Have the Navigator Walk Backward on the Current Path*

---

**Description**

Animate a navigator by having it walk on a path on the graph

**Usage**

```
l_navigator_walk_backward(navigator, to = "")
```

**Arguments**

<code>navigator</code>	navigator handle
<code>to</code>	node name that is part of the active path backward where the navigator should stop.

**Details**

Note that navigators have the states `animationPause` and `animationProportionIncrement` to control the animation speed. Further, you can stop the animation when clicking somewhere on the graph display or by using the mouse scroll wheel.

**See Also**

[l\\_navigator\\_add](#)

---

`l_navigator_walk_forward`*Have the Navigator Walk Forward on the Current Path*

---

**Description**

Animate a navigator by having it walk on a path on the graph

**Usage**

```
l_navigator_walk_forward(navigator, to = "")
```

**Arguments**

<code>navigator</code>	navigator handle
<code>to</code>	node name that is part of the active path forward where the navigator should stop.



**Details**

Note that navigators have the states `animationPause` and `animationProportionIncrement` to control the animation speed. Further, you can stop the animation when clicking somewhere on the graph display or by using the mouse scroll wheel.

**See Also**

[l\\_navigator\\_add](#)

---

`l_navigator_walk_path` *Have the Navigator Walk a Path on the Graph*

---

**Description**

Animate a navigator by having it walk on a path on the graph

**Usage**

```
l_navigator_walk_path(navigator, path)
```

**Arguments**

<code>navigator</code>	navigator handle
<code>path</code>	vector with node names of the host graph that form a valid path on that graph

**See Also**

[l\\_navigator\\_add](#)

---

`l_nestedTclList2Rlist` *Convert a Nested Tcl List to an R List*

---

**Description**

Helper function to work with R and Tcl

**Usage**

```
l_nestedTclList2Rlist(tclobj, transform = function(x) { as.numeric(x)
  })
```

**Arguments**

<code>tclobj</code>	a tcl object as returned by <code>tcl</code> and <code>.Tcl</code>
<code>transform</code>	a function to transform the string output to another data type

**Value**

a nested R list

**See Also**

[l\\_Rlist2nestedTclList](#)

**Examples**

```
tclobj <- .Tcl('set a {{1 2 3} {2 3 4 4} {3 5 3 3}}')
l_nestedTclList2Rlist(tclobj)
```

---

l_ng_plots	<i>2d navigation graph setup with with dynamic node filtering using a scatterplot matrix</i>
------------	--

---

**Description**

Generic function to create a navigation graph environment where user can filter graph nodes by selecting 2d spaces based on 2d measures displayed in a scatterplot matrix.

**Usage**

```
l_ng_plots(measures, ...)
```

**Arguments**

measures	object with measures are stored
...	argument passed on to methods

**Details**

For more information run: `l_help("learn_R_display_graph.html#l_ng_plots")`

**See Also**

[l\\_ng\\_plots.default](#), [l\\_ng\\_plots.measures](#), [l\\_ng\\_plots.scagnostics](#), [measures1d](#), [measures2d](#), [scagnostics2d](#), [l\\_ng\\_ranges](#)

---

l_ng_plots.default	<i>Select 2d spaces with variable associated measures displayed in scatterplot matrix</i>
--------------------	---

---

### Description

Measures object is a matrix or data.frame with measures (columns) for variable pairs (rows) and rownames of the two variates separated by separator

### Usage

```
## Default S3 method:
l_ng_plots(measures, data, separator = ":", ...)
```

### Arguments

measures	matrix or data.frame with measures (columns) for variable pairs (rows) and rownames of the two variates separated by separator
data	data frame for scatterplot
separator	a string that separates the variable pair string into the individual variables
...	arguments passed on to configure the scatterplot

### Details

For more information run: `l_help("learn_R_display_graph.html#l_ng_plots")`

### Value

named list with plots-, graph-, plot-, navigator-, and context handle. The list also contains the environment of the the function call in env.

### See Also

[l\\_ng\\_plots](#), [l\\_ng\\_plots.measures](#), [l\\_ng\\_plots.scagnostics](#), [measures1d](#), [measures2d](#), [scagnostics2d](#), [l\\_ng\\_ranges](#)

### Examples

```
## Not run:
n <- 100
dat <- data.frame(
  A = rnorm(n), B = rnorm(n), C = rnorm(n),
  D = rnorm(n), E = rnorm(n)
)
m2d <- data.frame(
  cov = with(dat, c(cov(A,B), cov(A,C), cov(B,D), cov(D,E), cov(A,E))),
  measure_1 = c(1, 3, 2, 1, 4),
```

```

    row.names = c('A:B', 'A:C', 'B:D', 'D:E', 'A:E')
  )

# or m2d <- as.matrix(m2d)

nav <- l_ng_plots(measures=m2d, data=dat)

# only one measure
m <- m2d[,1]
names(m) <- row.names(m2d)
nav <- l_ng_plots(measures=m, data=dat)

m2d[c(1,2),1]

# one d measures
m1d <- data.frame(
  mean = sapply(dat, mean),
  median = sapply(dat, median),
  sd = sapply(dat, sd),
  q1 = sapply(dat, function(x)quantile(x, probs=0.25)),
  q3 = sapply(dat, function(x)quantile(x, probs=0.75)),
  row.names = names(dat)
)

nav <- l_ng_plots(m1d, dat)

## more involved
q1 <- function(x)as.vector(quantile(x, probs=0.25))

# be careful that the vector names are correct
nav <- l_ng_plots(sapply(oliveAcids, q1), oliveAcids)

## End(Not run)

```

---

`l_ng_plots.measures`    *2d Navigation Graph Setup with dynamic node filtering using a scatterplot matrix*

---

### Description

Measures object is of class `measures`. When using measure objects then the measures can be dynamically re-calculated for a subset of the data.

### Usage

```

## S3 method for class 'measures'
l_ng_plots(measures, ...)

```

## Arguments

measures      object of class measures, see [measures1d](#), [measures2d](#).  
...            arguments passed on to configure the scatterplot

## Details

Note that we provide the [scagnostics2d](#) function to create a measures object for the scagnostics measures.

For more information run: `l_help("learn_R_display_graph.html#l_ng_plots")`

## Value

named list with plots-, graph-, plot-, navigator-, and context handle. The list also contains the environment of the the function call in env.

## See Also

[measures1d](#), [measures2d](#), [scagnostics2d](#), [l\\_ng\\_plots](#), [l\\_ng\\_ranges](#)

## Examples

```
## Not run:
# 2d measures
scags <- scagnostics2d(oliveAcids, separator='**')
scags()
ng <- l_ng_plots(scags, color=olive$Area)

# 1d measures
scale01 <- function(x){(x-min(x))/diff(range(x))}
m1d <- measures1d(sapply(iris[,-5], scale01),
  mean=mean, median=median, sd=sd,
  q1=function(x)as.vector(quantile(x, probs=0.25)),
  q3=function(x)as.vector(quantile(x, probs=0.75)))

m1d()

nav <- l_ng_plots(m1d, color=iris$Species)

# with only one measure
nav <- l_ng_plots(measures1d(oliveAcids, sd))

# with two measures
nav <- l_ng_plots(measures1d(oliveAcids, sd=sd, mean=mean))

## End(Not run)
```

---

`l_ng_plots.scagnostics`

*2d Navigation Graph Setup with dynamic node filtering based on scagnostic measures and by using a scatterplot matrix*

---

## Description

This method is useful when working with objects from the `scagnostics` function from the `scagnostics` R package. In order to dynamically re-calculate the scagnostic measures for a subset of the data use the `scagnostics2d` measures creature function.

## Usage

```
## S3 method for class 'scagnostics'  
l_ng_plots(measures, data, separator = ":", ...)
```

## Arguments

<code>measures</code>	objects from the <code>scagnostics</code> function from the <code>scagnostics</code> R package
<code>data</code>	data frame for scatterplot
<code>separator</code>	a string that separates the variable pair string into the individual variables
<code>...</code>	arguments passed on to configure the scatterplot

## Value

named list with `plots-`, `graph-`, `plot-`, `navigator-`, and `context handle`. The list also contains the environment of the the function call in `env`.

## See Also

[l\\_ng\\_plots](#), [l\\_ng\\_plots.default](#), [l\\_ng\\_plots.measures](#), [measures1d](#), [measures2d](#), [scagnostics2d](#), [l\\_ng\\_ranges](#)

## Examples

```
## Not run:  
library(scagnostics)  
scags <- scagnostics::scagnostics(oliveAcids)  
l_ng_plots(scags, oliveAcids, color=olive$Area)  
  
## End(Not run)
```

---

l_ng_ranges	<i>2d navigation graph setup with with dynamic node fitering using a slider</i>
-------------	---

---

**Description**

Generic function to create a navigation graph environment where user can filter graph nodes using as slider to select 2d spaces based on 2d measures.

**Usage**

```
l_ng_ranges(measures, ...)
```

**Arguments**

measures	object with measures are stored
...	argument passed on to methods

**Details**

For more information run: `l_help("learn_R_display_graph.html#l_ng_ranges")`

**See Also**

[l\\_ng\\_ranges.default](#), [l\\_ng\\_ranges.measures](#), [l\\_ng\\_ranges.scagnostics](#), [measures1d](#), [measures2d](#), [scagnostics2d](#), [l\\_ng\\_ranges](#)

---

l_ng_ranges.default	<i>Select 2d spaces with variable associated measures using a slider</i>
---------------------	--

---

**Description**

Measures object is a matrix or data.frame with measures (columns) for variable pairs (rows) and rownames of the two variates separated by separator

**Usage**

```
## Default S3 method:
l_ng_ranges(measures, data, separator = ":", ...)
```

**Arguments**

measures	matrix or data.frame with measures (columns) for variable pairs (rows) and rownames of the two variates separated by separator
data	data frame for scatterplot
separator	a string that separates the variable pair string into the individual variables
...	arguments passed on to configure the scatterplot

**Details**

For more information run: `l_help("learn_R_display_graph.html#l_ng_ranges")`

**Value**

named list with plots-, graph-, plot-, navigator-, and context handle. The list also contains the environment of the the function call in env.

**See Also**

[l\\_ng\\_ranges](#), [l\\_ng\\_ranges.measures](#), [l\\_ng\\_ranges.scagnostics](#), [measures1d](#), [measures2d](#), [scagnostics2d](#), [l\\_ng\\_ranges](#)

**Examples**

```
# Simple example with generated data
n <- 100
dat <- data.frame(
  A = rnorm(n), B = rnorm(n), C = rnorm(n),
  D = rnorm(n), E = rnorm(n)
)
m2d <- data.frame(
  cor = with(dat, c(cor(A,B), cor(A,C), cor(B,D), cor(D,E), cor(A,E))),
  my_measure = c(1, 3, 2, 1, 4),
  row.names = c('A:B', 'A:C', 'B:D', 'D:E', 'A:E')
)

# or m2d <- as.matrix(m2d)

nav <- l_ng_ranges(measures=m2d, data=dat)

# With 1d measures
m1d <- data.frame(
  mean = sapply(dat, mean),
  median = sapply(dat, median),
  sd = sapply(dat, sd),
  q1 = sapply(dat, function(x)quantile(x, probs=0.25)),
  q3 = sapply(dat, function(x)quantile(x, probs=0.75)),
  row.names = names(dat)
)

nav <- l_ng_ranges(m1d, dat)
```

---

`l_ng_ranges.measures`    *2d Navigation Graph Setup with dynamic node filtering using a slider*

---

**Description**

Measures object is of class measures. When using measure objects then the measures can be dynamically re-calculated for a subset of the data.



**Usage**

```
## S3 method for class 'measures'
l_ng_ranges(measures, ...)
```

**Arguments**

measures            object of class measures, see [measures1d](#), [measures2d](#).

...                 arguments passed on to configure the scatterplot

**Details**

Note that we provide the [scagnostics2d](#) function to create a measures object for the scagnostics measures.

For more information run: `l_help("learn_R_display_graph.html#l_ng_ranges")`

**Value**

named list with plots-, graph-, plot-, navigator-, and context handle. The list also contains the environment of the the function call in env.

**See Also**

[measures1d](#), [measures2d](#), [scagnostics2d](#), [l\\_ng\\_ranges](#), [l\\_ng\\_plots](#)

**Examples**

```
# 2d measures
# s <- scagnostics2d(oliveAcids)
# nav <- l_ng_ranges(s, color=olive$Area)

# 1d measures
scale01 <- function(x){(x-min(x))/diff(range(x))}
m1d <- measures1d(sapply(iris[,-5], scale01),
  mean=mean, median=median, sd=sd,
  q1=function(x)as.vector(quantile(x, probs=0.25)),
  q3=function(x)as.vector(quantile(x, probs=0.75)))

m1d()

nav <- l_ng_ranges(m1d, color=iris$Species)
```

---

l\_ng\_ranges.scagnostics

*2d Navigation Graph Setup with dynamic node filtering based on scagnostic measures and using a slider*

---

## Description

This method is useful when working with objects from the [scagnostics](#) function from the scagnostics R package. In order to dynamically re-calculate the scagnostic measures for a subset of the data use the [scagnostics2d](#) measures creature function.

## Usage

```
## S3 method for class 'scagnostics'
l_ng_ranges(measures, data, separator = ":", ...)
```

## Arguments

measures	objects from the <a href="#">scagnostics</a> function from the scagnostics R package
data	data frame for scatterplot
separator	a string that separates the variable pair string into the individual variables
...	arguments passed on to configure the scatterplot

## Details

For more information run: `l_help("learn_R_display_graph.html#l_ng_ranges")`

## Value

named list with plots-, graph-, plot-, navigator-, and context handle. The list also contains the environment of the the function call in env.

## See Also

[l\\_ng\\_ranges](#), [l\\_ng\\_ranges.default](#), [l\\_ng\\_ranges.measures](#), [measures1d](#), [measures2d](#), [scagnostics2d](#), [l\\_ng\\_ranges](#)

## Examples

```
## Not run:
if (requireNamespace("scagnostics", quietly = TRUE)) {
  s <- scagnostics::scagnostics(oliveAcids)
  ng <- l_ng_ranges(s, oliveAcids, color=olive$Area)
}

## End(Not run)
```

---

l_pairs	<i>An interactive scatterplot matrix</i>
---------	--

---

### Description

Function creates a scatterplot matrix using loon's scatterplot widgets

### Usage

```
l_pairs(data, linkingGroup, linkingKey, showItemLabels = TRUE, itemLabel,
        showHistograms = FALSE, histLocation = c("edge", "diag"),
        histHeightProp = 1, histArgs = list(), showSerialAxes = FALSE,
        serialAxesArgs = list(), parent = NULL, ...)
```

### Arguments

data	a data.frame with numerical data to create the scatterplot matrix
linkingGroup	string giving the linkingGroup for all plots. If missing, a default linkingGroup will be determined from parsing the data.
linkingKey	a vector of strings to provide a linking identity for each row of the data data.frame. If missing, a default linkingKey will be $\emptyset$ : $(nrows(data)-1)$ .
showItemLabels	TRUE, logical indicating whether its itemLabel pops up over a point when the mouse hovers over it.
itemLabel	a vector of strings to be used as pop up information when the mouse hovers over a point. If missing, the default itemLabel will be the row.names(data).
showHistograms	logical (default FALSE) to show histograms of each variable or not
histLocation	one "edge" or "diag", when showHistograms = TRUE
histHeightProp	a positive number giving the height of the histograms as a proportion of the height of the scatterplots
histArgs	additional arguments to modify the 'l_hist' states
showSerialAxes	logical (default FALSE) indication of whether to show a serial axes plot in the bottom left of the pairs plot (or not)
serialAxesArgs	additional arguments to modify the 'l_serialaxes' states
parent	parent widget path
...	named arguments to modify the 'l_plot' states of the scatterplots

### Value

an 'l\_pairs' object (an 'l\_compound' object), being a list with named elements, each representing a separate interactive plot. The names of the plots should be self explanatory and a list of all plots can be accessed from the 'l\_pairs' object via 'l\_getPlots()'. All plots are linked by default (name taken from data set if not provided). Panning and zooming are constrained to work together within the scatterplot matrix (and histograms).

**See Also**

[l\\_plot](#) and [l\\_getPlots](#)

**Examples**

```
p <- l_pairs(iris[,-5], color=iris$Species, linkingGroup = "iris")

p <- l_pairs(iris[,-5], color=iris$Species, linkingGroup = "iris",
            showHistograms = TRUE, showSerialAxes = TRUE)

# plot names
names(p)

# Each plot must be accessed to make changes not managed through
# linking.
# E.g. to change the glyph on all scatterplots to open circles
for (plot in l_getPlots(p)) {
  if (is(plot, "l_plot")) {
    plot["glyph"] <- "ocircle"}
}
```

---

l\_plot

---

*Create an interactive loon plot widget*


---

**Description**

`l_plot` is a generic function for creating interactive visualization environments for R objects.

**Usage**

```
l_plot(x, y, ...)
```

**Arguments**

`x` the coordinates of points in the `l_plot`. Alternatively, a single plotting structure, function, or any R object having an `l_plot` method can be provided.

`y` the y coordinates of points in the `l_plot`, optional if `x` is an appropriate structure.

`...` named arguments to modify plot states. See [l\\_info\\_states](#) of any instantiated `l_plot` for examples of names and values.

**Details**

Like `plot` in R, `l_plot` is the generic plotting function for objects in loon. The default method `l_plot.default` produces the interactive scatterplot in loon. This is the workhorse of ‘loon’ and is often a key part of many other displays (e.g. [l\\_pairs](#) and [l\\_navgraph](#)).

For example, the methods include `l_plot.default` (the basic interactive scatterplot), `l_plot.density` (layers output of `density` in an empty scatterplot), `l_plot.map` (layers a map in an empty scatterplot), and `l_plot.stl` (a compound display of the output of `stl`).

A complete list is had from `methods(l_plot)`.

To get started with loon it is recommended to follow the introductory loon vignette `vignette(topic = "introduction", package = "loon")` and to explore loon's website accessible via `l_help()`.

### Value

widget handle

### See Also

[l\\_plot\\_arguments](#) and [l\\_plot.default](#).

Other two-dimensional plotting functions: [l\\_plot.decomposed.ts](#), [l\\_plot.default](#), [l\\_plot.density](#), [l\\_plot.stl](#)

Other loon interactive states: [l\\_info\\_states](#), [l\\_plot.default](#), [l\\_state\\_names](#), [names.loon](#)

### Examples

```
# ordinary use
p <- with(iris, l_plot(Sepal.Width, Petal.Length, color=Species))

versi <- iris$Species == "versicolor"
p["glyph"][versi] <- "ctriangle"

# Get an R (grid) graphics plot of the current loon plot
plot(p)
# or with more control about grid parameters
grid.loon(p)
# or to save the grid data structure (grob) for later use
pg <- loonGrob(p)

# plot a density estimate
set.seed(314159)
ds <- density(rnorm(1000))
p <- l_plot(ds, title = "density estimate",
            xlabel = "x", ylabel = "density",
            showScales = TRUE)

plot(p)
```

---

`l_plot.decomposed.ts` *The l\_plot method for classical seasonal decomposition of a time series by moving averages (i.e. via decompose)*

---

### Description

Creates a decomposed time series plot based on objects of class `decomposed.ts`

**Usage**

```
## S3 method for class 'decomposed.ts'
l_plot(x, y = NULL, xlabel = NULL,
       ylabel = NULL, title = NULL, tk_title = NULL, color = NULL,
       size = 1, lcolor = NULL, linewidth = 1, linkingGroup = NULL,
       showScales = TRUE, showGuides = TRUE, showLabels = TRUE, ...)
```

**Arguments**

<code>x</code>	a <code>decomposed.ts</code> object
<code>y</code>	NULL, ignored
<code>xlabel</code>	the labels for the x axes. This is a length four character vector one for each: of the original time series, the trend component, the seasonality component, and the remainder. If of length 1, the label is repeated; if NULL, <code>xlabel</code> is "time".
<code>ylabel</code>	the labels for the vertical axes. This is a length four character vector one for each: of the original time series, the trend component, the seasonality component, and the remainder. If NULL, the default, <code>ylabel</code> will be <code>c("data", "trend", "seasonality", "remainder")</code> if a character vector of length 1, the label is repeated four times.
<code>title</code>	an overall title for the entire display. If NULL (the default), the title will be "Seasonal Trend Analysis".
<code>tk_title</code>	provides an alternative window name to Tk's <code>wm title</code> . If NULL, <code>stl</code> will be used.
<code>color</code>	points colour of all time series. If NULL (the default) color will be <code>l_getOption("foreground")</code> .
<code>size</code>	points size of all time series. Default value is 1.
<code>lcolor</code>	line colour of all time series. If NULL (the default) <code>lcolor</code> will be <code>l_getOption("foreground")</code> .
<code>linewidth</code>	line width of all time series (incl. original and decomposed components. Default is 1.
<code>linkingGroup</code>	name of linking group. If NULL, one is created from the data name and class associated with <code>stlOrDecomposedTS</code> .
<code>showScales</code>	a logical as to whether to display the scales on all axes, default is TRUE.
<code>showGuides</code>	a logical as to whether to display background guide lines on all plots, default is TRUE.
<code>showLabels</code>	a logical as to whether to display axes labels on all plots, default is TRUE.
<code>...</code>	keyword value pairs passed off to <code>l_plot()</code> which constructs each loon scatter-plot component.

**Value**

A structure of class "l\_ts" containing four loon plots each representing a part of the decomposition by name: "original", "trend", "seasonal", and "remainder".

**See Also**

Other two-dimensional plotting functions: [l\\_plot.default](#), [l\\_plot.density](#), [l\\_plot.stl](#), [l\\_plot](#)  
 Other time series decomposition plotting functions: [l\\_plot.stl](#), [l\\_plot\\_ts](#)

**Examples**

```

decompose <- decompose(co2)
p <- l_plot(decompose, title = "Atmospheric carbon dioxide over Mauna Loa")
# names of plots in the display
names(p)
# names of states associated with the seasonality plot
names(p$seasonal)
# which can be set
p$seasonal['color'] <- "steelblue"

```

---

l_plot.default	<i>The default l_plot for plotting data</i>
----------------	---

---

**Description**

Creates an interactive 2d scatterplot. Also, if no loon inspector is open then the l\_plot call will also open a loon inspector.

**Usage**

```

## Default S3 method:
l_plot(x, y = NULL, color = "grey60",
       glyph = "ccircle", size = 4, active = TRUE, selected = FALSE,
       xlabel, ylabel, title, showLabels = TRUE, showScales = FALSE,
       showGuides = TRUE, guidelines = "white",
       guidesBackground = "grey92", foreground = "black",
       background = "white", parent = NULL, ...)

```

**Arguments**

x	the x and y arguments provide the x and y coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function <a href="#">xy.coords</a> for details. If supplied separately, they must be of the same length.
y	argument description is as for the x argument above.
color	colours of points (default "grey60"); colours are repeated until matching the number points,
glyph	shape of point; must be one of the primitive glyphs "circle", "ccircle", "ocircle", "square", "csquare", "osquare", "triangle", "ctriangle", "otriangle", "diamond", "cdiamond", or "odiamond". Prefixes "c" and "o" mean closed and open, respectively. Default is "ccircle" meaning a closed circle glyph.  Non-primitive glyphs such as polygons, images, text, point ranges, and even interactive glyphs like serial axes glyphs may be added, but only after the plot has been created.
size	size of the symbol (roughly in terms of area)

<code>active</code>	a logical determining whether points appear or not (default is TRUE for all points). If a logical vector is given of length equal to the number of points, then it identifies which points appear (TRUE) and which do not (FALSE).
<code>selected</code>	a logical determining whether points appear selected at first (default is FALSE for all points). If a logical vector is given of length equal to the number of points, then it identifies which points are (TRUE) and which are not (FALSE).
<code>xlabel</code>	Label for the horizontal (x) axis. If missing, one will be inferred from x if possible.
<code>ylabel</code>	Label for the vertical (y) axis. If missing, one will be inferred from y (or x) if possible.
<code>title</code>	Title for the plot, default is an empty string.
<code>showLabels</code>	logical to determine whether axes label (and title) should be presented.
<code>showScales</code>	logical to determine whether numerical scales should be presented on both axes.
<code>showGuides</code>	logical to determine whether to present background guidelines to help determine locations.
<code>guidelines</code>	colour of the guidelines shown when <code>showGuides = TRUE</code> (default "white").
<code>guidesBackground</code>	colour of the background to the guidelines shown when <code>showGuides = TRUE</code> (default "grey92").
<code>foreground</code>	foreground colour used by all other drawing (default "black").
<code>background</code>	background colour used for the plot (default "white")
<code>parent</code>	a valid Tk parent widget path. When the parent widget is specified (i.e. not NULL) then the plot widget needs to be placed using some geometry manager like <code>tkpack</code> or <code>tkplace</code> in order to be displayed. See the examples below.
<code>...</code>	named arguments to modify plot states.

## Details

The scatterplot displays a number of direct interactions with the mouse and keyboard, these include: zooming towards the mouse cursor using the mouse wheel, panning by right-click dragging and various selection methods using the left mouse button such as sweeping, brushing and individual point selection. See the documentation for `l_plot` for more details about the interaction gestures.

## See Also

[l\\_plot\\_arguments](#)

Other two-dimensional plotting functions: [l\\_plot.decomposed.ts](#), [l\\_plot.density](#), [l\\_plot.stl](#), [l\\_plot](#)

Other loon interactive states: [l\\_info\\_states](#), [l\\_plot](#), [l\\_state\\_names](#), [names.loon](#)



**Examples**

```

# default use as scatterplot

p1 <- with(iris, l_plot(Sepal.Length, Sepal.Width, color=Species,
                      title = "First plot"))

# The names of the info states that can be
# accessed or set. They can also be given values as
# arguments to l_plot.default()
names(p1)
p1["size"] <- 10

p2 <- with(iris, l_plot(Petal.Length ~ Petal.Width,
                      linkingGroup="iris_data",
                      title = "Second plot",
                      showGuides = FALSE))
p2["showScales"] <- TRUE

# link first plot with the second plot requires
# l_configure to coordinate the synchronization
l_configure(p1, linkingGroup = "iris_data", sync = "push")

p1['selected'] <- iris$Species == "versicolor"
p2["glyph"][p1['selected']] <- "cdiamond"

gridExtra::grid.arrange(loonGrob(p1), loonGrob(p2), nrow = 1)

# Use with other tk widgets
tt <- tktoplevel()
tktitle(tt) <- "Loon plots with custom layout"

p1 <- l_plot(parent=tt, x=c(1,2,3), y=c(3,2,1))
p2 <- l_plot(parent=tt, x=c(4,3,1), y=c(6,8,4))

tkgrid(p1, row=0, column=0, sticky="nesw")
tkgrid(p2, row=0, column=1, sticky="nesw")

tkgrid.columnconfigure(tt, 0, weight=1)
tkgrid.columnconfigure(tt, 1, weight=1)

tkgrid.rowconfigure(tt, 0, weight=1)

```

---

l\_plot.density

*The l\_plot method for density objects.*


---

**Description**

Creates an loon plot displaying contours of the supplied (kernel) density estimate

**Usage**

```
## S3 method for class 'density'
l_plot(x, y = NULL, xlabel = NULL, ylabel = NULL,
       title = NULL, linewidth = 2, color = NULL, size = 0.5, ...)
```

**Arguments**

x	a density object
y	NULL, ignored.
xlabel	the graphical parameter xlabel labelling the x axis of the plot. If NULL (the default), an xlabel is created based on the information available from the density objects.
ylabel	the graphical parameter ylabel labelling the y axis of the plot. If NULL (the default), an ylabel is created based on the combination of data name and "density".
title	providing a title for the plot. If NULL (the default), the title will be the call which produced the result.
linewidth	line width, default value is 2.
color	line colour. If NULL (the default) colour will be taken from l_getOption("foreground").
size	points size, default value is 0.5
...	named arguments being states passed to l_plot()

**See Also**

[l\\_layer.density](#)

Other two-dimensional plotting functions: [l\\_plot.decomposed.ts](#), [l\\_plot.default](#), [l\\_plot.stl](#), [l\\_plot](#)

**Examples**

```
ds <- density(faithful$eruptions)
p <- l_plot(ds, color = "steelblue")
```

---

l\_plot.map

*Create a plot with a map layer*


---

**Description**

Creates a scatterplot widget and layers the map in front.

**Usage**

```
## S3 method for class 'map'
l_plot(x, ...)
```

**Arguments**

x                    object of class map (defined in the maps library)  
 ...                  arguments forwarded to `l_layer.map`

**Value**

Scatterplot widget plot handle

**See Also**

`l_layer`, `l_layer.map`, `map`

**Examples**

```
if (requireNamespace("maps", quietly = TRUE)) {
  p <- l_plot(maps::map('world', fill=TRUE, plot=FALSE))
}
```

---

`l_plot.stl`

*The l\_plot methods for seasonally decomposed by loess time series objects (i.e. via stl)*

---

**Description**

Creates a decomposed time series plot based on objects of class `stl`

**Usage**

```
## S3 method for class 'stl'
l_plot(x, y = NULL, xlabel = NULL, ylabel = NULL,
       title = NULL, tk_title = NULL, color = NULL, size = 1,
       lcolor = NULL, linewidth = 1, linkingGroup = NULL,
       showScales = TRUE, showGuides = TRUE, showLabels = TRUE, ...)
```

**Arguments**

x                    an `stl` object  
 y                    NULL, ignored  
 xlabel              the labels for the x axes. This is a length four character vector one for each: of the original time series, the trend component, the seasonality component, and the remainder. If of length 1, the label is repeated; if NULL, xlabel is "time".  
 ylabel              the labels for the vertical axes. This is a length four character vector one for each: of the original time series, the trend component, the seasonality component, and the remainder. If NULL, the default, ylabel will be `c("data", "trend", "seasonality", "remainder")`. If a character vector of length 1, the label is repeated four times.

<code>title</code>	an overall title for the entire display. If NULL (the default), the title will be "Seasonal Trend Analysis".
<code>tk_title</code>	provides an alternative window name to Tk's <code>wm title</code> . If NULL, <code>stl</code> will be used.
<code>color</code>	points colour of all time series. If NULL (the default) color will be <code>l_getOption("foreground")</code> .
<code>size</code>	points size of all time series. Default value is 1.
<code>lcolor</code>	line colour of all time series. If NULL (the default) <code>lcolor</code> will be <code>l_getOption("foreground")</code> .
<code>linewidth</code>	line width of all time series (incl. original and decomposed components. Default is 1.
<code>linkingGroup</code>	name of linking group. If NULL, one is created from the data name and class associated with <code>stlOrDecomposedTS</code> .
<code>showScales</code>	a logical as to whether to display the scales on all axes, default is TRUE.
<code>showGuides</code>	a logical as to whether to display background guide lines on all plots, default is TRUE.
<code>showLabels</code>	a logical as to whether to display axes labels on all plots, default is TRUE.
<code>...</code>	keyword value pairs passed off to <code>l_plot()</code> which constructs each loon scatter-plot component.

### Value

A structure of class "l\_ts" containing four loon plots each representing a part of the decomposition by name: "original", "trend", "seasonal", and "remainder".

### See Also

Other time series decomposition plotting functions: [l\\_plot.decomposed.ts](#), [l\\_plot\\_ts](#)

Other two-dimensional plotting functions: [l\\_plot.decomposed.ts](#), [l\\_plot.default](#), [l\\_plot.density](#), [l\\_plot](#)

### Examples

```
co2_stl <- stl(co2, "per")
p <- l_plot(co2_stl, title = "Atmospheric carbon dioxide over Mauna Loa")
# names of plots in the display
names(p)
# names of states associated with the seasonality plot
names(p$seasonal)
# which can be set
p$seasonal['color'] <- "steelblue"
```

---

l_plot3D	<i>Create an interactive loon 3d plot widget</i>
----------	--

---

### Description

l\_plot3D is a generic function for creating interactive visualization environments for R objects.

### Usage

```
l_plot3D(x, y, z, axisScaleFactor, ...)
```

### Arguments

x	the x, y and z arguments provide the x, y and z coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details. If supplied separately, they must be of the same length.
y	the y coordinates of points in the plot, optional if x is an appropriate structure.
z	the z coordinates of points in the plot, optional if x is an appropriate structure.
axisScaleFactor	the amount to scale the axes at the centre of the rotation. Default is 1. All numerical values are acceptable (0 removes the axes, < 0 reverses their direction.)
...	named arguments to modify plot states.

### Details

To get started with loon it is recommended to read loons website which can be accessed via the `l_help()` function call.

NOTE: Although it is possible to programmatically add layers to an `l_plot3D`, these will not appear as part of the 3D plot's display. There is no provision at present to incorporate rotation of 3D geometric objects other than point glyphs.

### Value

widget handle

### See Also

[l\\_info\\_states](#)

Other three-dimensional plotting functions: [l\\_plot3D.default](#), [l\\_scale3D](#)

**Examples**

```

with(quakes,
  l_plot3D(long, lat, depth, linkingGroup = "quakes")
)

with(l_scale3D(quakes),
  l_plot3D(long, lat, depth, linkingGroup = "quakes")
)

scaled_quakes <- l_scale3D(quakes)
with(scaled_quakes,
  l_plot3D(long, lat, depth, linkingGroup = "quakes")
)

with(scaled_quakes,
  l_plot3D(mag, stations, depth, linkingGroup = "quakes")
)

# Or together:
with(scaled_quakes,{
  l_plot3D(long, lat, depth, linkingGroup = "quakes")
  l_plot3D(mag, stations, depth, linkingGroup = "quakes")
}
)

# Get an R (grid) graphics plot of a loon plot
p <- with(scaled_quakes,
  l_plot3D(long, lat, depth, linkingGroup = "quakes"))
plot(p)
# or with more control about grid parameters
grid.loon(p)
# or to save the grid data structure (grob) for later use
pg <- loonGrob(p)

# Use with other tk widgets
tt <- tktoplevel()
p1 <- l_plot3D(parent=tt, x=c(1,2,3), y=c(3,2,1), z=c(1,2,3))
p2 <- l_plot3D(parent=tt, x=c(4,3,1), y=c(6,8,4), z=c(3,2,1))

tkgrid(p1, row=0, column=0, sticky="nesw")
tkgrid(p2, row=0, column=1, sticky="nesw")
tkgrid.columnconfigure(tt, 0, weight=1)
tkgrid.columnconfigure(tt, 1, weight=1)
tkgrid.rowconfigure(tt, 0, weight=1)

tktitle(tt) <- "Loon plots with custom layout"

```

---

l_plot3D.default	<i>The default l_plot method to create 3d interactive scatterplot</i>
------------------	---

---

### Description

Creates an interactive 3d scatterplot. Also, if no loon inspector is open then the l\_plot3D call will also open a loon inspector.

### Usage

```
## Default S3 method:
l_plot3D(x, y = NULL, z = NULL,
  axisScaleFactor = 1, color = "grey60", glyph = "ccircle",
  size = 4, active = TRUE, selected = FALSE, xlabel, ylabel, zlabel,
  title, showLabels = TRUE, showScales = FALSE, showGuides = TRUE,
  guidelines = "white", guidesBackground = "grey92",
  foreground = "black", background = "white", parent = NULL, ...)
```

### Arguments

x	the x, y and z arguments provide the x, y and z coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function xyz.coords for details. If supplied separately, they must be of the same length.
y	the y coordinates of points in the plot, optional if x is an appropriate structure.
z	the z coordinates of points in the plot, optional if x is an appropriate structure.
axisScaleFactor	the amount to scale the axes at the centre of the rotation. Default is 1. All numerical values are acceptable (0 removes the axes, < 0 inverts the direction of all axes.)
color	colours of points; colours are repeated until matching the number points,
glyph	shape of point; must be one of the primitive glyphs "circle", "ccircle", "ocircle", "square", "csquare", "osquare", "triangle", "ctriangle", "otriangle", "diamond", "cdiamond", or "odiamond". Prefixes "c" and "o" mean closed and open, respectively. Default is "ccircle" meaning a closed circle glyph. Non-primitive glyphs such as polygons, images, text, point ranges, and even interactive glyphs like serial axes glyphs may be added, but only after the plot has been created.
size	size of the symbol (roughly in terms of area)
active	a logical determining whether points appear or not (default is TRUE for all points). If a logical vector is given of length equal to the number of points, then it identifies which points appear (TRUE) and which do not (FALSE).

<code>selected</code>	a logical determining whether points appear selected at first (default is FALSE for all points). If a logical vector is given of length equal to the number of points, then it identifies which points are (TRUE) and which are not (FALSE).
<code>xlabel</code>	Label for the horizontal (x) axis. If missing, one will be inferred from x if possible.
<code>ylabel</code>	Label for the vertical (y) axis. If missing, one will be inferred from y (or x) if possible.
<code>zlabel</code>	Label for the third (perpendicular to the screen) (z) axis. If missing, one will be inferred from z (or x) if possible.
<code>title</code>	Title for the plot, default is an empty string.
<code>showLabels</code>	logical to determine whether axes label (and title) should be presented.
<code>showScales</code>	logical to determine whether numerical scales should be presented on both axes.
<code>showGuides</code>	logical to determine whether to present background guidelines to help determine locations.
<code>guidelines</code>	colour of the guidelines shown when <code>showGuides = TRUE</code> (default "white").
<code>guidesBackground</code>	colour of the background to the guidelines shown when <code>showGuides = TRUE</code> (default "grey92").
<code>foreground</code>	foreground colour used by all other drawing (default "black").
<code>background</code>	background colour used for the plot (default "white")
<code>parent</code>	a valid Tk parent widget path. When the parent widget is specified (i.e. not NULL) then the plot widget needs to be placed using some geometry manager like <code>tkpack</code> or <code>tkplace</code> in order to be displayed. See the examples below.
<code>...</code>	named arguments to modify plot states.

### Details

The scatterplot displays a number of direct interactions with the mouse and keyboard, these include: rotating, zooming towards the mouse cursor using the mouse wheel, panning by right-click dragging and various selection methods using the left mouse button such as sweeping, brushing and individual point selection. See the documentation for [l\\_plot3D](#) for more details about the interaction gestures.

### See Also

Other three-dimensional plotting functions: [l\\_plot3D](#), [l\\_scale3D](#)

### Examples

```
# default use as scatterplot

p1 <- with(quakes,
  l_plot3D(long, lat, depth)
)

p2 <- with(quakes,
```



```

    l_plot3D(mag, stations, depth)
  )

# link the two plots p1 and p2
l_configure(p1, linkingGroup = "quakes", sync = "push")
l_configure(p2, linkingGroup = "quakes", sync = "push")

```

---

l\_plot\_arguments

*Arguments common to l\_plot functions*


---

## Description

Like `plot` in R, `l_plot` is the generic plotting function for objects in loon.

This is the workhorse of loon and is often a key part of many other displays (e.g. `l_pairs` and `l_navgraph`)

Because plots in loon are interactive, the functions which create them have many arguments in common. The value of these arguments become ‘infostates’ once the plot is instantiated. These can be accessed and set using the usual R square bracket operators ‘`[]`’ and ‘`[]<-`’ using the statename as a string. The state names can be found from an instantiated loon plot either via `l_info_states()` or, more in keeping with the R programming style, via `names()` (uses the method `names.loon()` for loon objects).

The same state names can be passed as arguments with values to a `l_plot()` call. As arguments many of the common ones are described below.

## Arguments

<code>x</code>	the <code>x</code> and <code>y</code> arguments provide the <code>x</code> and <code>y</code> coordinates for the plot. Any reasonable way of defining the coordinates is acceptable. See the function <code>xy.coords</code> for details. If supplied separately, they must be of the same length.
<code>y</code>	argument description is as for the <code>x</code> argument above.
<code>linkingGroup</code>	a string naming a group of plots to be linked. All plots with the same <code>linkingGroup</code> will have the same values of their linked states (see <code>l_getLinkedStates()</code> and <code>l_setLinkedStates()</code> ).
<code>linkingKey</code>	an <code>n</code> -dimensional character vector of unique entries. The entries identify which points match other points in other plots. Default is <code>c("0", "1", ..., "n-1")</code> (for numerical <code>n</code> ).
<code>itemLabel</code>	an <code>n</code> -dimensional character vector whose values are displayed in a pop-up box over any point whenever the mouse hovers over that point (provided <code>showItemLabels = TRUE</code> ).  This action is commonly known as providing a "tool tip". Note that all objects drawn in any layer of a plot (e.g. maps) will have an <code>itemLabel</code> .
<code>showItemLabels</code>	a logical (default <code>FALSE</code> ) which indicates whether the "tool tip" <code>itemLabel</code> is to be displayed whenever the mouse hovers over it.

color	colours of points (default "grey60"); colours are repeated until matching the number points,
glyph	the visual representation of the point. Argument values can be any of <b>the string names of primitive glyphs</b> "circle", "ccircle", "ocircle", <b>squares or boxes</b> "square", "csquare", "osquare", <b>triangles</b> "triangle", "ctriangle", "otriangle", <b>diamonds</b> "diamond", "cdiamond", or "odiamond". Note that prefixes "c" and "o" may be thought of as closed and open, respectively. The set of values are returned by <code>l_primitiveGlyphs()</code> . <b>the string names of constructed glyphs</b> <b>text as glyphs</b> see <code>l_glyph_add_text()</code> <b>point ranges</b> see <code>l_glyph_add_pointrange()</code> <b>polygons</b> see <code>l_glyph_add_polygon()</code> <b>parallel coordinates</b> see <code>l_glyph_add_serialaxes()</code> <b>star or radial axes</b> see <code>l_glyph_add_serialaxes()</code> <b>or any plot created using R</b> see <code>l_make_glyphs()</code> Note that glyphs are constructed and given a stringname to be used in the inspector.
size	size of the symbol (roughly in terms of area)
active	a logical determining whether points appear or not (default is TRUE for all points). If a logical vector is given of length equal to the number of points, then it identifies which points appear (TRUE) and which do not (FALSE).
selected	a logical determining whether points appear selected at first (default is FALSE for all points). If a logical vector is given of length equal to the number of points, then it identifies which points are (TRUE) and which are not (FALSE).
xlabel	Label for the horizontal (x) axis. If missing, one will be inferred from x if possible.
ylabel	Label for the vertical (y) axis. If missing, one will be inferred from y (or x) if possible.
title	Title for the plot, default is an empty string.
minimumMargins	the minimal size (in pixels) of the margins around the plot (bottom, left, top, right)
showLabels	logical to determine whether axes label (and title) should be presented.
showScales	logical to determine whether numerical scales should be presented on both axes.
showGuides	logical to determine whether to present background guidelines to help determine locations.
guidelines	colour of the guidelines shown when showGuides = TRUE (default "white").
guidesBackground	colour of the background to the guidelines shown when showGuides = TRUE (default "grey92").
foreground	foreground colour used by all other drawing (default "black").
background	background colour used for the plot (default "white")
parent	a valid Tk parent widget path. When the parent widget is specified (i.e. not NULL) then the plot widget needs to be placed using some geometry manager like <code>tkpack</code> or <code>tkplace</code> in order to be displayed. See the examples below.
...	named arguments to modify plot states.

**Details**

The interactive displays permit a number of direct interactions using the mouse and keyboard, these include: zooming towards the mouse cursor using the mouse wheel, panning by right-click dragging and various selection methods using the left mouse button such as sweeping, brushing and individual point selection. See the documentation for [l\\_plot](#) for more details about the interaction gestures.

**See Also**

the demos `demo(l_glyph_sizes, package = "loon")`, `demo(l_glyphs, package = "loon")`, and `demo(l_make_glyphs, package = "loon")`.

**Examples**

```
## Not run:
# default use as scatterplot

p1 <- with(iris, l_plot(x = Sepal.Length,
                      y = Sepal.Width,
                      color=Species,
                      title = "Sepal sizes"))

# The names of the info states that can be
# accessed or set. They can also be given values as
# arguments to l_plot.default()
names(p1)
versicolor <- (iris$Species == "versicolor")
p1["size"] <- 10
p1["glyph"][versicolor]<- "csquare"
p1["minimumMargins"][1] <- 100

## End(Not run)
```

---

`l_plot_inspector`      *Create a Scatterplot Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_plot_inspector(parent = NULL, ...)
```

**Arguments**

<code>parent</code>	parent widget path
<code>...</code>	state arguments

**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_plot_inspector()
```

---

`l_plot_inspector_analysis`

*Create a Scatterplot Analysis Inspector*

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_plot_inspector_analysis(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_plot_inspector_analysis()
```

---

l_plot_ts	<i>Draw a decomposed time series loon plot</i>
-----------	--

---

### Description

`l_plot_ts` is a generic function for creating a decomposed time series plot. It is mainly used in `l_plot.decomposed.ts` and `l_plot.stl`

### Usage

```
l_plot_ts(x, color = NULL, size = 1, lcolor = NULL, linewidth = 1,
          xlabel = NULL, ylabel = NULL, title = NULL, tk_title = NULL,
          linkingGroup = NULL, showScales = TRUE, showGuides = TRUE,
          showLabels = TRUE, ...)
```

### Arguments

<code>x</code>	Either an <code>stl</code> object or a <code>decomposed.ts</code> object.
<code>color</code>	points colour of all time series. If <code>NULL</code> (the default) color will be <code>l_getOption("foreground")</code> .
<code>size</code>	points size of all time series. Default value is 1.
<code>lcolor</code>	line colour of all time series. If <code>NULL</code> (the default) <code>lcolor</code> will be <code>l_getOption("foreground")</code> .
<code>linewidth</code>	line width of all time series (incl. original and decomposed components. Default is 1.
<code>xlabel</code>	the labels for the x axes. This is a length four character vector one for each: of the original time series, the trend component, the seasonality component, and the remainder. If of length 1, the label is repeated; if <code>NULL</code> , <code>xlabel</code> is "time".
<code>ylabel</code>	the labels for the vertical axes. This is a length four character vector one for each: of the original time series, the trend component, the seasonality component, and the remainder. If <code>NULL</code> , the default, <code>ylabel</code> will be <code>c("data", "trend", "seasonality", "remainder")</code> . If a character vector of length 1, the label is repeated four times.
<code>title</code>	an overall title for the entire display. If <code>NULL</code> (the default), the title will be "Seasonal Trend Analysis".
<code>tk_title</code>	provides an alternative window name to Tk's <code>wm title</code> . If <code>NULL</code> , <code>stl</code> will be used.
<code>linkingGroup</code>	name of linking group. If <code>NULL</code> , one is created from the data name and class associated with <code>stlOrDecomposedTS</code> .
<code>showScales</code>	a logical as to whether to display the scales on all axes, default is <code>TRUE</code> .
<code>showGuides</code>	a logical as to whether to display background guide lines on all plots, default is <code>TRUE</code> .
<code>showLabels</code>	a logical as to whether to display axes labels on all plots, default is <code>TRUE</code> .
<code>...</code>	keyword value pairs passed off to <code>l_plot()</code> which constructs each loon scatter-plot component.

**Value**

A structure of class "l\_ts" containing four loon plots each representing a part of the decomposition by name: "original", "trend", "seasonal", and "remainder".

**See Also**

[l\\_plot.stl](#), [l\\_plot.decomposed.ts.stl](#), or [decompose](#).

Other time series decomposition plotting functions: [l\\_plot.decomposed.ts](#), [l\\_plot.stl](#)

---

l_primitiveGlyphs	<i>The primitive glyphs available to a scatterplot or graph display</i>
-------------------	---

---

**Description**

Returns a vector of the available primitive glyphs.

**Usage**

```
l_primitiveGlyphs()
```

**Details**

The scatterplot and graph displays both have the n-dimensional state 'glyph' that assigns each data point or graph node a glyph (i.e. a visual representation).

Loon distinguishes between primitive and non-primitive glyphs: the primitive glyphs are always available for use whereas the non-primitive glyphs need to be first specified and added to a plot before they can be used.

The primitive glyphs are:

```
'circle', 'ocircle', 'ccircle'
'square', 'osquare', 'csquare'
'triangle', 'otriangle', 'ctriangle'
'diamond', 'odiamond', 'cdiamond'
```

Note that the letter 'o' stands for outline only, and the letter 'c' stands for contrast and adds an outline with the 'foreground' color (black by default).

For more information run: `l_help("learn_R_display_plot.html#glyphs")`

**Value**

A character vector of the names of all primitive glyphs in loon.

**See Also**

Other glyph functions: [l\\_glyph\\_add.default](#), [l\\_glyph\\_add\\_image](#), [l\\_glyph\\_add\\_pointrange](#), [l\\_glyph\\_add\\_polygon](#), [l\\_glyph\\_add\\_serialaxes](#), [l\\_glyph\\_add\\_text](#), [l\\_glyph\\_add](#), [l\\_glyph\\_delete](#),

[l\\_glyph\\_getLabel](#), [l\\_glyph\\_getType](#), [l\\_glyph\\_ids](#), [l\\_glyph\\_relabel](#)

---

`l_redraw` *Force a Content Redraw of a Plot*

---

### Description

Force redraw the plot to make sure that all the visual elements are placed correctly.

### Usage

```
l_redraw(widget)
```

### Arguments

`widget` widget path as a string or as an object handle

### Details

Note that this function is intended for debugging. If you find that the display does not display the data according to its plot states then please contact loon's package maintainer.

### Examples

```
p <- l_plot(iris)
l_redraw(p)
```

---

`l_resize` *Resize Plot Widget*

---

### Description

Resizes the toplevel widget to a specific size.

### Usage

```
l_resize(widget, width, height)
```

### Arguments

`widget` widget path as a string or as an object handle  
`width` width in pixels  
`height` in pixels

### See Also

[l\\_size](#), [l\\_size<-](#)

**Examples**

```
p <- l_plot(iris)

l_resize(p, 300, 300)
l_size(p) <- c(500, 500)
```

---

`l_Rlist2nestedTclList` *Convert an R list to a nested Tcl list*

---

**Description**

This is a helper function to create a nested Tcl list from an R list (i.e. a list of vectors).

**Usage**

```
l_Rlist2nestedTclList(x)
```

**Arguments**

`x` a list of vectors

**Value**

a string that represents the tcl nested list

**See Also**

[l\\_nestedTclList2Rlist](#)

**Examples**

```
x <- list(c(1,3,4), c(4,3,2,1), c(4,3,2,5,6))
l_Rlist2nestedTclList(x)
```

---

`l_saveStatesRDS` *Save the info states of a loon plot widget as an RDS file*

---

**Description**

`l_saveStatesRDS` uses `saveRDS()` to save the info states of a loon plot as an R object to the named file. This is helpful, for example, when using RMarkdown or some other notebooking facility to recreate an earlier saved loon plot so as to present it in the document.

**Usage**

```
l_saveStatesRDS(p, states = c("color", "active", "selected",
  "linkingKey", "linkingGroup"), file = stop("missing name of file"),
  ...)
```



**Arguments**

p	the 'l_plot' object whose info states are to be saved.
states	either the logical 'TRUE' or a character vector of info states to be saved. Default value 'c("color", "active", "selected", "linkingKey", "linkingGroup")' consists of 'n' dimensional states that are common to many 'l_plot's and which are most important to reconstruct the plot's display in any summary. If 'states' is the logical 'TRUE', by 'names(p)' are saved.
file	is a string giving the file name where the saved information' will be written (custom suggests this file name end in the suffix '.rds'.
...	further arguments passed to saveRDS().

**Value**

a list of class 'l\_savedStates' containing the states and their values. Also has an attribute 'l\_plot\_class' which contains the class vector of the plot 'p'

**See Also**

[l\\_copyStates](#) [l\\_info\\_states](#) [readRDS](#) [saveRDS](#)

**Examples**

```
#
# Suppose you some plot that you created like
p <- l_plot(iris, showGuides = TRUE)
# and coloured groups by hand (using the mouse and inspector)
# so that you ended up with these colours:
p["color"] <- rep(c( "lightgreen", "firebrick", "skyblue"), each = 50)
# Having determined the colours you could save them (and other states)
# in a file of your choice, here some tempfile:
myFileName <- tempfile("myPlot", fileext = ".rds")
# Save the names states of p
l_saveStatesRDS(p,
                states = c("color", "active", "selected"),
                file = myFileName)
# These can later be retrieved and used on a new plot
# (say in RMarkdown) to set the new plot's values to those
# previously determined interactively.
p_new <- l_plot(iris, showGuides = TRUE)
p_saved_info <- readRDS(myFileName)
p_new["color"] <- p_saved_info$color
# The result is that p_new looks like p did
# (after your interactive exploration)
# and can now be plotted as part of the document
plot(p_new)
```

---

`l_scale3D`*Scale for 3d plotting*

---

**Description**

`l_scale3D` scales its argument in a variety of ways used for 3D visualization.

**Usage**

```
l_scale3D(x, center = TRUE, method = c("box", "sphere"))
```

**Arguments**

<code>x</code>	the matrix or data.frame whose columns are to be scaled.
<code>center</code>	either a logical value or numeric-alike vector of length equal to the number of columns of <code>x</code> , where 'numeric-alike' means that <code>as.numeric(.)</code> will be applied successfully if <code>is.numeric(.)</code> is not true.
<code>method</code>	the scaling method to use: if "box" (the default) then the columns are scaled to have equal ranges; if "sphere" then <code>x</code> is centered, scaled to equal standard deviation and then decomposed via a singular value decomposition so that the resulting variables are uncorrelated.

**Value**

a data.frame whose columns are centred and scaled according to the given arguments.

**See Also**

[l\\_plot3D scale](#)

Other three-dimensional plotting functions: [l\\_plot3D.default](#), [l\\_plot3D](#)

**Examples**

```
##### Iris
with(l_scale3D(iris[,1:4]),
     l_plot3D(Petal.Length, Petal.Width, Sepal.Length, linkingGroup = "iris"))

with(l_scale3D(iris[,1:4], method = "sphere"),
     l_plot3D(pc1, pc2, pc3, linkingGroup = "iris"))

# With the Species as a factor

with(l_scale3D(iris),
     l_plot3D(Petal.Length, Petal.Width, Sepal.Length, linkingGroup = "iris"))

with(l_scale3D(iris, method = "sphere"),
     l_plot3D(pc1, pc2, pc3, linkingGroup = "iris"))
```

---

l_scaleto_active	<i>Change Plot Region to Display All Active Data</i>
------------------	--

---

**Description**

The function modifies the zoomX, zoomY, panX, and panY so that all active data points are displayed.

**Usage**

```
l_scaleto_active(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

---

l_scaleto_layer	<i>Change Plot Region to Display All Elements of a Particular Layer</i>
-----------------	---

---

**Description**

The function modifies the zoomX, zoomY, panX, and panY so that all elements of a particular layer are displayed.

**Usage**

```
l_scaleto_layer(target, layer)
```

**Arguments**

target	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. '.l0.plot'), the remaining objects by their ids.
layer	layer id

**See Also**

[l\\_layer\\_ids](#)

---

l_scaleto_plot	<i>Change Plot Region to Display the All Data of the Model Layer</i>
----------------	--

---

**Description**

The function modifies the zoomX, zoomY, panX, and panY so that all elements in the model layer of the plot are displayed.

**Usage**

```
l_scaleto_plot(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

---

---

l_scaleto_selected	<i>Change Plot Region to Display All Selected Data</i>
--------------------	--

---

**Description**

The function modifies the zoomX, zoomY, panX, and panY so that all selected data points are displayed.

**Usage**

```
l_scaleto_selected(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

---

---

l_scaleto_world	<i>Change Plot Region to Display All Plot Data</i>
-----------------	--

---

**Description**

The function modifies the zoomX, zoomY, panX, and panY so that all elements in the plot are displayed.

**Usage**

```
l_scaleto_world(widget)
```

**Arguments**

widget	widget path as a string or as an object handle
--------	--

---

 l\_serialaxes                      *Create a Serialaxes Widget*


---

**Description**

The serialaxes widget displays multivariate data either as a stacked star glyph plot, or as a parallel coordinate plot.

**Usage**

```
l_serialaxes(data, sequence, scaling = "variable",
             axesLayout = "radial", showAxes = TRUE, parent = NULL, ...)
```

**Arguments**

data	a data frame with numerical data only
sequence	vector with variable names that defines the axes sequence
scaling	one of 'variable', 'data', 'observation' or 'none' to specify how the data is scaled. See Details and Examples for more information.
axesLayout	either "radial" or "parallel"
showAxes	boolean to indicate whether axes should be shown or not
parent	parent widget path
...	state arguments, see <a href="#">l_info_states</a> .

**Details**

The scaling state defines how the data is scaled. The axes display 0 at one end and 1 at the other. For the following explanation assume that the data is in a  $n \times p$  dimensional matrix. The scaling options are then

variable	per column scaling
observation	per row scaling
data	whole matrix scaling
none	do not scale

**Value**

plot handle object

**Examples**

```
s <- l_serialaxes(data=oliveAcids, color=olive$Area, title="olive data")
s['axesLayout'] <- 'parallel'
states <- l_info_states(s)
names(states)
```

```
#####
#
# Effect of the choice of the argument "scaling"
#
# To illustrate we will look at the four measurements of
# 150 iris flowers from the iris data of Edgar Anderson made
# famous by R.A. Fisher.
#
# First separate the measurements
irisFlowers <- iris[, 1:4]
# from their species
species <- iris[,5]
# and get some identifiers for the individual flowers
flowerIDs <- paste(species, 1:50)
#
# Now create parallel axes plots of the measurements
# using different scaling values.

#
# scaling = "variable"
#
# This is the standard scaling of most serial axes plots,
# scaling each axis from the minimum to the maximum of that variable.
# Hence it is the default scaling.
#
# More precisely, it maps the minimum value in each column (variable) to
# zero and the maximum to one. The result is every parallel
# axis will have a point at 0 and a point at 1.
#
# This scaling highlights the relationships (e.g. correlations)
# between the variables (removes the effect of the location and scale of
# each variable).
#
# For the iris data, ignoring species we see for example that
# Sepal.Length and Sepal.Width are negatively correlated (lots of
# crossings) across species but more positively correlated (mostly
# parallel lines) within each species (colour).
#
sa_var <- l_serialaxes(irisFlowers,
                      scaling = "variable", # scale within column
                      axesLayout = "parallel",
                      color = species,
                      linewidth = 2,
                      itemLabel = flowerIDs,
                      showItemLabels = TRUE,
                      title = "scaling = variable (initially)",
                      linkingGroup = "irisFlowers data")

#
# scaling = "observation"
#
# This maps the minimum value in each row (observation) to
# zero and the maximum value in each row to one.
```

```
#
# The result is that every observation (curve in the parallel
# coordinate plot) will touch 0 on at least one axis and touch
# 1 on another.
#
# This scaling highlights the differences between observations (rows)
# in terms of the relative measurements across the variables for each
# observation.
#
# For example, for the iris data we can see that for every flower (row)
# the Sepal.Length is the largest measurement and the Petal.Width
# is the smallest. Each curve gives some sense of the *shape* of each
# flower without regard to its size. Two species (versicolor and
# virginica) have similar shaped flowers (relatively long but narrow
# sepals and petals), whereas the third (setosa) has relatively large
# sepals compared to small petals.
#
sa_obs <- l_serialaxes(irisFlowers,
                      scaling = "observation", # scale within row
                      axesLayout = "parallel",
                      color = species,
                      linewidth = 2,
                      itemLabel = flowerIDs,
                      showItemLabels = TRUE,
                      title = "scaling = observation (initially)",
                      linkingGroup = "irisFlowers data")

#
# scaling = "data"
#
# This maps the minimum value in the whole dataset (over all elements)
# to zero and the maximum value in the whole dataset to one.
#
# The result is that every measurement is on the same numeric (if not
# measurement) scale. Highlighting the relative magnitudes of all
# numerical values in the data set, each curve shows the relative magnitudes
# without rescaling by variable.
#
# This is most sensible data such as the iris flower where all four measurements
# appear to have been taken on the same measuring scale.
#
# For example, for the iris data full data scaling preserves the size
# and shape of each flower. Again virginica is of roughly the same
# shape as versicolor but has distinctly larger petals.
# Setosa in contrast is quite differently shaped in both sepals and petals
# but with sepals more similar in size to the two other flowers and
# with significantly smaller petals.
sa_dat <- l_serialaxes(irisFlowers,
                      scaling = "data",          # scale using all data
                      axesLayout = "parallel",
                      color = species,
                      linewidth = 2,
                      itemLabel = flowerIDs,
```

```

        showItemLabels = TRUE,
        title = "scaling = data (initially)",
        linkingGroup = "irisFlowers data")

#
# scaling = "none"
#
# Sometimes we might wish to choose a min and max to use
# for the whole data set; or perhaps a separate min and max
# for each variable.

# This would be done outside of the construction of the plot
# and displayed by having scaling = "none" in the plot.
#
# For example, for the iris data, we might choose scales so that
# the minimum and the maximum values within the data set do not
# appear at the end points 0 and 1 of the axes but instead inside.
#
# Suppose we choose the following limits for all variables
lower_lim <- -3 ; upper_lim <- max(irisFlowers) + 1

# These are the limits we want to use to define the end points of
# the axes for all variables.
# We need only scale the data as
irisFlowers_0_1 <- (irisFlowers - lower_lim)/(upper_lim - lower_lim)
# Or alternatively using the built-in scale function
# (which allows different scaling for each variable)
irisFlowers_0_1 <- scale(irisFlowers,
                        center = rep(lower_lim, 4),
                        scale = rep((upper_lim - lower_lim), 4))

# Different scales for different
# And instruct the plot to not scale the data but plot it on the 0-1 scale
# for all axes. (Note any rescaled data outside of [0,1] will not appear.)
#
sa_none <- l_serialaxes(irisFlowers_0_1,
                      scaling = "none",          # do not scale
                      axesLayout = "parallel",
                      color = species,
                      linewidth = 2,
                      itemLabel = flowerIDs,
                      showItemLabels = TRUE,
                      title = "scaling = none (initially)",
                      linkingGroup = "irisFlowers data")

# This is particularly useful for "radial" axes to keep the polygons away from
# the centre of the display.
# For example
sa_none["axesLayout"] <- "radial"
# now displays each flower as a polygon where shapes and sizes are easily
# compared.
#
# NOTE: rescaling the data so that all values are within [0,1] is perhaps

```



```
# the best way to proceed (especially if there are natural lower and
# upper limits for each variable).
# Then scaling can always be changed via the inspector.
```

---

`l_serialaxes_inspector`*Create a Serialaxes Analysis Inspector*

---

### Description

Inspectors provide graphical user interfaces to oversee and modify plot states

### Usage

```
l_serialaxes_inspector(parent = NULL, ...)
```

### Arguments

parent	parent widget path
...	state arguments

### Value

widget handle

### See Also

[l\\_create\\_handle](#)

### Examples

```
i <- l_serialaxes_inspector()
```

---

`l_setAspect`*Set the aspect ratio of a plot*

---

### Description

The aspect ratio is defined by the ratio of the number of pixels for one data unit on the y axis and the number of pixels for one data unit on the x axes.

### Usage

```
l_setAspect(widget, aspect, x, y)
```

**Arguments**

widget	widget path as a string or as an object handle
aspect	aspect ratio, optional, if omitted then the x and y arguments have to be specified.
x	optional, if the aspect argument is missing then x and y can be specified and the aspect ratio is calculated using y/x.
y	see description for x argument above

**Examples**

```
p <- with(iris, l_plot(Sepal.Length ~ Sepal.Width, color=Species))

l_aspect(p)
l_setAspect(p, x = 1, y = 2)
```

---

<code>l_setColorList</code>	<i>Use custom colors for mapping nominal values to distinct colors</i>
-----------------------------	--

---

**Description**

Modify loon's color mapping list to a set of custom colors.

**Usage**

```
l_setColorList(colors)
```

**Arguments**

colors	vector with valid color names or hex-encoded colors
--------	---

**Details**

There are two commonly used mapping schemes of data values to colors: one scheme maps numeric values to colors on a color gradient and the other maps nominal data to colors that can be well differentiated visually (e.g. to highlight the different groups). Presently, loon always uses the latter approach for its color mappings. You can use specialized color palettes to map continuous values to color gradients as shown in the examples below.

When assigning values to a display state of type color then loon maps those values using the following rules

1. if all values already represent valid Tk colors (see [tkcolors](#)) then those colors are taken.
2. if the number of distinct values are less than number of values in loon's color mapping list then they get mapped according to the color list, see [l\\_setColorList](#) and [l\\_getColorList](#).
3. if there are more distinct values as there are colors in loon's color mapping list then loon's own color mapping algorithm is used. See [loon\\_palette](#) and for more details about the algorithm below in this documentation.

Loon's default color list is composed of the first 11 colors from the *hcl* color wheel (displayed below in the html version of the documentation). The letters in *hcl* stand for hue, chroma and luminance, and the *hcl* wheel is useful for finding "balanced colors" with the same chroma (radius) and luminance but with different hues (angles), see Ross Ihaka (2003) "Colour for presentation graphics", Proceedings of DSC, p. 2 (<https://www.stat.auckland.ac.nz/~ihaka/courses/787/color.pdf>).

The colors in loon's internal color list are also the default ones listed as the "modify color actions" in the analysis inspectors. To query and modify loon's color list use `l_getColorList` and `l_setColorList`.

In the case where there are more unique data values than colors in loon's color list then the colors for the mapping are taken from different locations distributed on the *hcl* color wheel (see above).

One of the advantages of using the *hcl* color wheel is that one can obtain any number of "balanced colors" with distinct hues. This is useful in encoding data with colors for a large number of groups; however, it should be noted that the more groups we have the closer the colors sampled from the wheel become and, therefore, the more similar in appearance.

A common way to sample distinct "balanced colors" on the *hcl* wheel is to choose evenly spaced hues distributed on the wheel (i.e. angles on the wheel). However, this approach leads to color sets where most colors change when the sample size (i.e. the number of sampled colors from the wheel) increases by one. For loon, it is desirable to have the first  $m$  colors of a color sample of size  $m+1$  to be the same as the colors in a color sample of size  $m$ , for all positive natural numbers  $m$ . Hence, we prefer to have a sequence of colors. This way, the colors on the inspectors stay relevant (i.e. they match with the colors of the data points) when creating plots that encode with color a data variable with different number of groups.

We implemented such a color sampling scheme (or color sequence generator) that also makes sure that neighboring colors in the sequence have different hues. In you can access this color sequence generator with `loon_palette`. The color wheels below show the color generating sequence twice, once for 16 colors and once for 32 colors.

Note, for the inspector: If there are more unique colors in the data points than there are on the inspectors then it is possible to add the next five colors in the sequence of the colors with the `+5` button. Alternatively, the `+` button on the modify color part of the analysis inspectors allows the user to pick any additional color with a color menu. Also, if you change the color mapping list and close and re-open the loon inspector these new colors show up in the modify color list.

When other color mappings of data values are required (e.g. numerical data to a color gradient) then the functions in the `scales` R package provide various mappings including mappings for qualitative, diverging and sequential values.

### See Also

[l\\_setColorList](#), [l\\_getColorList](#), [l\\_setColorList\\_ColorBrewer](#), [l\\_setColorList\\_hcl](#), [l\\_setColorList\\_baseR](#)

### Examples

```
l_plot(1:3, color=1:3) # loon's default mapping

cols <- l_getColorList()
l_setColorList(c("red", "blue", "green", "orange"))
```

```

## close and reopen inspector

l_plot(1:3, color=1:3) # use the new color mapping
l_plot(1:10, color=1:10) # use loons default color mapping as color list is too small

# reset to default
l_setColorList(cols)

## Not run:
# you can also perform the color mapping yourself, for example with
# the col_numeric function provided in the scales package
if (requireNamespace("scales", quietly = TRUE)) {
  p_custom <- with(olive, l_plot(stearic ~ oleic,
    color = scales::col_numeric("Greens", domain = NULL)(palmitic)))
}

## End(Not run)

```

---

`l_setColorList_baseR` *Set loon's color mapping list to the colors from base R*

---

### Description

Loon's color list is used to map nominal values to colors. See the documentation for [l\\_setColorList](#).

### Usage

```
l_setColorList_baseR()
```

### See Also

[l\\_setColorList](#), [l\\_setColorList\\_loon](#), [l\\_setColorList\\_ColorBrewer](#), [l\\_setColorList\\_hcl](#), [l\\_setColorList\\_baseR](#), [l\\_setColorList\\_ggplot2](#)

---

`l_setColorList_ColorBrewer`  
*Set loon's color mapping list to the colors from ColorBrewer*

---

### Description

Loon's color list is used to map nominal values to colors. See the documentation for [l\\_setColorList](#).

### Usage

```
l_setColorList_ColorBrewer(palette = c("Set1", "Set2", "Set3", "Pastel1",
  "Pastel2", "Paired", "Dark2", "Accent"))
```

**Arguments**

palette            one of the following RColorBrewer palette name: Set1, Set2, Set3, Pastel1, Pastel2, Paired, Dark2, or Accent

**Details**

Only the following palettes in ColorBrewer are available: Set1, Set2, Set3, Pastel1, Pastel2, Paired, Dark2, and Accent. See the examples below.

**See Also**

[l\\_setColorList](#), [l\\_setColorList\\_loon](#), [l\\_setColorList\\_ColorBrewer](#), [l\\_setColorList\\_hcl](#), [l\\_setColorList\\_baseR](#), [l\\_setColorList\\_ggplot2](#)

**Examples**

```
## Not run:
if (requireNamespace("RColorBrewer", quietly = TRUE)) {
  RColorBrewer::display.brewer.all()
}

## End(Not run)

l_setColorList_ColorBrewer("Set1")
p <- l_plot(iris)
```

---

`l_setColorList_ggplot2`

*Set loon's color mapping list to the colors from ggplot2*

---

**Description**

Loon's color list is used to map nominal values to colors. See the documentation for [l\\_setColorList](#).

**Usage**

```
l_setColorList_ggplot2()
```

**See Also**

[l\\_setColorList](#), [l\\_setColorList\\_loon](#), [l\\_setColorList\\_ColorBrewer](#), [l\\_setColorList\\_hcl](#), [l\\_setColorList\\_baseR](#), [l\\_setColorList\\_ggplot2](#)

---

`l_setColorList_hcl`     *Set loon's color mapping list to the colors from hcl color when*

---

**Description**

Loon's color list is used to map nominal values to colors. See the documentation for [l\\_setColorList](#).

**Usage**

```
l_setColorList_hcl(chroma = 56, luminance = 51, hue_start = 231)
```

**Arguments**

chroma	The chroma of the color. The upper bound for chroma depends on hue and luminance.
luminance	A value in the range [0,100] giving the luminance of the colour. For a given combination of hue and chroma, only a subset of this range is possible.
hue_start	The start hue for sampling. The hue of the color specified as an angle in the range [0,360]. 0 yields red, 120 yields green 240 yields blue, etc.

**Details**

Samples equally distant colors from the hcl color wheel. See the documentation for [hcl](#) for more information.

**See Also**

[l\\_setColorList](#), [l\\_setColorList\\_loon](#), [l\\_setColorList\\_ColorBrewer](#), [l\\_setColorList\\_hcl](#), [l\\_setColorList\\_baseR](#), [l\\_setColorList\\_ggplot2](#)

---

`l_setColorList_loon`     *Set loon's color mapping list to the colors from loon defaults*

---

**Description**

Loon's color list is used to map nominal values to colors. See the documentation for [l\\_setColorList](#).

**Usage**

```
l_setColorList_loon()
```

**See Also**

[l\\_setColorList](#), [l\\_setColorList\\_loon](#), [l\\_setColorList\\_ColorBrewer](#), [l\\_setColorList\\_hcl](#), [l\\_setColorList\\_baseR](#), [l\\_setColorList\\_ggplot2](#)

---

l_setLinkedStates	<i>Modify States of a Plot that are Linked in Loon's Standard Linking Model</i>
-------------------	---

---

### Description

Loon's standard linking model is based on three levels, the linkingGroup and linkingKey states and the *used linkable states*. See the details below.

### Usage

```
l_setLinkedStates(widget, states)
```

### Arguments

widget	widget path as a string or as an object handle
states	used linkable state names, see in details below

### Details

Loon's standard linking model is based on two states, linkingGroup and linkingKey. The full capabilities of the standard linking model are described here. However, setting the linkingGroup states for two or more displays to the same string is generally all that is needed for linking displays that plot data from the same data frame. Changing the linking group of a display is also the only linking-related action available on the analysis inspectors.

The first linking level is as follows: loon's displays are linked if they share the same string in their linkingGroup state. The default linking group 'none' is a keyword and leaves a display un-linked.

The second linking level is as follows. All n-dimensional states can be linked between displays. We call these states *linkable*. Further, only linkable states with the same name can be linked between displays. One consequence of this *shared state name* rule is that, with the standard linking model, the linewidth state of a serialaxes display cannot be linked with the size state of a scatterplot display. Also, each display maintains a list that defines which of its linkable states should be used for linking; we call these states the *used linkable states*. The default used linkable states are as follows

Display	Default <i>used linkable</i> states
scatterplot	selected, color, active, size
histogram	selected, color, active
serialaxes	selected, color, active
graph	selected, color, active, size

If any two displays are set to be linked (i.e. they share the same linking group) then the intersection of their *used linkable* states are actually linked.

The third linking level is as follows. Every display has a n-dimensional linkingKey state. Hence, every data point has an associated linking key. Data points between linked plots are linked if they share the same linking key.

---

`l_setOption`                    *Set the value of a loon display option*

---

**Description**

All of loon's displays access a set of common options. This function assigns the value to the named option.

**Usage**

```
l_setOption(option, value)
```

**Arguments**

<code>option</code>	the name of the option being set
<code>value</code>	the value to be assigned to the option. If <code>value == "default"</code> , then the option is set to loon's default value for it.

**Value**

the new value

**See Also**

[l\\_getOption](#), [l\\_getOptionNames](#), [l\\_userOptions](#), [l\\_userOptionDefault](#)

**Examples**

```
l_setOption("select-color", "red")  
l_setOption("select-color", "default")
```

---

`l_setTitleFont`                    *Set the title font of all loon displays*

---

**Description**

All of loon's displays access a set of common options. This function sets the font for the title bar of the displays.

**Usage**

```
l_setTitleFont(size = "16", weight = "bold", family = "Helvetica")
```



**Arguments**

size	the font size.
weight	the font size.
family	the font family.

**Value**

the value of the named option.

**See Also**

[l\\_getOptionNames](#), [l\\_userOptions](#), [l\\_userOptionDefault](#), [l\\_setOption](#)

**Examples**

```
loon::l_setTitleFont(14)
```

---

l\_size

*Query Size of a Plot Display*

---

**Description**

Get the width and height of a plot in pixels

**Usage**

```
l_size(widget)
```

**Arguments**

widget            widget path as a string or as an object handle

**Value**

Vector with width and height in pixels

**See Also**

[l\\_resize](#), [l\\_size<-](#)

---

`l_size<-` *Resize Plot Widget*

---

### Description

Resizes the toplevel widget to a specific size. This setter function uses [l\\_resize](#).

### Usage

```
l_size(widget) <- value
```

### Arguments

<code>widget</code>	widget path as a string or as an object handle
<code>value</code>	numeric vector of length 2 with width and height in pixels

### See Also

[l\\_resize](#), [l\\_size](#)

### Examples

```
p <- l_plot(iris)

l_resize(p, 300, 300)
l_size(p) <- c(500, 500)
```

---

`l_state_names` *Get State Names of Loon Object*

---

### Description

States of loon objects can be accessed with ``[`` and [l\\_cget](#) and modified with [l\\_configure](#).

### Usage

```
l_state_names(target)
```

### Arguments

<code>target</code>	either an object of class loon or a vector that specifies the widget, layer, glyph, navigator or context completely. The widget is specified by the widget path name (e.g. <code>' .l0.plot '</code> ), the remaining objects by their ids.
---------------------	---

### Details

In order to access values of a states use [l\\_info\\_states](#).

**Value**

state names

**See Also**

[l\\_info\\_states](#), [l\\_cget](#), [l\\_configure](#)

Other loon interactive states: [l\\_info\\_states](#), [l\\_plot.default](#), [l\\_plot](#), [names.loon](#)

l\_subwin

*Create a child widget path*

**Description**

This function is similar to `.Tk.subwin` except that does not the environment of the "tkwin" object to keep track of numbering the subwidgets. Instead it creates a widget path `(parent).looni`, where `i` is the smallest integer for which no widget exists yet.

**Usage**

```
l_subwin(parent, name = "w")
```

**Arguments**

parent	parent widget path
name	child name

**Value**

widget path name as a string

l\_throwErrorIfNotLoonWidget

*Throw an error if string is not associated with a loon widget*

**Description**

Helper function to ensure that a widget path is associated with a loon widget.

**Usage**

```
l_throwErrorIfNotLoonWidget(widget)
```

**Arguments**

widget	widget path name as a string
--------	------------------------------

**Value**

TRUE if the string is associated with a loon widget, otherwise an error is thrown.

---

<code>l_toR</code>	<i>Convert a Tcl Object to some other R object</i>
--------------------	--

---

**Description**

Return values from `.Tcl` and `tcl` are of class `tclObj` and often need to be mapped to a different data structure in R. This function is a helper class to do this mapping.

**Usage**

```
l_toR(x, cast = as.character)
```

**Arguments**

<code>x</code>	a <code>tclObj</code> object
<code>cast</code>	a function to conver the object to some other R object

**Value**

A object that is returned by the function specified with the `cast` argument.

---

<code>l_userOptionDefault</code>	<i>Get loon's system default value for the named display option.</i>
----------------------------------	--

---

**Description**

All of loon's displays access a set of common options. This function accesses and returns the default value for the named option.

**Usage**

```
l_userOptionDefault(option)
```

**Arguments**

<code>option</code>	the name of the user changeable loon display option whose default value is to be determined.
---------------------	--

**Value**

the default value for the named option

**See Also**

[l\\_getOptionNames](#), [l\\_getOption](#), [l\\_userOptionDefault](#), [l\\_userOptions](#)

**Examples**

```
l_userOptionDefault("background")
```

---

l_userOptions	<i>Get the names of all loon display options that can be set by the user.</i>
---------------	---

---

**Description**

All of loon's displays access a set of common options. This function accesses and returns the names of the subset of loon options which can be changed by the user.

**Usage**

```
l_userOptions()
```

**Value**

a vector of all user settable option names.

**See Also**

[l\\_getOptionNames](#), [l\\_getOption](#), [l\\_userOptionDefault](#), [l\\_setOption](#)

**Examples**

```
l_userOptions()
```

---

l_web	<i>Open a browser with loon's R documentation webpage</i>
-------	---

---

**Description**

l\_web opens a browser with the relevant page on the official loon documentation website at <http://great-northern-diver.github.io/loon/>.

**Usage**

```
l_web(page = "index", directory = c("home", "reference", "articles"),  
      ...)
```

**Arguments**

page	relative path to a page, the .html part may be omitted
directory	if "home" then page is ignored and the browser will open at the home page of the official loon documentation website at <a href="http://great-northern-diver.github.io/loon/">http://great-northern-diver.github.io/loon/</a> . If page refers to a loon manual reference, then directory must be "reference"; if page refers to the name of a vignette file, then directory should be "articles"
...	arguments forwarded to browseURL, e.g. to specify a browser

**See Also**

[l\\_help](#), [help](#), [vignette](#)

**Examples**

```
## Not run:
l_web()
#
vignette("introduction", package = "loon")
# or
l_web(page = "introduction", directory = "articles")
#
help(l_hist)
l_web(page = "l_hist", directory = "reference")

## End(Not run)
```

---

l\_widget

*Dummy function to be used in the Roxygen documentation*


---

**Description**

Dummy function to be used in the Roxygen documentation

**Usage**

```
l_widget(widget)
```

**Arguments**

widget            widget path name as a string

**Value**

widget path name as a string

---

l_worldview	<i>Create a Worldview Inspector</i>
-------------	-------------------------------------

---

**Description**

Inspectors provide graphical user interfaces to oversee and modify plot states

**Usage**

```
l_worldview(parent = NULL, ...)
```

**Arguments**

parent	parent widget path
...	state arguments

**Value**

widget handle

**See Also**

[l\\_create\\_handle](#)

**Examples**

```
i <- l_worldview()
```

---

l_zoom	<i>Zoom from and towards the center</i>
--------	---

---

**Description**

This function changes the plot states panX, panY, zoomX, and zoomY to zoom towards or away from the center of the current view.

**Usage**

```
l_zoom(widget, factor = 1.1)
```

**Arguments**

widget	widget path as a string or as an object handle
factor	a zoom factor

---

`measures1d`*Closure of One Dimensional Measures*

---

**Description**

Function creates a 1d measures object that can be used with [l\\_ng\\_plots](#) and [l\\_ng\\_ranges](#).

**Usage**

```
measures1d(data, ...)
```

**Arguments**

<code>data</code>	a data.frame with the data used to calculate the measures
<code>...</code>	named arguments, name is the function name and argument is the function to calculate the measure for each variable.

**Details**

For more information run: `l_help("learn_R_display_graph.html#measures")`

**Value**

a measures object

**See Also**

[l\\_ng\\_plots](#), [l\\_ng\\_ranges](#), [measures2d](#)

**Examples**

```
m1 <- measures1d(oliveAcids, mean=mean, median=median,
  sd=sd, q1=function(x)as.vector(quantile(x, probs=0.25)),
  q3=function(x)as.vector(quantile(x, probs=0.75)))
```

```
m1
m1()
m1(olive$palmitoleic>100)
m1('data')
m1('measures')
```



---

`measures2d`*Closure of Two Dimensional Measures*

---

### Description

Function creates a 2d measures object that can be used with [l\\_ng\\_plots](#) and [l\\_ng\\_ranges](#).

### Usage

```
measures2d(data, ...)
```

### Arguments

<code>data</code>	a data.frame with the data used to calculate the measures
<code>...</code>	named arguments, name is the function name and argument is the function to calculate the measure for each variable.

### Details

For more information run: `l_help("learn_R_display_graph.html#measures")`

### Value

a measures object

### See Also

[l\\_ng\\_plots](#), [l\\_ng\\_ranges](#), [measures2d](#)

### Examples

```
m <- measures2d(oliveAcids, separator='*', cov=cov, cor=cor)
m
m()
m(keep=olive$palmitic>1360)
m('data')
m('grid')
m('measures')
```

---

 minority
 

---



---

 Canadian Visible Minority Data 2006
 

---

**Description**

Population census count of various named visible minority groups in each of 33 major census metropolitan areas of Canada in 2006.

These data are from the 2006 Canadian census, publicly available from Statistics Canada.

**Usage**

minority

**Format**

A data frame with 33 rows and 18 variates

**Arab** Number identifying as ‘Arab’.

**Black** Number identifying as ‘Black’.

**Chinese** Number identifying as ‘Chinese’.

**Filipino** Number identifying as ‘Filipino’.

**Japanese** Number identifying as ‘Japanese’.

**Korean** Number identifying as ‘Korean’.

**Latin.American** Number identifying as ‘Latin American’.

**Multiple.visible.minority** Number identifying as being a member of more than one visible minority.

**South.Asian** Number identifying as ‘South Asian’.

**Southeast.Asian** Number identifying as ‘Southeast Asian’.

**Total.population** Total population of the metropolitan census area.

**Visible.minority.not.included.elsewhere** Number identifying as a member of a visible minority that was not included elsewhere.

**Visible.minority.population** Total number identifying as a member of some visible minority.

**West.Asian** Number identifying as ‘West Asian’.

**lat, long** Latitude and longitude (in degrees) of the metropolitan census area.

**googleLat, googleLong** Latitude and longitude in degrees determined using the Google Maps Geocoding API.

rownames(minority) are the names of the metropolitan areas or cities.

**Source**

<https://www.statcan.gc.ca/>

---

names.loon	<i>Get State Names of Loon Object</i>
------------	---------------------------------------

---

**Description**

States of loon objects can be accessed ``[`` and `l_cget` and modified with `l_configure`.

**Usage**

```
## S3 method for class 'loon'
names(x)
```

**Arguments**

x	loon object
---	-------------

**Value**

state names

**See Also**

Other loon interactive states: `l_info_states`, `l_plot.default`, `l_plot`, `l_state_names`

---

ndtransitiongraph	<i>Create a n-d transition graph</i>
-------------------	--------------------------------------

---

**Description**

A n-d transition graph has k-d nodes and all edges that connect two nodes that from a n-d subspace

**Usage**

```
ndtransitiongraph(nodes, n, separator = ":")
```

**Arguments**

nodes	node names of graph
n	integer, dimension an edge should represent
separator	character that separates spaces in node names

**Details**

For more information run: `l_help("learn_R_display_graph.html.html#graph-utilities")`

**Value**

graph object of class loongraph

**Examples**

```
g <- ndtransitiongraph(nodes=c('A:B', 'A:F', 'B:C', 'B:F'), n=3, separator=':')
```

---

olive

*Fatty Acid Composition of Italian Olive Oils*

---

**Description**

This data set records the percentage composition of 8 fatty acids found in the lipid fraction of 572 Italian olive oils. The oils are samples taken from three Italian regions varying number of areas within each region. The regions and their areas are recorded as shown in the following table:

<b>Region</b>	<b>Area</b>
North	North-Apulia, South-Apulia, Calabria, Sicily
South	East-Liguria, West-Liguria, Umbria
Sardinia	Coastal-Sardinia, Inland-Sardinia

**Usage**

olive

**Format**

A data frame containing 572 cases and 10 variates.

**Region** Italian olive oil general growing region: North, South, or Sardinia

**Area** These are "Administrative Regions" of Italy (e.g. Sicily, or Umbria), or parts of such a region like "Coastal-Sardinia" and "Inland-Sardinia" or "North-Apulia" and "South-Apulia". Administrative regions are larger than, and contain, Italian provinces.

**palmitic** Percentage (in hundredths of a percent) of Palmitic acid, or hexadecanoic acid in the olive oil. It is the most common saturated fatty acid found in animals, plants and micro-organisms.

**palmitoleic** Percentage (in hundredths of a percent) of Palmitoleic acid, an omega-7 monounsaturated fatty acid.

**stearic** Percentage (in hundredths of a percent) of Stearic acid, a saturated fatty acid. It is a waxy solid and its name comes from the Greek word for tallow. Like palmitic acid, it is one of the most common saturated fatty acids found in nature.

**oleic** Percentage (in hundredths of a percent) of Oleic acid, the most common fatty acid occurring in nature found in various animal and vegetable fats and oils.

**linoleic** Percentage (in hundredths of a percent) of Linoleic acid, a polyunsaturated omega-6 fatty acid. It is one of two essential fatty acids for humans.

**linolenic** Percentage (in hundredths of a percent) of Linolenic acid, a type of fatty acid. It can refer to one of two types of fatty acids or a mixture of both. One is an omega-3 essential fatty acid; the other an omega-6.

**arachidic** Percentage (in hundredths of a percent) of Arachidic acid, also known as eicosanoic acid, a saturated fatty acid that is used for the production of detergents, photographic materials and lubricants.

**eicosenoic** Percentage (in hundredths of a percent) of Eicosenoic acid, which may refer to one of three closely related fatty acids: gadoleic acid (omega-11), gondoic acid (omega-9), or paullinic acid (omega-7).

Note that the percentages (in hundredths of a percent) should sum to approximately 10,000 for each oil (row).

## References

Forina, M., Armanino, C., Lanteri, S., and Tiscornia, E. (1983) "Classification of Olive Oils from their Fatty Acid Composition", in Food Research and Data Analysis (Martens, H., Russwurm, H., eds.), p. 189, Applied Science Publ., Barking.

Wikipedia <https://en.wikipedia.org/>

## See Also

[oliveLocations](#)

---

oliveAcids

*Just the Fatty Acid Composition of Italian Olive Oils*

---

## Description

This is the [olive](#) data set minus the Region and Area variables.

## Usage

oliveAcids

## Format

A data frame containing 572 cases and 8 variates.

## See Also

[olive](#)

---

oliveLocations	<i>Geographic location of each Italian olive growing area named in the <a href="#">olive</a> data.</i>
----------------	--

---

**Description**

A longitude and latitude for each Area named in the [olive](#) data set.

**Usage**

```
oliveLocations
```

**Format**

A data frame containing 9 cases and 3 variates.

**Area** name of the Italian growing area of the olive oil.

**lat, long** latitude and longitude in degrees of the approximate centre of the named growing area

**Source**

<https://www.latlong.net>

**See Also**

[olive](#)

---

plot.loon	<i>Plot the current view of any loon plot in the current device.</i>
-----------	--

---

**Description**

This is a wrapper for `grid.loon()` to simplify the plotting of loon plots on any device. Frequent users are recommended to use `grid.loon()` for more control.

**Usage**

```
## S3 method for class 'loon'  
plot(x, y = NULL, ...)
```

**Arguments**

x	the loon plot to be plotted on the current device
y	NULL, will be ignored.
...	parameters passed to loonGrob

**Value**

invisible()

**See Also**

[loonGrob](#), [grid.loon](#), [l\\_export](#)

**Examples**

```
loonPlot <- with(iris, l_plot(Sepal.Length, Sepal.Width))
loonPlot['color'] <- iris$Species
loonPlot['selected'] <- iris$Species == "versicolor"
l_scaleto_selected(loonPlot)
loonPlot['showGuides'] <- TRUE
plot(loonPlot)
```

---

plot.loongraph

*Plot a loon graph object with base R graphics*

---

**Description**

This function converts the loongraph object to one of class graph and the plots it with its respective plot method.

**Usage**

```
## S3 method for class 'loongraph'
plot(x, ...)
```

**Arguments**

x	object of class loongraph
...	arguments forwarded to method

**Examples**

```
if (requireNamespace("Rgraphviz", quietly = TRUE)) {
  g <- loongraph(letters[1:4], letters[1:3], letters[2:4], FALSE)
  plot(g)
}
```

print.l\_layer            *Print a summary of a loon layer object*

---

**Description**

Prints the layer label and layer type

**Usage**

```
## S3 method for class 'l_layer'  
print(x, ...)
```

**Arguments**

x                    an l\_layer object  
...                  additional arguments are not used for this method

**See Also**

[l\\_layer](#)

---

print.measures1d        *Print function names from measure1d object*

---

**Description**

Prints the function names of a measure1d object using print.default.

**Usage**

```
## S3 method for class 'measures1d'  
print(x, ...)
```

**Arguments**

x                    measures1d object  
...                  arguments passed on to print.default



---

print.measures2d	<i>Print function names from measure2d object</i>
------------------	---

---

**Description**

Prints the function names of a measure2d object using `print.default`.

**Usage**

```
## S3 method for class 'measures2d'
print(x, ...)
```

**Arguments**

x	measures2d object
...	arguments passed on to <code>print.default</code>

---

scagnostics2d	<i>Closure of Two Dimensional Scagnostic Measures</i>
---------------	---

---

**Description**

Function creates a 2d measures object that can be used with [l\\_ng\\_plots](#) and [l\\_ng\\_ranges](#).

**Usage**

```
scagnostics2d(data, scagnostics = c("Clumpy", "Monotonic", "Convex",
  "Stringy", "Skinny", "Outlying", "Sparse", "Striated", "Skewed"),
  separator = ":")
```

**Arguments**

data	a data.frame with the data used to calculate the measures
scagnostics	vector with valid scanostics measure names, i.e "Clumpy", "Monotonic", "Convex", "Stringy", "Skinny", "Outlying", "Sparse", "Striated", "Skewed". Also the prefix "Not" can be added to each measure which equals 1-measure.
separator	string the separates variable names in 2d graph nodes

**Details**

For more information run: `l_help("learn_R_display_graph.html#measures")`

**Value**

a measures object

**See Also**

[l\\_ng\\_plots](#), [l\\_ng\\_ranges](#), [measures2d](#)

**Examples**

```
## Not run:
m <- scagnostics2d(oliveAcids, separator='**')
m
m()
m(olive$palmitoleic > 80)
m('data')
m('grid')
m('measures')

## End(Not run)
```

---

tkcolors

*List the valid Tk color names*


---

**Description**

The core of Loon is implemented in Tcl and Tk. Hence, when defining colors using color names, Loon uses the Tcl color representation and not those of R. The colors are taken from the Tk sources: `doc/colors.n`.

If you want to make sure that the color names are represented exactly as they are in R then you can convert the color names to hexencoded color strings, see the examples below.

**Usage**

```
tkcolors()
```

**Examples**

```
# check if R colors names and TK color names are the same
setdiff(tolower(colors()), tolower(tkcolors()))
setdiff(tolower(tkcolors()), tolower(colors()))

# hence there are currently more valid color names in Tk
# than there are in R

# Let's compare the colors of the R color names in R and Tk
tohex <- function(x) {
  sapply(x, function(xi) {
    crgb <- as.vector(col2rgb(xi))
    rgb(crgb[1], crgb[2], crgb[3], maxColorValue = 255)
  })
}

df <- data.frame(
```

```

    R_col = tohex(colors()),
    Tcl_col = hex12tohex6(1_hexcolor(colors())),
    row.names = colors(),
    stringsAsFactors = FALSE
  )

df_diff <- df[df$R_col != df$Tcl_col,]

if (requireNamespace("grid", quietly = TRUE)) {
  grid::grid.newpage()
  grid::pushViewport(grid::plotViewport())

  x_col <- grid::unit(0, "npc")
  x_R <- grid::unit(6, "lines")
  x_Tcl <- grid::unit(10, "lines")

  grid::grid.text('color', x=x_col, y=grid::unit(1, "npc"),
                 just='left', gp=grid::gpar(fontface='bold'))
  grid::grid.text('R', x=x_R, y=grid::unit(1, "npc"), just='center',
                 gp=grid::gpar(fontface='bold'))
  grid::grid.text('Tcl', x=x_Tcl, y=grid::unit(1, "npc"), just='center',
                 gp=grid::gpar(fontface='bold'))
  for (i in 1:nrow(df_diff)) {
    y <- grid::unit(1, "npc") - grid::unit(i*1.2, "lines")
    grid::grid.text(row.names(df_diff)[i], x=x_col, y=y, just='left')
    grid::grid.rect(x=x_R, y=y, width=grid::unit(3, "line"),
                   height=grid::unit(1, "line"), gp=grid::gpar(fill=df_diff[i,1]))
    grid::grid.rect(x=x_Tcl, y=y, width=grid::unit(3, "line"),
                   height=grid::unit(1, "line"), gp=grid::gpar(fill=df_diff[i,2]))
  }
}

```

---

UsAndThem

*Data to re-create Hans Rosling's famous "Us and Them" animation*


---

### Description

This data was sourced from <https://www.gapminder.org/> and contains Population, Life Expectancy, Fertility, Income, and Geographic.Region information between 1962 and 2013 for 198 countries.

### Usage

```
UsAndThem
```

### Format

A data frame with 9855 rows and 8 variables

**Country** country name

**Year** year of recorded measurements

**Population** country's population

**LifeExpectancy** average life expectancy in years at birth

**Fertility** in number of babies per woman

**Income** Gross domestic product per person adjusted for inflation and purchasing power (in international dollars)

**Geographic.Region** one of six large global regions

**Geographic.Region.ID** two letter identification of country

**Source**

<http://www.gapminder.org/>

# Index

## \*Topic **datasets**

- minority, 226
- olive, 228
- oliveAcids, 229
- oliveLocations, 230
- UsAndThem, 235
- .Tcl, 169, 220
- [.loon (l\_cget), 52
- [<-.loon (l\_configure), 54
  
- as.character, 82
- as.graph, 7, 19
- as.loongraph, 8
- as.raster, 147
  
- col\_factor, 9
- col\_numeric, 9
- color\_loon, 9
- complement, 10, 19
- complement.loongraph, 11
- completegraph, 11, 19
- condGrob, 12
- contourLines, 121
  
- decompose, 198
- density, 108, 180
- dist, 16
  
- graphreduce, 13
- grid.loon, 14, 21, 231
  
- hcl, 214
- heat.colors, 130
- help, 97, 222
- hex12tohex6, 15
  
- image, 130
  
- L2\_distance, 15
- l\_after\_idle, 26
- l\_aspect, 26
- l\_aspect<-, 27
- l\_bind\_canvas, 28, 29–32
- l\_bind\_canvas\_delete, 28, 29, 30–32
- l\_bind\_canvas\_get, 28, 29, 29, 31, 32
- l\_bind\_canvas\_ids, 28–30, 30, 32
- l\_bind\_canvas\_reorder, 28–31, 31
- l\_bind\_context, 32, 33–35
- l\_bind\_context\_delete, 33, 33, 34, 35
- l\_bind\_context\_get, 33, 33, 34, 35
- l\_bind\_context\_ids, 33, 34, 34, 35
- l\_bind\_context\_reorder, 33, 34, 35
- l\_bind\_glyph, 35, 36–38
- l\_bind\_glyph\_delete, 36, 36, 37, 38
- l\_bind\_glyph\_get, 36, 37, 38
- l\_bind\_glyph\_ids, 36, 37, 37, 38
- l\_bind\_glyph\_reorder, 36–38, 38
- l\_bind\_item, 39, 40–42, 63, 64
- l\_bind\_item\_delete, 39, 40, 41, 42
- l\_bind\_item\_get, 39, 40, 40, 41, 42
- l\_bind\_item\_ids, 39–41, 41, 42
- l\_bind\_item\_reorder, 39–41, 42
- l\_bind\_layer, 42, 43–45
- l\_bind\_layer\_delete, 43, 43, 44, 45
- l\_bind\_layer\_get, 43, 44, 45
- l\_bind\_layer\_ids, 43, 44, 44, 45
- l\_bind\_layer\_reorder, 43–45, 45
- l\_bind\_navigator, 46, 47–49
- l\_bind\_navigator\_delete, 46, 46, 47–49
- l\_bind\_navigator\_get, 46, 47, 47, 48, 49
- l\_bind\_navigator\_ids, 46–48, 48, 49
- l\_bind\_navigator\_reorder, 46–48, 48
- l\_bind\_state, 49, 50–52
- l\_bind\_state\_delete, 49, 50, 51, 52
- l\_bind\_state\_get, 49, 50, 50, 51, 52
- l\_bind\_state\_ids, 49–51, 51, 52
- l\_bind\_state\_reorder, 49–51, 52
- l\_cget, 52, 54, 106, 218, 219, 227
- l\_colRemoveAlpha, 53
- l\_configure, 53, 54, 65, 106, 218, 219, 227

- `l_context_add_context2d`, 54, 55, 57–59
- `l_context_add_geodesic2d`, 55, 55, 57–59
- `l_context_add_slicing2d`, 55, 56, 57–59
- `l_context_delete`, 57, 58, 59
- `l_context_getLabel`, 55, 57, 57, 58, 59
- `l_context_ids`, 55, 57, 58
- `l_context_relabel`, 55, 57, 58, 58
- `l_copyStates`, 59, 201
- `l_create_handle`, 53, 54, 62, 72–74, 95–97, 100, 101, 120, 125, 133, 196, 209, 223
- `l_createCompoundGrob`, 61
- `l_currentindex`, 63, 64
- `l_currenttags`, 63, 64
- `l_data`, 65
- `l_export`, 65, 231
- `l_export_valid_formats`, 66, 66
- `l_get_arrangeGrobArgs`, 71
- `l_getColorList`, 9, 66, 210, 211
- `l_getGraph`, 67
- `l_getLinkedStates`, 67, 193
- `l_getLocations`, 68
- `l_getOption`, 69, 70, 216, 221
- `l_getOptionNames`, 69, 70, 216, 217, 221
- `l_getPlots`, 70, 180
- `l_glyph_add`, 75, 77–79, 81–84, 198
- `l_glyph_add.default`, 76, 76, 77–79, 81–84, 198
- `l_glyph_add_image`, 75–77, 77, 78, 79, 81–84, 198
- `l_glyph_add_pointrange`, 75–77, 78, 79, 81–84, 194, 198
- `l_glyph_add_polygon`, 75–78, 79, 81–84, 194, 198
- `l_glyph_add_serialaxes`, 75–79, 80, 82–84, 194, 198
- `l_glyph_add_text`, 75–79, 81, 81, 82–84, 194, 198
- `l_glyph_delete`, 76–79, 81, 82, 82, 83, 84, 198
- `l_glyph_getLabel`, 76–79, 81–83, 83, 84, 199
- `l_glyph_getType`, 76–79, 81–83, 83, 84, 199
- `l_glyph_ids`, 76–79, 81–84, 84, 199
- `l_glyph_relabel`, 76–79, 81–84, 84, 199
- `l_glyphs_inspector`, 71
- `l_glyphs_inspector_image`, 72
- `l_glyphs_inspector_pointrange`, 73
- `l_glyphs_inspector_serialaxes`, 73
- `l_glyphs_inspector_text`, 74
- `l_graph`, 67, 85, 86, 87
- `l_graph.default`, 85, 85
- `l_graph.graph`, 85, 86, 86, 87
- `l_graph.loongraph`, 85, 86, 87
- `l_graph_inspector`, 95
- `l_graph_inspector_analysis`, 96
- `l_graph_inspector_navigators`, 96
- `l_graphswitch`, 87, 88–95
- `l_graphswitch_add`, 88, 88
- `l_graphswitch_add.default`, 89
- `l_graphswitch_add.graph`, 90
- `l_graphswitch_add.loongraph`, 90
- `l_graphswitch_delete`, 88, 91
- `l_graphswitch_get`, 88, 92
- `l_graphswitch_getLabel`, 88, 92
- `l_graphswitch_ids`, 88, 93, 94
- `l_graphswitch_move`, 88, 93
- `l_graphswitch_relabel`, 88, 94
- `l_graphswitch_reorder`, 88, 94
- `l_graphswitch_set`, 88, 95
- `l_help`, 54–56, 97, 222
- `l_hexcolor`, 15, 98
- `l_hist`, 98
- `l_hist_inspector`, 100
- `l_hist_inspector_analysis`, 101
- `l_image_import_array`, 77, 102, 103
- `l_image_import_files`, 77, 103
- `l_imageviewer`, 101, 103
- `l_info_states`, 49, 53–55, 60, 86, 87, 104, 107, 122, 128, 133, 136–138, 140–142, 144, 149, 150, 153–155, 180, 181, 184, 189, 193, 201, 205, 218, 219, 227
- `l_isLoonWidget`, 105
- `l_layer`, 105, 108–119, 122, 124–129, 132–142, 144, 145, 147, 149–154, 187, 232
- `l_layer.density`, 107, 186
- `l_layer.Line`, 108
- `l_layer.Lines`, 109
- `l_layer.map`, 110, 187
- `l_layer.Polygon`, 111
- `l_layer.Polygons`, 112
- `l_layer.SpatialLines`, 113
- `l_layer.SpatialLinesDataFrame`, 114
- `l_layer.SpatialPoints`, 115

- `l_layer.SpatialPointsDataFrame`, 116
- `l_layer.SpatialPolygons`, 117
- `l_layer.SpatialPolygonsDataFrame`, 118
- `l_layer_bbox`, 106, 120
- `l_layer_contourLines`, 121
- `l_layer_delete`, 106, 122, 124
- `l_layer_demote`, 106, 123
- `l_layer_expunge`, 106, 124
- `l_layer_getChildren`, 106, 124, 126, 145
- `l_layer_getLabel`, 106, 125, 151
- `l_layer_getParent`, 106, 125, 126, 145
- `l_layer_getType`, 106, 127
- `l_layer_group`, 105, 128
- `l_layer_groupVisibility`, 106, 129, 132, 135, 136, 152
- `l_layer_heatImage`, 130
- `l_layer_hide`, 106, 129, 131, 135, 136, 152
- `l_layer_ids`, 106, 132, 203
- `l_layer_index`, 106, 134, 139
- `l_layer_isVisible`, 106, 129, 132, 134, 135, 136, 152
- `l_layer_layerVisibility`, 106, 129, 132, 135, 135, 136, 152
- `l_layer_line`, 106, 121, 131, 136, 147
- `l_layer_lines`, 106, 137
- `l_layer_lower`, 106, 138, 147
- `l_layer_move`, 106, 134, 138, 139, 147
- `l_layer_oval`, 106, 140
- `l_layer_points`, 106, 141
- `l_layer_polygon`, 105, 142
- `l_layer_polygons`, 106, 143
- `l_layer_printTree`, 106, 139, 145
- `l_layer_promote`, 106, 145
- `l_layer_raise`, 106, 138, 146
- `l_layer_rasterImage`, 147
- `l_layer_rectangle`, 106, 148
- `l_layer_rectangles`, 106, 149
- `l_layer_relabel`, 106, 126, 151
- `l_layer_show`, 106, 129, 132, 135, 136, 152
- `l_layer_text`, 105, 106, 153
- `l_layer_texts`, 153, 154
- `l_layers_inspector`, 119
- `l_loon_inspector`, 155
- `l_make_glyphs`, 76, 77, 155, 194
- `l_move_grid`, 157, 158–163
- `l_move_halign`, 158, 158, 159–163
- `l_move_hdist`, 158, 159, 159, 160–163
- `l_move_jitter`, 158–160, 160, 161–163
- `l_move_reset`, 158–161, 161, 162, 163
- `l_move_valign`, 158–162, 162, 163
- `l_move_vdist`, 158–163, 163
- `l_navgraph`, 164, 180, 193
- `l_navigator_add`, 164, 166–169
- `l_navigator_delete`, 165, 165
- `l_navigator_getLabel`, 165, 166
- `l_navigator_getPath`, 166
- `l_navigator_ids`, 165, 167
- `l_navigator_relabel`, 165, 167
- `l_navigator_walk_backward`, 165, 168
- `l_navigator_walk_forward`, 165, 168
- `l_navigator_walk_path`, 165, 169
- `l_nestedTclList2Rlist`, 169, 200
- `l_ng_plots`, 170, 171, 173, 174, 177, 224, 225, 233, 234
- `l_ng_plots.default`, 170, 171, 174
- `l_ng_plots.measures`, 170, 171, 172, 174
- `l_ng_plots.scagnostics`, 170, 171, 174
- `l_ng_ranges`, 170, 171, 173–175, 175, 176–178, 224, 225, 233, 234
- `l_ng_ranges.default`, 175, 175, 178
- `l_ng_ranges.measures`, 175, 176, 176, 178
- `l_ng_ranges.scagnostics`, 175, 176, 178
- `l_pairs`, 179, 180, 193
- `l_plot`, 99, 104, 180, 180, 182, 184, 186, 188, 193, 195, 219, 227
- `l_plot.decomposed.ts`, 181, 181, 184, 186, 188, 198
- `l_plot.default`, 104, 180–182, 183, 186, 188, 219, 227
- `l_plot.density`, 180–182, 184, 185, 188
- `l_plot.map`, 180, 186
- `l_plot.stl`, 180–182, 184, 186, 187, 198
- `l_plot3D`, 189, 192, 202
- `l_plot3D.default`, 189, 191, 202
- `l_plot_arguments`, 181, 184, 193
- `l_plot_inspector`, 195
- `l_plot_inspector_analysis`, 196
- `l_plot_ts`, 182, 188, 197
- `l_primitiveGlyphs`, 76–79, 81–84, 194, 198
- `l_redraw`, 199
- `l_resize`, 199, 217, 218
- `l_Rlist2nestedTclList`, 170, 200
- `l_saveStatesRDS`, 60, 200
- `l_scale3D`, 189, 192, 202
- `l_scaletto_active`, 203
- `l_scaletto_layer`, 107, 203

- `l_scaletto_plot`, 204
- `l_scaletto_selected`, 204
- `l_scaletto_world`, 99, 107, 204
- `l_serialaxes`, 205
- `l_serialaxes_inspector`, 209
- `l_setAspect`, 209
- `l_setColorList`, 9, 25, 26, 53, 66, 67, 210, 210, 211–214
- `l_setColorList_baseR`, 211, 212, 212, 213, 214
- `l_setColorList_ColorBrewer`, 211, 212, 212, 213, 214
- `l_setColorList_ggplot2`, 212, 213, 213, 214
- `l_setColorList_hcl`, 211–214, 214
- `l_setColorList_loon`, 212–214, 214
- `l_setLinkedStates`, 67, 68, 193, 215
- `l_setOption`, 69, 70, 216, 217, 221
- `l_setTitleFont`, 216
- `l_size`, 199, 217, 218
- `l_size<-`, 218
- `l_state_names`, 104, 181, 184, 218, 227
- `l_subwin`, 219
- `l_throwErrorIfNotLoonWidget`, 219
- `l_toR`, 220
- `l_userOptionDefault`, 69, 70, 216, 217, 220, 221
- `l_userOptions`, 69, 70, 216, 217, 221, 221
- `l_web`, 97, 221
- `l_widget`, 222
- `l_worldview`, 223
- `l_zoom`, 223
- `linegraph`, 16, 19
- `linegraph.loongraph`, 17
- `loon`, 18
- `loon-package (loon)`, 18
- `loon_palette`, 9, 25, 210, 211
- `loongraph`, 19, 67, 86, 87, 89, 90, 92
- `loonGrob`, 14, 20, 231
- `loonGrob_layoutType`, 24
  
- `map`, 110, 187
- `measures1d`, 170, 171, 173–178, 224
- `measures2d`, 170, 171, 173–178, 224, 225, 225, 234
- `minority`, 226
  
- `names`, 193
- `names.loon`, 104, 181, 184, 193, 219, 227
  
- `ndtransitiongraph`, 227
  
- `olive`, 228, 229, 230
- `oliveAcids`, 229
- `oliveLocations`, 229, 230
  
- `plot`, 180, 193
- `plot.loon`, 14, 66, 230
- `plot.loongraph`, 231
- `png`, 156
- `print.l_layer`, 232
- `print.measures1d`, 232
- `print.measures2d`, 233
  
- `rainbow`, 130
- `rasterImage`, 147
- `readRDS`, 201
  
- `saveRDS`, 60, 201
- `scagnostics`, 174, 178
- `scagnostics2d`, 170, 171, 173–178, 233
- `scale`, 202
- `scales`, 211
- `sp`, 108–119
- `stl`, 180, 198
  
- `tcl`, 169, 220
- `terrain.colors`, 130
- `tkcolors`, 9, 210, 234
- `tkpack`, 184, 192, 194
- `tkplace`, 184, 192, 194
- `topo.colors`, 130
  
- `UsAndThem`, 235
  
- `vignette`, 222
  
- `xy.coords`, 136, 141, 154, 183, 193