Package 'loon'

July 26, 2017

Type Package

Title Interactive Statistical Data Visualization

Version 1.1.0

Date 2017-07-26

URL http://waddella.github.io/loon/

Description A toolkit for interactive data visualization and exploration.

License GPL-2

Depends R (>= 3.4.0), methods, tcltk

Imports tools, graphics, grDevices, utils, stats

Suggests maps, sp, graph, scagnostics, PairViz, RColorBrewer, RnavGraphImageData, rworldmap, rgl, Rgraphviz, RDRToolbox, kernlab, scales, MASS, dplyr, testthat, knitr, rmarkdown

LazyData true

RoxygenNote 6.0.1

VignetteBuilder knitr

NeedsCompilation no

Author Adrian Waddell [aut, cre], R. Wayne Oldford [aut]

Maintainer Adrian Waddell <adrian@waddell.ch>

R topics documented:

`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

```
library(graph)
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

```
library(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 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 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
library(grid)
grid.newpage()
pushViewport(plotViewport())
grid.rect()
n <- 2^(1:5)
pushViewport(dataViewport(xscale=c(0, max(n)+1), yscale=c(0, length(n)+1)))
grid.yaxis(at=c(1:length(n)), label=paste("n =", n))
for (i in rev(seq_along(n))) {
 cols <- pal(1:n[i])
 grid.points(x = 1:n[i], y = rep(i, n[i]), default.units = "native", pch=15, gp=gpar(col=cols))
}
grid.text("note the fist i colors are shared for each n" , y=unit(1,"npc")+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

completegraph *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

```
completegraph(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 idead 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])
```

---

graphreduce *Make each space in a node apprear only once*

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)

## Not run:
library(Rgraphviz)
```

```
plot(graphreduce(LLG))

## End(Not run)
```

`l_after_idle` *Evaluate a function on once the processor is idle*

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` *Query 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)
```

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 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 *Create a Canvas Binding*

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 http://www.tcl.tk/man/tcl8.6/TkCmd/bind.htm#M5.
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 webpage, 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 machtches 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 webpage, 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 l_bind_canvas_ids 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_canvas, l_bind_canvas_ids, l_bind_canvas_get, l_bind_canvas_delete

---

`l_bind_context` *Add a context binding*

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 webpage, 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 webpage, 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 machtches 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 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_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 webpage, 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

widget	widget path as a string or as an object handle
ids	new context binding id evaluation order, this must be a rearrangement of the elements returned by the l_bind_context_ids 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 *Get the event pattern and callback Tcl code of a glyph 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_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 *List glyph binding ids*

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 webpage, 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

`widget`	widget path as a string or as an object handle
`ids`	new glyph binding id evaluation order, this must be a rearrangement of the elements returned by the `l_bind_glyph_ids` 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_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

widget	widget path as a string or as an object handle
tags	item tags as as explained in `l_help("learn_R_bind.html#item-bindings")`
event	event patterns as defined for Tk canvas widget http://www.tcl.tk/man/tcl8.6/TkCmd/bind.htm#M5.
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

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

widget	widget path as a string or as an object handle
id	item 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_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

widget	widget path as a string or as an object handle
id	item 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_item, l_bind_item_ids, l_bind_item_delete, l_bind_item_reorder

---

`l_bind_item_ids` *List item binding ids*

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 webpage, 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

widget	widget path as a string or as an object handle
ids	new item binding id evaluation order, this must be a rearrangement of the elements returned by the l_bind_item_ids 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 *Get the event pattern and callback Tcl code of a layer 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_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 *List layer binding ids*

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 webpage, 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

`widget`	widget path as a string or as an object handle
`ids`	new layer binding id evaluation order, this must be a rearrangement of the elements returned by the `l_bind_layer_ids` 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_layer`, `l_bind_layer_ids`, `l_bind_layer_get`, `l_bind_layer_delete`

l_bind_navigator *Add a navigator binding*

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

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

navigator binding id

See Also

l_bind_navigator_ids, l_bind_navigator_get, l_bind_navigator_delete, l_bind_navigator_reorder

l_bind_navigator_delete *Delete a navigator binding*

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 webpage, 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 machtches 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 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_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 webpage, 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 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_navigator`, `l_bind_navigator_ids`, `l_bind_navigator_get`, `l_bind_navigator_delete`

`l_bind_state` *Add a state change binding*

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. `'.l0.plot'`), 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 webpage, 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

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.
id	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 machtches the event pattern.

Usage

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

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.
id	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")`

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` *List state binding ids*

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. `'.l0.plot'`), the remaining objects by their ids.

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 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

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.
ids	new state binding id evaluation order, this must be a rearrangement of the elements returned by the l_bind_state_ids 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_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 refered 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']
```

---

l_configure *Modify one or multiple plot states*

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 refered 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

navigator	navigator handle object
...	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` *Delete a context from a navigator*

Description

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

Usage

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

Arguments

`navigator`	navigator hanlde
`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 *Query the label of a context*

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 hanlde
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

l_context_ids *List context ids of a navigator*

Description

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

Usage

```
l_context_ids(navigator)
```

Arguments

navigator	navigator hanlde

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

---

l_context_relabel *Change the label of a context*

Description

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

Usage

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

Arguments

navigator	navigator hanlde
id	context id
label	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

---

l_create_handle *Create a loon object handle*

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(completegraph(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 *Get layer-relative index of the item below the mouse cursor*

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 *Get tags of the item below the mouse cursor*

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 *Convert an R data.frame to a Tcl dictionary*

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` *Export a loon plot as an image*

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-T` key combination opens a dialog to export he 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`

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 *Extract a loongraph or graph object from loon's graph display*

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 *Query the States that are Linked with Loon's Standard Linking Model*

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_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	l_glyph_add_text
Serialaxes	l_glyph_add_serialaxes
Pointranges	l_glyph_add_pointrange
Images	l_glyph_add_image
Polygon	l_glyph_add_polygon

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 cunstruct an arbitrary glyph using any system and save it as a png and then re-import them as 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, make_glyphs

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

---

l_glyph_add_image *Add an image glyphs*

Description

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

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 l_image_import_array and l_image_import_files 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, make_glyphs

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, 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 with in pixel.
`label`	label of a glyph (currently shown only in the glyph inspector)
`...`	state arguments

See Also

`l_glyph_add`

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, showArea = TRUE, label = "", ...)
```

Arguments

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

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 for more information
`axesLayout`	either `"serial"` or `"parallel"`
`showAxes`	boolean to indicate whether axes should be shown or not

axesColor	color of axes
showEnclosing	boolean, circle (axesLayout=radial) or sqaure (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

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 observartion. If the object is a factor then the labels get extracted with as.character.
label	label of a glyph (currently shown only in the glyph inspector)
...	state arguments

See Also

l_glyph_add

Examples

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

l_glyph_delete *Delete a Glyph*

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

l_glyph_getLabel *Get Glyph Label*

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

l_glyph_getType *Get Glyph Type*

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

l_glyph_ids *List glyphs ids*

Description

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

Usage

```
l_glyph_ids(widget)
```

Arguments

widget	widget path as a string or as an object handle

See Also

l_glyph_add

l_glyph_relabel *Relabel Glyph*

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

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

See Also

l_glyph_add, l_glyph_ids, l_glyph_getLabel

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_glyphs_inspector *Create a Glyphs Inspector*

Description

Inpectors 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

Inpectors 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

Inpectors 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

Inpectors 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

Inpectors 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_graph *Generic funtction to create an interactive graph display*

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 *Create a graph display based on node names and from-to edges list*

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

---

l_graph.graph *Create a graph display based on a graph object*

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 *Create a graph display based on a loongraph object*

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 loongraph 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_graph_inspector *Create a Graph Inspector*

Description

Inpectors 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

Inpectors 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

Inpectors 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_graphswitch *Create a graphswitch widget*

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`

`l_graphswitch_add` *Add a graph to a graphswitch widget*

Description

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

Usage

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

Arguments

`widget`	widget path as a string or as an object handle
`graph`	a graph or a loongraph object
`...`	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

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

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 methiod

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 methiod

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

widget	graphswitch widget handle (or widget path)
id	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

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

See Also

l_graphswitch

l_graphswitch_ids *List the ids of the graphs in the graphswitch widget*

Description

Every graph in the graphswitch widget has an id. This function returns these ids preserving the oder 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 *Move a Graph in the Graph List*

Description

Change the postion 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

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

widget	graphswitch widget handle (or widget path)
ids	vector with all graph ids from the graph widget. Use l_graphswitch_ids to query the ids.

See Also

l_graphswitch

l_graphswitch_set *Change the Graph shown in the Active Graph Widget*

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_help *Open a browser with loon's documentation webpage*

Description

`l_help` opens a browser with the relevant page on the official loon documentation website at http://waddella.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

Examples

```
## Not run:
l_help()
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` *Convert color names to their 12 digit hexadecimal color representation*

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` *Create an Interactive Histogram*

Description

Create an interactive histogram display that can be linked with loon's other displays

Usage

```
l_hist(x, origin = min(x), binwidth = NULL, parent = NULL, ...)
```

Arguments

`x`	vector with numerical data to perform the binning on
`origin`	scalar to define the binning origin
`binwidth`	scalar to specify the binwidth, if NULL then it is set to `diff(range(x))/30` if that value is >= 0.0001 or 0.0001 otherwise
`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$Sepal.Length, color=iris$Species)
```

l_hist_inspector *Create a Histogram Inspector*

Description

Inpectors 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

Inpectors 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_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

`array`	of 0-255 grayscale value data.
`width`	of images in pixels.
`height`	of images in pixels.
`img_in_row`	logical, TRUE if every row of the array represents an image
`invert`	logical, for 'invert=FALSE' 0=withe, for 'invert=TRUE' 0=black
`rotate`	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_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_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 includin plots, layers, navigators, contexts.

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.

Examples

```
p <- l_plot(iris, linkingGroup="iris")
i <- l_info_states(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 *Check if a widget path is a valid loon widget*

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` *Loon layers*

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	`l_layer_group`
polygon	one polygon	`l_layer_polygon`
text	one text string	`l_layer_text`
line	one line (i.e. connected line segments)	`l_layer_line`
rectangle	one rectangle	`l_layer_rectangle`
oval	one oval	`l_layer_oval`
points	n points (filled) circle	`l_layer_points`
texts	n text strings	`l_layer_text`
polygons	n polygons	`l_layer_polygons`
rectangles	n rectangles	`l_layer_rectangles`
lines	n sets of connected line segments	`l_layer_lines`

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
`l_layer_ids`	List layer ids
`l_layer_getType`	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
l_layer_getParent	Get parent layer id of a layer
l_layer_getChildren	Get children of a group layer
l_layer_index	Get the order index of a layer among its siblings
l_layer_printTree	Print out the layer tree
l_layer_move	Move a layer
l_layer_lower	Switch the layer place with its sibling to the right
l_layer_raise	Switch the layer place with its sibling to the left
l_layer_demote	Moves the layer up to be a left sibling of its parent
l_layer_promote	Moves the layer to be a child of its right group layer sibling
l_layer_hide	Set the layers visibility flag to FALSE
l_layer_show	Set the layers visibility flag to TRUE
l_layer_isVisible	Return visibility flag of layer
l_layer_layerVisibility	Returns logical value for whether layer is actually seen
l_layer_groupVisibility	Returns all, part or none for expressing which part of the layers children are visible.
l_layer_delete	Delete a layer. If the layer is a group move all its children layers to the layers parent.
l_layer_expunge	Delete layer and all its children layer.
l_layer_getLabel	Get layer label.
l_layer_relabel	Change layer label.
l_layer_bbox	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_scaleto_layer, l_scaleto_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_scaleto_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)
```

`l_layer.Line` *Layer line in Line object*

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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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 sp
`asSingleLayer`	If TRUE then prefer a single layer over groups with nested 1-dimensinal layers
`...`	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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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

`widget`	widget path as a string or as an object handle
`x`	a map object of class `map` as defined in the `maps` R package
`color`	fill color, if empty string `""`, then the fill is transparant
`linecolor`	outline color
`linewidth`	linewidth of outline
`label`	label used in the layers inspector
`parent`	parent widget path
`index`	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 methiod

Value

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

Examples

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

`l_layer.Polygon` *Layer polygon in Polygon object*

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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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

widget	widget widget path as a string or as an object handle
x	an object defined in the class sp
asSingleLayer	If TRUE then prefer a single layer over groups with nested 1-dimensinal layers
...	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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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

`widget`	widget widget path as a string or as an object handle
`x`	an object defined in the class sp
`asSingleLayer`	If TRUE then prefer a single layer over groups with nested 1-dimensinal layers
`...`	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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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

`widget`	widget widget path as a string or as an object handle
`x`	an object defined in the class sp
`asSingleLayer`	If `TRUE` then prefer a single layer over groups with nested 1-dimensinal layers
`...`	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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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

`widget`	widget widget path as a string or as an object handle
`x`	an object defined in the class sp
`asMainLayer`	if `TRUE` and the widget is a scatterplot widget, then points can be chosen to be added to the `'model'` layer
`...`	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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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

widget	widget widget path as a string or as an object handle
x	an object defined in the class sp
asMainLayer	if TRUE and the widget is a scatterplot widget, then points can be chosen to be added to the 'model' layer
...	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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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

`widget`	widget widget path as a string or as an object handle
`x`	an object defined in the class sp
`asSingleLayer`	If TRUE then prefer a single layer over groups with nested 1-dimensinal layers
`...`	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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

`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

`widget`	widget widget path as a string or as an object handle
`x`	an object defined in the class sp
`asSingleLayer`	If `TRUE` then prefer a single layer over groups with nested 1-dimensinal layers
`...`	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

```
library(sp)
library(rworldmap)

world <- getMap(resolution = "coarse")
p <- l_plot()
lmap <- l_layer(p, world, asSingleLayer=TRUE)
l_scaleto_world(p)
```

l_layer_bbox *Get the bounding box of a layer.*

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 **iff** 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 l_layer_line

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)

library(MASS)
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

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_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 *Moves the layer to be a child of its right group layer sibling*

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 *Delete a layer and all its descendants*

Description

Delete a group layer and all it's 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` *Get children of a group layer*

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)
```

`l_layer_getLabel` *Get layer label.*

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)
```

l_layer_getParent *Get parent layer id of a layer*

Description

The toplevel parent is the 'root' layer.

Usage

```
l_layer_getParent(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

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 *Get layer type*

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 *layer a group node*

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 decendants

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

Visibile 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 ancesters visibile 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

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 finite, non-missing and in (strictly) 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 above
z	a numeric or logical matrix containing the values to be plotted (NAs are allowed). Note that x can be used instead of z for convenience.
zlim	the minimum and maximum z values for which colors should be plotted, defaulting to the range of the finite values of z. Each of the given colors will be used to color an equispaced interval of this range. The *midpoints* of the intervals cover the range, so that values just outside the range will be plotted.
xlim	range for the plotted x values, defaulting to the range of x

ylim	range for the plotted y values, defaulting to the range of y
col	a list of colors such as that generated by rainbow, heat.colors, topo.colors, terrain.colors or similar functions.
breaks	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.
oldstyle	logical. If true the midpoints of the colour intervals are equally spaced, and zlim[1] and zlim[2] 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
...	argumnets forwarded to l_layer_line

Details

For more information run: `l_help("learn_R_layer.html#countourlines-heatimage-rasterimage")`

Value

layer id of group or rectangles layer

Examples

```
library(MASS)
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')
limg <- 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

Visibile 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 ancesters visibile 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)
```

`l_layer_ids` *List ids of layers in Plot*

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=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 = "#96BDFF", 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 = "#E2EDFF", linecolor="",
    label = "95 % prediction interval",
    parent = gLayer, index='end'
)

l_info_states(piLayer)
```

`l_layer_index` *Get the order index of a layer among its siblings*

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

`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 index for layers is 0 based.

Value

numeric value

See Also

`l_layer`, `l_layer_move`

`l_layer_isVisible` *Return visibility flag of layer*

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

`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

Visibile 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 ancesters visibile 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 on of 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

`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

Visibile 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 ancesters visibile 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`

`l_layer_line` *Layer a line*

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

`widget`	widget path name as a string
`x`	the coordinates of line. Alternatively, a single plotting structure, function or any *R* object with a plot method can be provided as x and y are passed on to `xy.coords`
`y`	the y coordinates of the line, optional if x is an appropriate structure.
`color`	color of line
`linewidth`	linewidth of outline
`dash`	dash pattern of line, see https://www.tcl.tk/man/tcl8.6/TkCmd/canvas.htm#M26
`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 `l_info_states`

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 `l_info_states`

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

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_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` *Move a layer*

Description

The postition 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` *Layer a oval*

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 transparant
`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 `l_info_states`

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` *Layer a points*

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 *R* object with a plot method can be provided as x and y are passed on to `xy.coords`
`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 `l_info_states`

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` *Layer a polygon*

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 transparant
`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 `l_info_states`

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 = "#96BDFF", 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 = "#E2EDFF", 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 transparant
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 l_info_states

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 *Print the layer tree*

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 *Moves the layer up to be a left sibling of its parent*

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 *Switch the layer place with its sibling to the left*

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

widget	widget path as a string or as an object handle
image	a raster object, or an object that can be coerced to one by as.raster.
xleft	a vector (or scalar) of left x positions.
ybottom	a vector (or scalar) of bottom y positions.
xright	a vector (or scalar) of right x positions.
ytop	a vector (or scalar) of top y positions.
angle	angle of rotation (in degrees, anti-clockwise from positive x-axis, about the bottom-left corner).
interpolate	a logical vector (or scalar) indicating whether to apply linear interpolation to the image when drawing.
parent	parent widget path
index	position among its siblings. valid values are 0, 1, 2, ..., 'end'
...	argumnets forwarded to l_layer_line

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)

# from examples(rasterImage)

# set up the plot region:
op <- par(bg = "thistle")
plot(c(100, 250), c(300, 450), type = "n", xlab = "", ylab = "")
image <- as.raster(matrix(0:1, ncol = 5, nrow = 3))
rasterImage(image, 100, 300, 150, 350, interpolate = FALSE)
rasterImage(image, 100, 400, 150, 450)
rasterImage(image, 200, 300, 200 + 10, 300 + 10,
            interpolate = FALSE)

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)
```

---

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 transparant
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 l_info_states

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 transparant
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 l_info_states

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` *Change layer label*

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` *Show or unhide a Layer*

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

Visibile 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 ancesters visibile 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` *Layer a text*

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 `l_info_states`

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` *Layer a texts*

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 *R* object with a plot method can be provided as x and y are passed on to `xy.coords`
`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 `l_info_states`

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_layers_inspector *Create a Layers Inspector*

Description

Inpectors 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_loon_inspector *Create a loon linspector*

Description

The loon inspector is a singleton widget that provids an overview to view and modify the active plot.

Usage

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

Arguments

parent	parent widget path
...	state arguments, see l_info_states.

Details

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

Value

widget handle

Examples

```
i <- l_loon_inspector()
```

`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 wile 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 wile 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

`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 wile 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` *Jitter Points Or Nodes*

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 *amount* (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 wile 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

`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 wile 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` *Vertically Align Points or Nodes*

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 wile 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

`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 wile 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 *Explore a dataset with the canonical 2d navigation graph setting*

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 *Add a Navigator to a Graph*

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_ids` *List Navigators*

Description

Lists all navigatora that belong to a graph

Usage

```
l_navigator_ids(widget)
```

Arguments

`widget`	graph widget

See Also

l_navigator_add

l_navigator_relabel *Modify the Label of a Navigator*

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

navigator	navigator handle
to	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

`navigator`	navigator handle
`to`	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

`navigator`	navigator handle
`path`	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

`tclobj`	a tcl object as returned by `tcl` and `.Tcl`
`transform`	a function to transfrom 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` *2d navigation graph setup with with dynamic node fitering using a scatterplot matrix*

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 *Select 2d spaces with variable associated measures displayed in scatterplot matrix*

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

```
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 carful that the vector names are correct
nav <- l_ng_plots(sapply(oliveAcids, q1), oliveAcids)
```

---

l_ng_plots.measures *2d Navigation Graph Setup with dynamic node fitering 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 fitering 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-calcultate 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

`measures`	objects from the `scagnostics` 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

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

```
library(scagnostics)
scags <- scagnostics(oliveAcids)

l_ng_plots(scags, oliveAcids, color=olive$Area)
```

l_ng_ranges *2d navigation graph setup with with dynamic node fitering using a slider*

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 *Select 2d spaces with variable associated measures using a slider*

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 fitering 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 fitering 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-calcultate 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 `scagnostics` 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

```
library(scagnostics)
s <- scagnostics(oliveAcids)
ng <- l_ng_ranges(s, oliveAcids, color=olive$Area)
```

l_pairs *Scatterplot Matrix in Loon*

Description

Function creates a scatterplot matrix using loon's scatterplot widgets

Usage

```
l_pairs(data, parent = NULL, ...)
```

Arguments

data	a data.frame with numerical data to create the scatterplot matrix
parent	parent widget path
...	named arguments to modify the scatterplot states

Value

a list with scatterplot handles

See Also

l_plot

Examples

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

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 plot. Alternatively, a single plotting structure, function or *any* R *object with a* plot *method* can be provided.
y	the y coordinates of points in the plot, *optional* if x is an appropriate structure.
...	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.

Value

widget handle

See Also

`l_info_states`

Examples

```
# ordinary use
p <- with(iris, l_plot(Sepal.Width, Petal.Length, color=Species))

# link another plot with the previous plot
p['linkingGroup'] <- "iris_data"
p2 <- with(iris, l_plot(Sepal.Length, Petal.Width, linkingGroup="iris_data"))

# Use with other tk widgets
library(tcltk)
tt <- tktoplevel()
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)

tktitle(tt) <- "Loon plots with custom layout"
```

`l_plot.default` *Create an interactive 2d scatterplot display*

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, 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 xy.coords for details. If supplied separately, they must be of the same length.
y	please read in the argument description for the x argument above.
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 tkpack or tkplace in order to be displayed. See the examples below.
...	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.

Examples

```
p1 <- with(iris, l_plot(Sepal.Length, Sepal.Width, color=Species))

p2 <- with(iris, l_plot(Petal.Length ~ Petal.Width, color=Species))

# link the two plots p1 and p2
l_configure(p1, linkingGroup="iris", sync="push")
l_configure(p2, linkingGroup="iris", sync="push")
p1['selected'] <- iris$Species == "setosa"
```

---

l_plot.map *Create an plot with a map layered*

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

```
library(maps)
p <- l_plot(map('world', fill=TRUE, plot=FALSE))
```

---

l_plot_inspector *Create a Scatterplot Inspector*

Description

Inpectors provide graphical user interfaces to oversee and modify plot states

Usage

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

Arguments

parent	parent widget path
...	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

Inpectors 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_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_scaleto_active *Change Plot Region to Display All Active Data*

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 *Change Plot Region to Display All Elements of a Particular Layer*

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 *Change Plot Region to Display the All Data of the Model Layer*

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 *Change Plot Region to Display All Selected Data*

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 *Change Plot Region to Display All Plot Data*

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 seerialaxes 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 for more information
`axesLayout`	either `"serial"` or `"parallel"`
`showAxes`	boolean to indicate whether axes should be shown or not
`parent`	parent widget path
`...`	state arguments, see `l_info_states`.

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 nxp 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)
```

l_serialaxes_inspector *Create a Serialaxes Analysis Inspector*

Description

Inpectors 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 calculted usding 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)
```

---

l_setColorList *Use custom colors for mapping nominal values to distinct colors*

Description

Modify loon's color mapping list to a set of custom colors.

Usage

```
l_setColorList(colors)
```

Arguments

colors	vecor 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 pallettes 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
library(scales)
p_custom <- with(olive, l_plot(stearic ~ oleic,
     color = 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_ColorBrewer, l_setColorList_hcl, l_setColorList_baseR

---

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", "Pasetl1",
  "Pastel2", "Paired", "Dark2", "Accent"))
```

Arguments

palette	one of the following RColorBrewer palette name: Set1, Set2, Set3, Pasetl1, Pastel2, Paired, Dark2, or Accent

Details

Only the following palettes in ColorBrewer are available: Set1, Set2, Set3, Pasetl1, Pastel2, Paired, Dark2, and Accent. See the examples below.

See Also

l_setColorList, l_setColorList_ColorBrewer, l_setColorList_hcl, l_setColorList_baseR

Examples

```
## Not run:
library(RColorBrewer)
display.brewer.all()

## End(Not run)

l_setColorList_ColorBrewer("Set1")
p <- l_plot(iris)
```

---

l_setColorList_hcl *Set loon's color mapping list to the colors from hcl color wheen*

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_ColorBrewer, l_setColorList_hcl, l_setColorList_baseR

`l_setLinkedStates` *Modify States of a Plot that are Linked in Loon's Standard Linking Model*

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 *used linkable* 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_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 width 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

`widget`	widget path as a string or as an object handle
`value`	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_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.

l_toR *Convert a Tcl Object to some other R object*

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

x	a tclObj object
cast	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.

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 *Create a Worldview Inspector*

Description

Inpectors 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 *Zoom from and towards the center*

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

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 lingraph 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 methiod

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)
```

---

loon *loon: A Toolkit for Interactive Data Visualization and Exploration*

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://waddella.github.io/loon/learn_R_intro.html.

Author(s)

Maintainer: Adrian Waddell <adrian@waddell.ch>

Authors:

- R. Wayne Oldford <rwoldford@uwaterloo.ca>

See Also

Useful links:

- http://waddella.github.io/loon/

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)
```

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

nodes	a character vector with node names, each element defines a node hence the elements need to be unique
from	a character vector with node names, each element defines an edge
to	a character vector with node names, each element defines an edge
isDirected	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)
```

---

make_glyphs *Make arbitrary glyphs with R graphic devices*

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

```
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:
data(minority)
p <- l_plot(minority$long, minority$lat)

library(maps)
canada <- map("world",  "Canada", fill=TRUE, plot=FALSE)
l_map <- l_layer(p, canada, asSingleLayer=TRUE)
l_scaleto_world(p)

img <- 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
li <- make_glyphs(runif(6), function(x) {
    if(any(x>1 | x<0))
        stop("out of range")
    pushViewport(plotViewport(unit(c(1,1,1,1)*0, "points")))
    grid.rect(gp=gpar(fill=NA))
    grid.rect(0, 0, height = unit(x, "npc"), just = c("left", "bottom"),
```

```
              gp=gpar(col=NA, fill="steelblue"))
})

## End(Not run)
```

`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

`data`	a data.frame with the data used to calculate the measures
`...`	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

data	a data.frame with the data used to calculate the measures
...	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

This data contains information about the visible minority populations distributed across major census metropolitan areas of Canada. These data are from the 2006 Canadian census, publicly available from Statistics Canada Statistics Canada (2006). For each of the 33 Canadian census metropolitan areas, we have the total population and the population Implemen- tation of all its "visible minorities". These self-declared visible minorities are: "Arab", "Black", "Chinese", "Filipino", "Japanese", "Korean", "Latin American", "Multiple visible minority", "South Asian", "Southeast Asian", "Visible minority (not included elsewhere)", and "West Asian". For each metropolitan area, we also obtained the approximate latitude and longitude coordinates using the Google Maps Geocoding API and added them to the data set.

Usage

```
minority
```

Format

A data frame with 33 rows and 18 variates

Source

http://www.statcan.gc.ca

ndtransitiongraph *Create a n-d transition graph*

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 (palmitic, palmitoleic, stearic, oleic, linoleic, linolenic, arachidic, eicosenoic) 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:

Region	Area
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.

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.

---

oliveAcids *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

`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

```
library(Rgraphviz)
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 methiod

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 *Print function names from measure2d object*

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 print.default

scagnostics2d *Closure of Two Dimensional Scagnostic Measures*

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 meausure 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

```
m <- scagnostics2d(oliveAcids, separator='**')
m
m()
m(olive$palmitoleic > 80)
m('data')
m('grid')
m('measures')
```

---

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

# Lets 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 = loon:::hex12tohex6(l_hexcolor(colors())),
    row.names = colors(),
    stringsAsFactors = FALSE
```

```
)

df_diff <- df[df$R_col != df$Tcl_col,]

library(grid)
grid.newpage()
pushViewport(plotViewport())

x_col <- unit(0, "npc")
x_R <- unit(6, "lines")
x_Tcl <- unit(10, "lines")

grid.text('color', x=x_col, y=unit(1, "npc"), just='left', gp=gpar(fontface='bold'))
grid.text('R', x=x_R, y=unit(1, "npc"), just='center', gp=gpar(fontface='bold'))
grid.text('Tcl', x=x_Tcl, y=unit(1, "npc"), just='center', gp=gpar(fontface='bold'))
for (i in 1:nrow(df_diff)) {
    y <- unit(1, "npc") - unit(i*1.2, "lines")
    grid.text(rownames(df_diff)[i], x=x_col, y=y, just='left')
    grid.rect(x=x_R, y=y, width=unit(3, "line"),
              height=unit(1, "line"), gp=gpar(fill=df_diff[i,1]))
    grid.rect(x=x_Tcl, y=y, width=unit(3, "line"),
              height=unit(1, "line"), gp=gpar(fill=df_diff[i,2]))
}
```

UsAndThem *Data to re-create Hans Rosling's fameous "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

Source

http://www.gapminder.org/

Index