Package 'ggvis'

October 2, 2020

Title Interactive Grammar of Graphics

Description An implementation of an interactive grammar of graphics, taking the best parts of 'ggplot2', combining them with the reactive framework of 'shiny' and drawing web graphics using 'vega'.

Version 0.4.6

URL https://ggvis.rstudio.com/

Depends R (>= 3.0)

Imports assertthat, jsonlite (>= 0.9.11), shiny (>= 0.11.1), magrittr, dplyr (>= 0.5.0), lazyeval, htmltools (>= 0.2.4), methods

Suggests MASS, mgcv, lubridate, testthat (>= 0.8.1), knitr (>= 1.6), rmarkdown

Encoding UTF-8

License GPL-2 | file LICENSE

LazyData true

RoxygenNote 7.1.1

NeedsCompilation no

Author Winston Chang [aut, cre],
Hadley Wickham [aut],
RStudio [cph],
jQuery Foundation [cph] (jQuery library and jQuery UI library),
jQuery contributors [ctb, cph] (jQuery library; authors listed in inst/www/lib/jquery/AUTHORS.txt),
jQuery UI contributors [ctb, cph] (jQuery UI library; authors listed in inst/www/lib/jquery-ui/AUTHORS.txt),
Mike Bostock [ctb, cph] (D3 library),
D3 contributors [ctb] (D3 library; authors listed at https://github.com/d3/d3/graphs/contributors),
Trifacta Inc. [cph] (Vega library),
Vega contributors [ctb] (Vega library; authors listed at https://github.com/trifacta/vega/graphs/contributors),
Sebastián Décima [ctb, cph] (javascript-detect-element-resize library)

Maintainer Winston Chang <winston@rstudio.com>

Repository CRAN

Date/Publication 2020-10-02 00:10:02 UTC

R topics documented:

`add_axis` *Add a vega axis specification to a ggvis plot*

Description

Axis specifications allow you to either override the default axes, or additional axes.

Usage

```
add_axis(
  vis,
  type,
  scale = NULL,
  orient = NULL,
  title = NULL,
  title_offset = NULL,
  format = NULL,
  ticks = NULL,
```

```
  values = NULL,
  subdivide = NULL,
  tick_padding = NULL,
  tick_size_major = NULL,
  tick_size_minor = tick_size_major,
  tick_size_end = tick_size_major,
  offset = NULL,
  layer = "back",
  grid = TRUE,
  properties = NULL
)

hide_axis(vis, scale)
```

Arguments

vis	A ggvis object.
type	The type of axis. Either x or y.
scale	The name of the scale backing the axis component. Defaults to the scale type - you will need to specify if you want (e.g.) a scale for a secondary y-axis.
orient	The orientation of the axis. One of top, bottom, left or right. The orientation can be used to further specialize the axis type (e.g., a y axis oriented for the right edge of the chart) - defaults to bottom for x axes, and left for y axes.
title	A title for the axis. By default, it uses the name of the field in the first data set used by the scale. Use "" to suppress the title.
title_offset	The offset (in pixels) from the axis at which to place the title.
format	The formatting pattern for axis labels. Vega uses D3's format pattern.
ticks	A desired number of ticks. The resulting number may be different so that values are "nice" (multiples of 2, 5, 10) and lie within the underlying scale's range.
values	Explicitly set the visible axis tick values.
subdivide	If provided, sets the number of minor ticks between major ticks (the value 9 results in decimal subdivision).
tick_padding	The padding, in pixels, between ticks and text labels.
tick_size_major, tick_size_minor, tick_size_end	The size, in pixels, of major, minor and end ticks.
offset	The offset, in pixels, by which to displace the axis from the edge of the enclosing group or data rectangle.
layer	A string indicating if the axis (and any gridlines) should be placed above or below the data marks. One of "front" or "back" (default).
grid	A flag indicating if gridlines should be created in addition to ticks.
properties	Optional mark property definitions for custom axis styling. Should be an object created by axis_props, with properties for ticks, majorTicks, minorTicks, grid, labels, title, and axis.

Details

More information about axes can be found in the "axes and legends" vignettes.

Compared to ggplot2

In ggplot2, axis (and legend) properties are part of the scales specification. In vega, they are separate, which allows the specification of multiple axes, and more flexible linkage between scales and axes.

See Also

Vega axis documentation: https://github.com/trifacta/vega/wiki/Axes

Examples

```
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill = ~cyl) %>%
  layer_points() %>%
  add_axis("x", title = "Weight", orient = "top")

# Suppress axis with hide_axis
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill = ~cyl) %>%
  layer_points() %>%
  hide_axis("x") %>% hide_axis("y")

mtcars %>% ggvis(x = ~wt, y = ~mpg) %>% layer_points() %>%
  add_axis("x", title = "Weight", ticks = 40,
    properties = axis_props(
      ticks = list(stroke = "red"),
      majorTicks = list(strokeWidth = 2),
      grid = list(stroke = "red"),
      labels = list(
        fill = "steelblue",
        angle = 50,
        fontSize = 14,
        align = "left",
        baseline = "middle",
        dx = 3
      ),
      title = list(fontSize = 16),
      axis = list(stroke = "#333", strokeWidth = 1.5)
    )
  )
```

---

add_data *Add dataset to a visualisation*

Description

Add dataset to a visualisation

Usage

```
add_data(vis, data, name = deparse2(substitute(data)), add_suffix = TRUE)
```

Arguments

vis	Visualisation to modify.
data	Data set to add.
name	Data of data - optional, but helps produce informative error messages.
add_suffix	Should a unique suffix be added to the data object's ID? This should only be FALSE when the spec requires a data set with a specific name.

Examples

```
mtcars %>% ggvis(~mpg, ~wt) %>% layer_points()
NULL %>% ggvis(~mpg, ~wt) %>% add_data(mtcars) %>% layer_points()
```

---

add_guide_axis *Defunct function for adding an axis*

Description

This function has been replaced with add_axis.

Usage

```
add_guide_axis(...)
```

Arguments

...	Other arguments.

---

add_guide_legend *Defunct function for adding a legend*

Description

This function has been replaced with add_legend.

Usage

```
add_guide_legend(...)
```

Arguments

...	Other arguments.

add_legend *Add a vega legend specification to a ggvis plot*

Description

Axis specifications allow you to either override the default legends, or supply additional legends.

Usage

```
add_legend(
  vis,
  scales = NULL,
  orient = "right",
  title = NULL,
  format = NULL,
  values = NULL,
  properties = NULL
)

hide_legend(vis, scales)
```

Arguments

vis	A ggvis object.
scales	The name of one or more scales for which to add a legend. Typically one of "size", "shape", "fill", "stroke", although custom scale names may also be used. Multiple names can also be used, like `c("fill","shape")`.
orient	The orientation of the legend. One of "left" or "right". This determines how the legend is positioned within the scene. The default is "right".
title	A title for the legend. By default, it uses the name the fields used in the legend. Use `""` to suppress the title.
format	The formatting pattern for axis labels. Vega uses D3's format pattern.
values	Explicitly set the visible legend values.
properties	Optional mark property definitions for custom legend styling. Should be an object created by `legend_props`, with properties for title, label, symbols, gradient, legend.

Details

More information about axes can be found in the "axes and legends" vignettes.

Compared to ggplot2

In ggplot2, legend (and axis) properties are part of the scales specification. In vega, they are separate, which allows the specification of multiple legends, and more flexible linkage between scales and legends.

Examples

```
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill = ~cyl) %>%
  layer_points() %>%
  add_legend("fill", title = "Cylinders")

# Suppress legend with hide_legend
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill = ~cyl) %>%
  layer_points() %>%
  hide_legend("fill")

# Combining two properties in one legend
mtcars %>%
  ggvis(x = ~wt, y = ~mpg, fill = ~factor(cyl), shape = ~factor(cyl)) %>%
  layer_points() %>%
  add_legend(c("fill", "shape"))

# Control legend properties with a continuous legend, with x and y position
# in pixels.
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill = ~cyl) %>%
  layer_points() %>%
  add_legend("fill", title = "Cylinders",
    properties = legend_props(
      title = list(fontSize = 16),
      labels = list(fontSize = 12, fill = "#00F"),
      gradient = list(stroke = "red", strokeWidth = 2),
      legend = list(x = 500, y = 50)
    )
  )

# Control legend properties with a categorical legend, with x and y position
# in the scaled data space.
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill = ~factor(cyl)) %>%
  layer_points() %>%
  add_legend("fill", title = "Cylinders",
    properties = legend_props(
      title = list(fontSize = 16),
      labels = list(fontSize = 14, dx = 5),
      symbol = list(stroke = "black", strokeWidth = 2,
        shape = "square", size = 200),
      legend = list(
        x = scaled_value("x", 4.5),
        y = scaled_value("y", 30)
      )
    )
  )

# Control legend position using x_rel and y_rel which specify relative
# position, going from 0 to 1. (0, 0) is the bottom-left corner, and
# (1, 1) is the upper-right corner. The values control the position of
# the upper-left corner of the legend.
mtcars %>% ggvis(x = ~wt, y = ~mpg, fill = ~cyl) %>%
  layer_points() %>%
```

```
  add_relative_scales() %>%
  add_legend("fill", title = "Cylinders",
    properties = legend_props(
      legend = list(
        x = scaled_value("x_rel", 0.8),
        y = scaled_value("y_rel", 1)
      )
    )
  )
```

---

add_props *Add visual properties to a visualisation*

Description

Add visual properties to a visualisation

Usage

```
add_props(vis, ..., .props = NULL, inherit = NULL, env = parent.frame())
```

Arguments

vis	Visualisation to modify.
...	A set of name-value pairs. The name should be a valid vega property. The first two unnamed components are taken to be x and y. Any additional unnamed components will raise an error.
.props	When calling props from other functions, you'll often have a list of quoted function functions. You can pass that function to the .props argument instead of messing around with substitute. In other words, .props lets you opt out of the non-standard evaluation that props does.
inherit	If TRUE, the defaults, will inherit from properties from the parent layer If FALSE, it will start from nothing.
env	The environment in which to evaluate variable properties.

Examples

```
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points()
mtcars %>% ggvis() %>% add_props(~wt, ~mpg) %>% layer_points()
mtcars %>% ggvis(~wt) %>% add_props(y = ~mpg) %>% layer_points()
```

add_relative_scales *Add x_rel and y_rel scales*

Description

This function adds scales named `x_rel` and `y_rel`, each of which has a domain of 0 to 1, and the range is the plot's width or height. These scales are useful for positioning visual elements relative to the plotting area. For example, with legends.

Usage

```
add_relative_scales(vis)
```

Arguments

`vis`	A ggvis object.

See Also

`add_legend` for a usage example.

add_tooltip *Add tooltips to a plot.*

Description

Add tooltips to a plot.

Usage

```
add_tooltip(vis, html, on = c("hover", "click"))
```

Arguments

`vis`	Visualisation to add tooltips to.
`html`	A function that takes a single argument as input. This argument will be a list containing the data in the mark currently under the mouse. It should return a string containing HTML or `NULL` to hide tooltip for the current element.
`on`	Should tooltips appear on hover, or on click?

Examples

```
## Run these examples only in interactive R sessions
if (interactive()) {

all_values <- function(x) {
  if(is.null(x)) return(NULL)
  paste0(names(x), ": ", format(x), collapse = "<br />")
}

base <- mtcars %>% ggvis(x = ~wt, y = ~mpg) %>%
  layer_points()
base %>% add_tooltip(all_values, "hover")
base %>% add_tooltip(all_values, "click")

# The data sent from client to the server contains only the data columns that
# are used in the plot. If you want to get other columns of data, you should
# to use a key to line up the item from the plot with a row in the data.
mtc <- mtcars
mtc$id <- 1:nrow(mtc)  # Add an id column to use ask the key

all_values <- function(x) {
  if(is.null(x)) return(NULL)
  row <- mtc[mtc$id == x$id, ]
  paste0(names(row), ": ", format(row), collapse = "<br />")
}

mtc %>% ggvis(x = ~wt, y = ~mpg, key := ~id) %>%
  layer_points() %>%
  add_tooltip(all_values, "hover")

}
```

`auto_group` *Automatically group data by grouping variables*

Description

Use `auto_group` to group up a dataset on all categorical variables specified by props, and have each piece rendered by the same mark.

Usage

```
auto_group(vis, exclude = NULL)
```

Arguments

`vis`	The ggvis visualisation to modify.
`exclude`	A vector containing names of props to exclude from auto grouping. It is often useful to exclude `c("x","y")`, when one of those variables is categorical.

See Also

To manually specify grouping variables, see group_by.

Examples

```
# One line
mtcars %>% ggvis(~disp, ~mpg, stroke = ~factor(cyl)) %>% layer_paths()
# One line for each level of cyl
mtcars %>% ggvis(~disp, ~mpg, stroke = ~factor(cyl)) %>% group_by(cyl) %>%
  layer_paths()
mtcars %>% ggvis(~disp, ~mpg, stroke = ~factor(cyl)) %>% auto_group() %>%
  layer_paths()

# The grouping column can already be stored as a factor
mtcars2 <- mtcars
mtcars2$cyl <- factor(mtcars2$cyl)
mtcars2 %>% ggvis(~disp, ~mpg, stroke = ~cyl) %>% auto_group() %>%
  layer_paths()
```

---

axis_props *Create an axis_props object for controlling axis properties.*

Description

The items in each of the lists can be a literal value, like 5 or "blue", or they can be a scaled_value object.

Usage

```
axis_props(
  ticks = NULL,
  majorTicks = NULL,
  minorTicks = NULL,
  grid = NULL,
  labels = NULL,
  title = NULL,
  axis = NULL
)
```

Arguments

ticks	A named list of line properties for ticks.
majorTicks	A named list of line properties for major ticks.
minorTicks	A named list of line properties for minor ticks.
grid	A named list of line properties for grid lines.
labels	A named list of text properties for axis labels.
title	A named list of text properties for the axis title.
axis	A named list of line properties for the axis line.

band *A band*

Description

Bands are used to set the width or height on categorical scales - a band represent the height or width allocated for one level of a factor.

Usage

```
band(offset = NULL, mult = NULL)

is.prop_band(x)
```

Arguments

offset, mult	Additive and multiplicate offsets used to adjust the band size. For example, use mult = 0.9 to make a bar take up 90% of the space allocated for its category.
x	object to test for band-ness

Examples

```
df <- data.frame(label = c("a", "b", "c"), n = c(10, 9, 4))

base <- df %>% ggvis(~label, y2 = 0, y = ~n)
base %>% layer_rects(width = band())
base %>% layer_rects(width = band(offset = -1))
base %>% layer_rects(width = band(mult = 0.9))

# A nominal scale with padding is more symmetrical than band with a mult
base %>% layer_rects(width = band(mult = 0.75))
base %>% layer_rects(width = band()) %>%
  scale_nominal("x", padding = 0.25, points = FALSE)
```

cocaine *Cocaine seizures in the US.*

Description

This dataset comes from STRIDE, the System to Retrieve Information from Drug Evidence. It contains all concaine seizures in the US from 2007 that have a known weight.

Usage

```
cocaine
```

Format

Data frame with 3380 observations of 5 variables.

Variables

state State where seizure occured.

potency Purity of cocaine, as percentage (100% = pure cocaine, 0% = all filler)

weight Weight, in grams.

month Month in which seizure occured.

price Estimated value in USD.

Use

Use of this data requires your agreement to refer to your analyses as "unvalidated DEA data and to claim authorship and responsibility for any inferences and/or conclusions you may draw from this information."

---

compute_align *Align positions using length.*

Description

This compute function is often used in conjunction with `compute_count`, when used on data with a continuous x variable. By default, the computed width will be equal to the resolution of the data, or, in other words the smallest difference between two values in the data.

Usage

```
compute_align(x, var, length = NULL, align = 0.5, dir = "x")
```

Arguments

`x`	Dataset-like object to align. Built-in methods for data frames, grouped data frames and ggvis visualisations.
`var`	Name of variable to compute width of.
`length`	An absolute length to use. If `NULL` (the default), the width will be equivalent to the resolution of the data.
`align`	Where does the existing variable fall on the new bins? 0 = left edge, 0.5 = center, 1 = right edge.
`dir`	Direction, i.e. `"x"` or `"y"`. Used to generate variable names in output.

Details

An absolute width for each x can be specified by using the `width` argument. If `width` is NULL (the default), it will use the resolution of the data as the width.

Value

The original data frame, with additional columns:

'dir'min_	left boundary of bin
'dir'max_	right boundary of bin
'dir'len_	width of bin

See Also

compute_bin For counting cases within ranges of a continuous variable.

compute_count For counting cases at specific values of a variable.

Examples

```
mtcars %>% compute_count(~disp) %>% compute_align(~x_)
mtcars %>% compute_count(~mpg) %>% compute_align(~x_)

# Use a specific width
pressure %>% compute_count(~temperature) %>% compute_align(~x_)
pressure %>% compute_count(~temperature) %>% compute_align(~x_, length = 5)

# It doesn't matter whether you transform inside or outside of a vis
mtcars %>% compute_count(~cyl, ~wt) %>%
  compute_align(~x_, length = .5) %>%
  ggvis(x = ~xmin_, x2 = ~xmax_, y = ~count_, y2 = 0) %>%
  layer_rects()

mtcars %>%
  ggvis(x = ~xmin_, x2 = ~xmax_, y = ~count_, y2 = 0) %>%
  compute_count(~cyl, ~wt) %>%
  compute_align(~x_) %>%
  layer_rects()

# Varying align
mtcars %>%
  ggvis(x = ~xmin_, x2 = ~xmax_, y = ~count_, y2 = 0) %>%
  compute_count(~cyl, ~wt) %>%
  compute_align(~x_, length = 0.5, align = input_slider(0, 1)) %>%
  layer_rects()
```

---

compute_bin *Bin data along a continuous variable*

Description

Bin data along a continuous variable

Usage

```
compute_bin(
  x,
  x_var,
  w_var = NULL,
  width = NULL,
  center = NULL,
  boundary = NULL,
  closed = c("right", "left"),
  pad = FALSE,
  binwidth
)
```

Arguments

x	Dataset-like object to bin. Built-in methods for data frames, grouped data frames and ggvis visualisations.
x_var, w_var	Names of x and weight variables. The x variable must be continuous.
width	The width of the bins. The default is NULL, which yields 30 bins that cover the range of the data. You should always override this value, exploring multiple widths to find the best to illustrate the stories in your data.
center	The center of one of the bins. Note that if center is above or below the range of the data, things will be shifted by an appropriate number of widths. To center on integers, for example, use width=1 and center=0, even if 0 is outside the range of the data. At most one of center and boundary may be specified.
boundary	A boundary between two bins. As with center, things are shifted when boundary is outside the range of the data. For example, to center on integers, use width = 1 and boundary = 0.5, even if 1 is outside the range of the data. At most one of center and boundary may be specified.
closed	One of "right" or "left" indicating whether right or left edges of bins are included in the bin.
pad	If TRUE, adds empty bins at either end of x. This ensures frequency polygons touch 0. Defaults to FALSE.
binwidth	Deprecated; use width instead.

Value

A data frame with columns:

count_	the number of points
x_	mid-point of bin
xmin_	left boundary of bin
xmax_	right boundary of bin
width_	width of bin

See Also

compute_count For counting cases at specific locations of a continuous variable. This is useful when the variable is continuous but the data is granular.

Examples

```
mtcars %>% compute_bin(~mpg)
mtcars %>% compute_bin(~mpg, width = 10)
mtcars %>% group_by(cyl) %>% compute_bin(~mpg, width = 10)

# It doesn't matter whether you transform inside or outside of a vis
mtcars %>% compute_bin(~mpg) %>% ggvis(~x_, ~count_) %>% layer_paths()
mtcars %>% ggvis(~ x_, ~ count_) %>% compute_bin(~mpg) %>% layer_paths()

# Missing values get own bin
mtcars2 <- mtcars
mtcars2$mpg[sample(32, 5)] <- NA
mtcars2 %>% compute_bin(~mpg, width = 10)

# But are currently silently dropped in histograms
mtcars2 %>% ggvis() %>% layer_histograms(~mpg)
```

---

compute_boxplot *Calculate boxplot values*

Description

Calculate boxplot values

Usage

```
compute_boxplot(x, var = NULL, coef = 1.5)
```

Arguments

x	Dataset-like object to compute boxplot values. There are built-in methods for data frames, grouped data frames, and ggvis visualisations.
var	Name of variable for which to compute boxplot values. The variable must be continuous.
coef	The maximum length of the whiskers as multiple of the inter-quartile range. Default value is 1.5.

Value

A data frame with columns:

min_	Lower whisker = smallest observation greater than or equal to lower hinge - 1.5 * IQR

lower_	Lower hinge (25th percentile)
median_	Median (50th percentile)
upper_	Upper hinge (75th percentile)
max_	Upper whisker = largest observation less than or equal to upper hinge + 1.5 * IQR
outliers_	A vector of values that are outside of the min and max

See Also

layer_boxplots

Examples

```
mtcars %>% compute_boxplot(~mpg)
mtcars %>% group_by(cyl) %>% compute_boxplot(~mpg)
```

---

compute_count *Count data at each location*

Description

Count data at each location

Usage

```
compute_count(x, x_var, w_var = NULL)
```

Arguments

x	Dataset-like object to count. Built-in methods for data frames, grouped data frames and ggvis visualisations.
x_var, w_var	Names of x and weight variables.

Value

A data frame with columns:

count_	the number of points
x_	the x value where the count was made

The width of each "bin" is set to the resolution of the data – that is, the smallest difference between two x values.

See Also

compute_bin For counting cases within ranges of a continuous variable.

compute_align For calculating the "width" of data.

Examples

```
mtcars %>% compute_count(~cyl)

# Weight the counts by car weight value
mtcars %>% compute_count(~cyl, ~wt)

# If there's one weight value at each x, it effectively just renames columns.
pressure %>% compute_count(~temperature, ~pressure)
# Also get the width of each bin
pressure %>% compute_count(~temperature, ~pressure) %>% compute_align(~x_)

# It doesn't matter whether you transform inside or outside of a vis
mtcars %>% compute_count(~cyl, ~wt) %>%
  compute_align(~x_) %>%
  ggvis(x = ~xmin_, x2 = ~xmax_, y = ~count_, y2 = 0) %>%
  layer_rects()

mtcars %>%
  ggvis(x = ~xmin_, x2 = ~xmax_, y = ~count_, y2 = 0) %>%
  compute_count(~cyl, ~wt) %>%
  compute_align(~x_) %>%
  layer_rects()
```

---

compute_density *Compute density of data.*

Description

Compute density of data.

Usage

```
compute_density(
  x,
  x_var,
  w_var = NULL,
  kernel = "gaussian",
  trim = FALSE,
  n = 256L,
  na.rm = FALSE,
  ...
)
```

Arguments

x	Dataset (data frame, grouped_df or ggvis) object to work with.
x_var, w_var	Names of variables to use for x position, and for weights.
kernel	Smoothing kernel. See density for details.

trim	If TRUE, the default, density estimates are trimmed to the actual range of the data. If FALSE, they are extended by the default 3 bandwidths (as specified by the cut parameter to density).
n	Number of points (along x) to use in the density estimate.
na.rm	If TRUE missing values will be silently removed, otherwise they will be removed with a warning.
...	Additional arguments passed on to density.

Value

A data frame with columns:

pred_	regularly spaced grid of n locations
resp_	density estimate

Examples

```
mtcars %>% compute_density(~mpg, n = 5)
mtcars %>% group_by(cyl) %>% compute_density(~mpg, n = 5)
mtcars %>% ggvis(~mpg) %>% compute_density(~mpg, n = 5) %>%
  layer_points(~pred_, ~resp_)
```

---

compute_model_prediction

Create a model of a data set and compute predictions.

Description

Fit a 1d model, then compute predictions and (optionally) standard errors over an evenly spaced grid.

Usage

```
compute_model_prediction(
  x,
  formula,
  ...,
  model = NULL,
  se = FALSE,
  level = 0.95,
  n = 80L,
  domain = NULL,
  method
)

compute_smooth(x, formula, ..., span = 0.75, se = FALSE)
```

Arguments

x	Dataset-like object to model and predict. Built-in methods for data frames, grouped data frames and ggvis visualisations.
formula	Formula passed to modelling function. Can use any variables from data.
...	arguments passed on to model function
model	Model fitting function to use - it must support R's standard modelling interface, taking a formula and data frame as input, and returning predictions with predict. If not supplied, will use loess for <= 1000 points, otherwise it will use gam. Other modelling functions that will work include lm, glm and rlm.
se	include standard errors in output? Requires appropriate method of predict_grid, since the interface for returning predictions with standard errors is not consistent acrossing modelling frameworks.
level	the confidence level of the standard errors.
n	the number of grid points to use in the prediction
domain	If NULL (the default), the domain of the predicted values will be the same as the domain of the prediction variable in the data. It can also be a two-element numeric vector specifying the min and max.
method	Deprecated. Please use model instead.
span	Smoothing span used for loess model.

Details

compute_model_prediction fits a model to the data and makes predictions with it. compute_smooth is a special case of model predictions where the model is a smooth loess curve whose smoothness is controlled by the span parameter.

Value

A data frame with columns:

resp_	regularly spaced grid of n locations
pred_	predicted value from model
pred_lwr_ and pred_upr_	upper and lower bounds of confidence interval (if se = TRUE)
pred_se_	the standard error (width of the confidence interval) (if se = TRUE)

Examples

```
# Use a small value of n for these examples
mtcars %>% compute_model_prediction(mpg ~ wt, n = 10)
mtcars %>% compute_model_prediction(mpg ~ wt, n = 10, se = TRUE)
mtcars %>% group_by(cyl) %>% compute_model_prediction(mpg ~ wt, n = 10)

# compute_smooth defaults to loess
mtcars %>% compute_smooth(mpg ~ wt)
```

```
# Override model to suppress message or change approach
mtcars %>% compute_model_prediction(mpg ~ wt, n = 10, model = "loess")
mtcars %>% compute_model_prediction(mpg ~ wt, n = 10, model = "lm")

# Set the domain manually
mtcars %>%
  compute_model_prediction(mpg ~ wt, n = 20, model = "lm", domain = c(0, 8))

# Plot the results
mtcars %>% compute_model_prediction(mpg ~ wt) %>%
  ggvis(~pred_, ~resp_) %>%
  layer_paths()
mtcars %>% ggvis() %>%
  compute_model_prediction(mpg ~ wt) %>%
  layer_paths(~pred_, ~resp_)
```

`compute_stack` *Stack overlapping data.*

Description

Stack overlapping data.

Usage

```
compute_stack(x, stack_var = NULL, group_var = NULL)
```

Arguments

`x`	A data object
`stack_var`	A string specifying the stacking variable.
`group_var`	A string specifying the grouping variable.

Value

A data frame with columns:

`stack_upr_`	the lower y coordinate for a stack bar
`stack_lwr_`	the upper y coordinate for a stack bar

Examples

```
mtcars %>% cbind(count = 1) %>% compute_stack(~count, ~cyl)

# Shouldn't use or affect existing grouping
mtcars %>% cbind(count = 1) %>% group_by(am) %>% compute_stack(~count, ~cyl)

# If given a ggvis object, will use x variable for stacking by default
mtcars %>% ggvis(x = ~cyl, y = ~wt) %>%
```

```
  compute_stack(stack_var = ~wt, group_var = ~cyl) %>%
  layer_rects(x = ~cyl - 0.5, x2 = ~cyl + 0.5, y = ~stack_upr_,
    y2 = ~stack_lwr_)

# Collapse across hair & eye colour data across sex
hec <- as.data.frame(xtabs(Freq ~ Hair + Eye, HairEyeColor))
hec %>% compute_stack(~Freq, ~Hair)

# Without stacking - bars overlap
hec %>% ggvis(~Hair, ~Freq, fill = ~Eye, fillOpacity := 0.5) %>%
  layer_rects(y2 = 0, width = band())

# With stacking
hec %>% ggvis(x = ~Hair, y = ~Freq, fill = ~Eye, fillOpacity := 0.5) %>%
  compute_stack(~Freq, ~Hair) %>%
  layer_rects(y = ~stack_lwr_, y2 = ~stack_upr_, width = band())

# layer_bars stacks automatically:
hec %>% ggvis(~Hair, ~Freq, fill = ~Eye, fillOpacity := 0.5) %>%
  group_by(Eye) %>%
  layer_bars(width = 1)
```

`compute_tabulate` *Count data at each location of a categorical variable*

Description

Count data at each location of a categorical variable

Usage

```
compute_tabulate(x, x_var, w_var = NULL)
```

Arguments

`x`	Dataset-like object to count. Built-in methods for data frames, grouped data frames and ggvis visualisations.
`x_var, w_var`	Names of x and weight variables.

Value

A data frame with columns:

`count_`	the number of points
`x_`	value of bin

See Also

compute_bin For counting cases within ranges of a continuous variable.

compute_count For counting cases at specific locations of a continuous variable. This is useful when the variable is continuous but the data is granular.

Examples

```
library(dplyr)
# The tabulated column must be countable (not numeric)
## Not run: mtcars %>% compute_tabulate(~cyl)
mtcars %>% mutate(cyl = factor(cyl)) %>% compute_tabulate(~cyl)

# Or equivalently:
mtcars %>% compute_tabulate(~factor(cyl))

# If there's one weight value at each x, it effectively just renames columns.
pressure %>% compute_tabulate(~factor(temperature), ~pressure)

# It doesn't matter whether you transform inside or outside of a vis
mtcars %>% compute_tabulate(~factor(cyl)) %>%
  ggvis(x = ~x_, y = ~count_, y2 = 0) %>%
  layer_rects(width = band())

mtcars %>%
  ggvis(x = ~x_, y = ~count_, y2 = 0) %>%
  compute_tabulate(~factor(cyl)) %>%
  layer_rects(width = band())

# compute_tabulate is used automatically in layer_bars when no y prop
# is supplied.
mtcars %>% ggvis(x = ~factor(cyl)) %>% layer_bars()
```

---

explain *Explain details of an object*

Description

This is a generic function which gives more details about an object than print, and is more focussed on human readable output than str.

See Also

dplyr::explain for more information.

Examples

```
p <- mtcars %>% ggvis(x = ~cyl) %>% layer_bars()
explain(p)
```

explain.ggvis *Print out the structure of a ggvis object in a friendly format*

Description

Print out the structure of a ggvis object in a friendly format

Usage

```
## S3 method for class 'ggvis'
explain(x, ...)
```

Arguments

x	Visualisation to explain
...	Needed for compatibility with generic. Ignored by this method.

export_png *Export a PNG or SVG from a ggvis object*

Description

This requires that the external program vg2png is installed. This is part of the vega node.js module.

Usage

```
export_png(vis, file = NULL)

export_svg(vis, file = NULL)
```

Arguments

vis	A ggvis object.
file	Output file name. If NULL, defaults to "plot.svg" or "plot.png".

See Also

https://github.com/trifacta/vega for information on installing vg2png and vg2svg.

Examples

```
## Not run:
mtcars %>% ggvis(x = ~wt) %>% export_png()

## End(Not run)
```

get_data *Get data from a ggvis object*

Description

This function is useful for inspecting the data in a ggvis object.

Usage

```
get_data(vis)
```

Arguments

vis	A ggvis object.

Examples

```
p <- cocaine %>% ggvis(~price) %>% layer_bars()
get_data(p)
```

ggvis *Visualise a data set with a ggvis graphic.*

Description

ggvis is used to turn a dataset into a visualisation, setting up default mappings between variables in the dataset and visual properties. Nothing will be displayed until you add additional layers.

Usage

```
ggvis(data = NULL, ..., env = parent.frame())
```

Arguments

data	A data object.
...	Property mappings. If not named, the first two mappings are taken to be x and y. Common properties are x, y, stroke, fill, opacity, shape
env	Environment in which to evaluate properties.

Examples

```
# If you don't supply a layer, ggvis uses layer_guess() to guess at
# an appropriate type:
mtcars %>% ggvis(~mpg, ~wt)
mtcars %>% ggvis(~mpg, ~wt, fill = ~cyl)
mtcars %>% ggvis(~mpg, ~wt, fill := "red")
mtcars %>% ggvis(~mpg)

# ggvis has a functional interface: every ggvis function takes a ggvis
# an input and returns a modified ggvis as output.
layer_points(ggvis(mtcars, ~mpg, ~wt))

# To make working with this interface more natural, ggvis imports the
# pipe operator from magrittr. x %>% f(y) is equivalent to f(x, y) so
# we can rewrite the previous command as
mtcars %>% ggvis(~mpg, ~wt) %>% layer_points()

# For more complicated plots, add a line break after %>%
mtcars %>%
  ggvis(~mpg, ~wt) %>%
  layer_points() %>%
  layer_smooths()
```

---

ggvisControlOutput *Create a ggvis control output element in UI*

Description

This is effectively the same as `uiOutput`, except that on the client side it may call some plot resizing functions after new controls are drawn.

Usage

```
ggvisControlOutput(outputId, plotId = NULL)
```

Arguments

outputId	The output variable to read the value from.
plotId	An optional plot ID or vector of plot IDs. The plots will have their .onControlOutput functions called after the controls are drawn.

Details

`ggvisControlOutput` is intended to be used with `bind_shiny` on the server side.

Examples

```
ggvisControlOutput("plot1")
```

ggvis_message *Send a message to ggvis running on client*

Description

This will be sent to the client and passed to a handler in ggvis.messages on the client side. The handler is specified by `type`.

Usage

```
ggvis_message(session, type, data = NULL, id = NULL)
```

Arguments

`session`	A session object.
`type`	A string representing the type of the message.
`data`	An object (typically a list) containing information for the client.
`id`	A unique identifier for ggvis message handler (optional).

group_by *Divide data into groups.*

Description

Divide data into groups.

Arguments

`x`	a visualisation
`...`	variables to group by.
`add`	By default, when `add = FALSE`, `group_by` will override existing groups. To instead add to the existing groups, use `add = TRUE`

handle_brush *Handle brush events on a visualisation.*

Description

Currently for brush events to be triggered on a visualisation, you must use a `.brush` property. This limitation will be lifted in the future.

Usage

```
handle_brush(vis, on_move = NULL, fill = "black")
```

Arguments

`vis`	Visualisation to listen to.
`on_move`	Callback function with arguments: **items** A data frame containing information about the items under the plot. An empty data.frame if no points under the brush. **page_loc** Location of the brush with repsect to the page **plot_loc** Location of the brush with respect to the plot **session** The session, used to communicate with the browser
`fill`	Colour of the brush.

Examples

```
# Display tooltip when objects are brushed
mtcars %>%
  ggvis(x = ~wt, y = ~mpg, size.brush := 400) %>%
  layer_points() %>%
  handle_brush(function(items, page_loc, session, ...) {
    show_tooltip(session, page_loc$r + 5, page_loc$t, html = nrow(items))
  })
```

handle_click *Handle mouse actions on marks.*

Description

Handle mouse actions on marks.

Usage

```
handle_click(vis, on_click = NULL)

handle_hover(vis, on_mouse_over = NULL, on_mouse_out = NULL)
```

Arguments

vis	Visualisation to listen to.
on_click, on_mouse_over	Callback function with arguments: **data** A data frame with one row **location** A named list with components x and y **session** The session, used to communicate with the browser
on_mouse_out	Callback function with argument: **session** The session, used to communicate with the browser

Examples

```
location <- function(location, ...) cat(location$x, "x", location$y, "\n")
mtcars %>% ggvis(~mpg, ~wt) %>% layer_points() %>%
  handle_click(location)
mtcars %>% ggvis(~mpg, ~wt) %>% layer_points() %>%
  handle_hover(function(...) cat("over\n"), function(...) cat("off\n"))
mtcars %>% ggvis(~mpg, ~wt) %>% layer_points() %>%
  handle_hover(function(data, ...) str(data))
```

---

handle_resize *Handlers and interactive inputs for plot sizing.*

Description

Handlers and interactive inputs for plot sizing.

Usage

```
handle_resize(vis, on_resize)

plot_width(vis)

plot_height(vis)
```

Arguments

vis	Visualisation to listen to.
on_resize	Callback function with arguments: **width,height** Width and height in pixels **padding** A named list of four components giving the padding in each direction **session** The session, used to communicate with the browser

Examples

```
# This example just prints out the current dimensions to the console
mtcars %>% ggvis(~mpg, ~wt) %>%
  layer_points() %>%
  handle_resize(function(width, height, ...) cat(width, "x", height, "\n"))

# Use plot_width() and plot_height() to dynamically get the plot size
# inside the plot.
mtcars %>% ggvis(~mpg, ~wt) %>% layer_text(text := plot_width())
mtcars %>% ggvis(~mpg, ~wt) %>% layer_text(text := plot_height())
```

---

input_checkbox *Create an interactive checkbox.*

Description

Create an interactive checkbox.

Usage

```
input_checkbox(
  value = FALSE,
  label = "",
  id = rand_id("checkbox_"),
  map = identity
)
```

Arguments

value	Initial value (TRUE or FALSE).
label	Display label for the control, or NULL for no label.
id	A unique identifier for this input. Usually generated automatically.
map	A function with single argument x, the value of the control on the client. Returns a modified value.

See Also

Other interactive input: input_select(), input_slider(), input_text()

Examples

```
input_checkbox(label = "Confidence interval")
input_checkbox(label = "Confidence interval", value = TRUE)

# Used in layer_smooths
mtcars %>% ggvis(~wt, ~mpg) %>%
```

```
  layer_smooths(se = input_checkbox(label = "Confidence interval"))

# Used with a map function, to convert the boolean to another type of value
model_type <- input_checkbox(label = "Use flexible curve",
  map = function(val) if(val) "loess" else "lm")
mtcars %>% ggvis(~wt, ~mpg) %>%
  layer_model_predictions(model = model_type)
```

`input_select` *Create interactive control to select one (or more options) from a list.*

Description

- `input_radiobuttons` only ever selects one value
- `input_checkboxgroup` can alway select multiple values
- `input_select` can select only one if `multiple = FALSE`, otherwise the user can select multiple by using modifier keys

Usage

```
input_select(
  choices,
  selected = NULL,
  multiple = FALSE,
  label = "",
  id = rand_id("select_"),
  map = identity,
  selectize = FALSE
)

input_radiobuttons(
  choices,
  selected = NULL,
  label = "",
  id = rand_id("radio_"),
  map = identity
)

input_checkboxgroup(
  choices,
  selected = NULL,
  label = "",
  id = rand_id("radio_"),
  map = identity
)
```

Arguments

choices	List of values to select from. If elements of the list are named, then that name — rather than the value — is displayed to the user. It's also possible to group related inputs by providing a named list whose elements are (either named or unnamed) lists, vectors, or factors. In this case, the outermost names will be used as the group labels (leveraging the <optgroup> HTML tag) for the elements in the respective sublist. See the example section for a small demo of this feature.
selected	The initially selected value (or multiple values if `multiple = TRUE`). If not specified then defaults to the first value for single-select lists and no values for multiple select lists.
multiple	Is selection of multiple items allowed?
label	Display label for the control, or `NULL` for no label.
id	A unique identifier for this input. Usually generated automatically.
map	A function with single argument `x`, the value of the control on the client. Returns a modified value.
selectize	Whether to use **selectize.js** or not.

See Also

Other interactive input: `input_checkbox()`, `input_slider()`, `input_text()`

Examples

```
# Dropdown
input_select(c("a", "b", "c"))
input_select(c("a", "b", "c"), multiple = TRUE)
input_select(c("a", "b", "c"), selected = "c")

# If you want to select variable names, you need to convert
# the output of the input to a name with map so that they get
# computed correctly
input_select(names(mtcars), map = as.name)

# Radio buttons
input_radiobuttons(choices = c("Linear" = "lm", "LOESS" = "loess"),
                   label = "Model type")
input_radiobuttons(choices = c("Linear" = "lm", "LOESS" = "loess"),
                   selected = "loess",
                   label = "Model type")

# Used in layer_model_predictions
mtcars %>% ggvis(~wt, ~mpg) %>%
  layer_model_predictions(model = input_radiobuttons(
    choices = c("Linear" = "lm", "LOESS" = "loess"),
    selected = "loess",
    label = "Model type"))

# Checkbox group
mtcars %>% ggvis(x = ~wt, y = ~mpg) %>%
```

```
  layer_points(
    fill := input_checkboxgroup(
      choices = c("Red" = "r", "Green" = "g", "Blue" = "b"),
      label = "Point color components",
      map = function(val) {
        rgb(0.8 * "r" %in% val, 0.8 * "g" %in% val, 0.8 * "b" %in% val)
      }
    )
  )
```

`input_slider` *Create an interactive slider.*

Description

Create an interactive slider.

Usage

```
input_slider(
  min,
  max,
  value = (min + max)/2,
  step = NULL,
  round = FALSE,
  format = NULL,
  locale = "us",
  ticks = TRUE,
  animate = FALSE,
  sep = ",",
  pre = NULL,
  post = NULL,
  label = "",
  id = rand_id("slider_"),
  map = identity
)
```

Arguments

`min`	The minimum value (inclusive) that can be selected.
`max`	The maximum value (inclusive) that can be selected.
`value`	The initial value of the slider. A numeric vector of length one will create a regular slider; a numeric vector of length two will create a double-ended range slider. A warning will be issued if the value doesn't fit between `min` and `max`.
`step`	Specifies the interval between each selectable value on the slider (if `NULL`, a heuristic is used to determine the step size). If the values are dates, `step` is in days; if the values are times (POSIXt), `step` is in seconds.

round	TRUE to round all values to the nearest integer; FALSE if no rounding is desired; or an integer to round to that number of digits (for example, 1 will round to the nearest 10, and -2 will round to the nearest .01). Any rounding will be applied after snapping to the nearest step.
format	Deprecated.
locale	Deprecated.
ticks	FALSE to hide tick marks, TRUE to show them according to some simple heuristics.
animate	TRUE to show simple animation controls with default settings; FALSE not to; or a custom settings list, such as those created using animationOptions().
sep	Separator between thousands places in numbers.
pre	A prefix string to put in front of the value.
post	A suffix string to put after the value.
label	Display label for the control, or NULL for no label.
id	A unique identifier for this input. Usually generated automatically.
map	A function with single argument x, the value of the control on the client. Returns a modified value.

See Also

Other interactive input: input_checkbox(), input_select(), input_text()

Examples

```
input_slider(0, 100)
input_slider(0, 100, label = "binwidth")
input_slider(0, 100, value = 50)

# Supply two values to value to make a double-ended sliders
input_slider(0, 100, c(25, 75))

# You can use map to transform the outputs
input_slider(-5, 5, label = "Log scale", map = function(x) 10 ^ x)
```

---

input_text *Create an interactive text or numeric input box.*

Description

input_numeric only allows numbers and comes with a spin box control. input_text allows any type of input.

Usage

```
input_text(value, label = "", id = rand_id("text_"), map = identity)

input_numeric(value, label = "", id = rand_id("numeric_"), map = identity)
```

Arguments

value	Initial value.
label	Display label for the control, or NULL for no label.
id	A unique identifier for this input. Usually generated automatically.
map	A function with single argument x, the value of the control on the client. Returns a modified value.

See Also

Other interactive input: input_checkbox(), input_select(), input_slider()

Examples

```
fill_text <- input_text(label = "Point color", value = "red")
mtcars %>% ggvis(~wt, ~mpg, fill := fill_text) %>% layer_bars()

size_num <- input_numeric(label = "Point size", value = 25)
mtcars %>% ggvis(~wt, ~mpg, size := size_num) %>% layer_points()
```

---

is.broker *Determine if an object is a broker object*

Description

Determine if an object is a broker object

Usage

```
is.broker(x)
```

Arguments

x	An object to test.

`layer_bars` *Display data with bars (a barchart).*

Description

This will add bars to a plot. The exact behavior is complicated because the term bar chart is used to describe four important variations on a theme. The action of `layer_bars` depends on two factors: whether or not a `y` prop has been specified, and whether the `x` props is continuous or categorical.

Usage

```
layer_bars(vis, ..., stack = TRUE, width = NULL)
```

Arguments

`vis`	Visualisation to modify
`...`	Visual properties used to override defaults.
`stack`	If there are multiple bars to be drawn at an x location, should the bars be stacked? If FALSE, the bars will be overplotted on each other.
`width`	Width of each bar. When x is continuous, this controls the width in the same units as x. When x is categorical, this controls the width as a proportion of the spacing between items (default is 0.9).

Visualisations

If no y prop has been specified, then this will count the number of entries at each unique x value. There will be one bar at each unique x value, and the y value (or height) of each bar will represent the count at that x value.

If a y prop has been specified, then those y values will be used as weights for a weighted count at each unique x value. If no x values appear more than once in the data, then the end result is a plot where the height of the bar at each x value is simply the y value. However, if an x value appear more than once in the data, then this will sum up the y values at each x.

If the x variable is continuous, then a continuous x axis will be used, and the width of each bar is by default equal to the resolution of the data – that is, the smallest difference between any two x values.

If the x variable is categorical, then a categorical x axis will be used. By default, the width of each bar is 0.9 times the space between the items.

See Also

`layer_histograms` For bar graphs of counts at each unique x value, in contrast to a histogram's bins along x ranges.

`compute_count` and `compute_tabulate` for more information on how data is transformed.

Examples

```
# Discrete x: bar graph of counts at each x value
cocaine %>% ggvis(~state) %>% layer_bars()
# Continuous x: bar graph of counts at unique locations
cocaine %>% ggvis(~month) %>% layer_bars()

# Use y prop to weight by additional variable. This is also useful
# if you have pretabulated data
cocaine %>% ggvis(~state, ~weight) %>% layer_bars()
cocaine %>% ggvis(~month, ~weight) %>% layer_bars()

# For continuous x, layer_bars is useful when the variable has a few
# unique values that you want to preserve. If you have many unique
# values and you want to bin, use layer_histogram
cocaine %>% ggvis(~price) %>% layer_bars()
cocaine %>% ggvis(~price) %>% layer_histograms(width = 100)

# If you have unique x values, you can use layer_bars() as an alternative
# to layer_points()
pressure %>% ggvis(~temperature, ~pressure) %>% layer_points()
pressure %>% ggvis(~temperature, ~pressure) %>% layer_bars()

# When x is continuous, width controls the width in x units
pressure %>% ggvis(~temperature, ~pressure) %>% layer_bars(width = 10)
# When x is categorical, width is proportional to spacing between bars
pressure %>% ggvis(~factor(temperature), ~pressure) %>%
  layer_bars(width = 0.5)

# Stacked bars
# If grouping var is continuous, you need to manually specify grouping
ToothGrowth %>% group_by(dose) %>%
  ggvis(x = ~supp, y = ~len, fill = ~dose) %>% layer_bars()
# If grouping var is categorical, grouping is done automatically
cocaine %>% ggvis(x = ~state, fill = ~as.factor(month)) %>%
  layer_bars()
```

`layer_boxplots` *Display data with a boxplot.*

Description

This will add boxplots to a plot. The action of `layer_boxplots` depends on whether the `x` prop is continuous or categorical.

Usage

```
layer_boxplots(vis, ..., coef = 1.5, width = NULL)
```

Arguments

vis	Visualisation to modify
...	Visual properties used to override defaults.
coef	The maximum length of the whiskers as multiple of the inter-quartile range. Default value is 1.5.
width	Width of each bar. When x is continuous, this controls the width in the same units as x. When x is categorical, this controls the width as a proportion of the spacing between items (default is 0.9).

Details

The upper and lower "hinges" correspond to the first and third quartiles (the 25th and 75th percentiles). This differs slightly from the method used by the boxplot function, and may be apparent with small samples. See boxplot.stats for more information on how hinge positions are calculated for boxplot.

The upper whisker extends from the hinge to the highest value that is within 1.5 * IQR of the hinge, where IQR is the inter-quartile range, or distance between the first and third quartiles. The lower whisker extends from the hinge to the lowest value within 1.5 * IQR of the hinge. Data beyond the end of the whiskers are outliers and plotted as points (as specified by Tukey).

See Also

compute_boxplot for more information on how data is transformed.

Examples

```
library(dplyr)

mtcars %>% ggvis(~factor(cyl), ~mpg) %>% layer_boxplots()
# Set the width of the boxes to half the space between tick marks
mtcars %>% ggvis(~factor(cyl), ~mpg) %>% layer_boxplots(width = 0.5)

# Continuous x: boxes fill width between data values
mtcars %>% ggvis(~cyl, ~mpg) %>% layer_boxplots()
# Setting width=0.5 makes it 0.5 wide in the data space, which is 1/4 of the
# distance between data values in this particular case.
mtcars %>% ggvis(~cyl, ~mpg) %>% layer_boxplots(width = 0.5)

# Smaller outlier points
mtcars %>% ggvis(~factor(cyl), ~mpg) %>% layer_boxplots(size := 20)
```

---

layer_densities *Transformation: density estimate*

Description

`transform_density` is a data transformation that computes a kernel density estimate from a dataset. `layer_density` combines `transform_density` with `mark_path` and `mark_area` to display a smooth line and its standard errror.

Usage

```
layer_densities(
  vis,
  ...,
  kernel = "gaussian",
  adjust = 1,
  density_args = list(),
  area = TRUE
)
```

Arguments

`vis`	The visualisation to modify
`...`	Visual properties, passed on to `props`.
`kernel`	Smoothing kernel. See `density` for details.
`adjust`	Multiple the default bandwidth by this amount. Useful for controlling wiggliness of density.
`density_args`	Other arguments passed on to `compute_density` and thence to `density`.
`area`	Should there be a shaded region drawn under the curve?

Examples

```
# Basic density estimate
faithful %>% ggvis(~waiting) %>% layer_densities()
faithful %>% ggvis(~waiting) %>% layer_densities(area = FALSE)

# Control bandwidth with adjust
faithful %>% ggvis(~waiting) %>% layer_densities(adjust = .25)
faithful %>% ggvis(~waiting) %>%
  layer_densities(adjust = input_slider(0.1, 5))

# Control stroke and fill
faithful %>% ggvis(~waiting) %>%
  layer_densities(stroke := "red", fill := "red")

# With groups
PlantGrowth %>% ggvis(~weight, fill = ~group) %>% group_by(group) %>%
  layer_densities()
PlantGrowth %>% ggvis(~weight, stroke = ~group) %>% group_by(group) %>%
  layer_densities(strokeWidth := 3, area = FALSE)
```

layer_guess *Guess the right type of layer based on current properties.*

Description

layer_guess provides the magic behind the default behaviour of ggvis.

Usage

```
layer_guess(vis, ...)
```

Arguments

vis	The visualisation to add the new layer to.
...	Other arguments passed on individual layers.

Defaults

- Continuous x, layer_histograms
- Categorical x, layer_bars
- Continuous x and y, layer_points

Examples

```
# A scatterplot:
mtcars %>% ggvis(~mpg, ~wt)
mtcars %>% ggvis(~mpg, ~wt) %>% layer_guess()

# A histogram:
mtcars %>% ggvis(~mpg)
mtcars %>% ggvis(~mpg) %>% layer_guess()
```

layer_histograms *Display binned data*

Description

Display binned data

Usage

```
layer_histograms(
  vis,
  ...,
  width = NULL,
  center = NULL,
  boundary = NULL,
  closed = c("right", "left"),
  stack = TRUE,
  binwidth
)

layer_freqpolys(
  vis,
  ...,
  width = NULL,
  center = NULL,
  boundary = NULL,
  closed = c("right", "left"),
  binwidth
)
```

Arguments

vis	Visualisation to modify
...	Visual properties used to override defaults.
width	The width of the bins. The default is NULL, which yields 30 bins that cover the range of the data. You should always override this value, exploring multiple widths to find the best to illustrate the stories in your data.
center	The center of one of the bins. Note that if center is above or below the range of the data, things will be shifted by an appropriate number of widths. To center on integers, for example, use width=1 and center=0, even if 0 is outside the range of the data. At most one of center and boundary may be specified.
boundary	A boundary between two bins. As with center, things are shifted when boundary is outside the range of the data. For example, to center on integers, use width = 1 and boundary = 0.5, even if 1 is outside the range of the data. At most one of center and boundary may be specified.
closed	One of "right" or "left" indicating whether right or left edges of bins are included in the bin.
stack	If TRUE, will automatically stack overlapping bars.
binwidth	Deprecated; use width instead.

See Also

layer_bars For bar graphs of counts at each unique x value, in contrast to a histogram's bins along x ranges.

Examples

```
# Create histograms and frequency polygons with layers
mtcars %>% ggvis(~mpg) %>% layer_histograms()
mtcars %>% ggvis(~mpg) %>% layer_histograms(width = 2)
mtcars %>% ggvis(~mpg) %>% layer_freqpolys(width = 2)

# These are equivalent to combining compute_bin with the corresponding
# mark
mtcars %>% compute_bin(~mpg) %>% ggvis(~x_, ~count_) %>% layer_paths()

# With grouping
mtcars %>% ggvis(~mpg, fill = ~factor(cyl)) %>% group_by(cyl) %>%
  layer_histograms(width = 2)
mtcars %>% ggvis(~mpg, stroke = ~factor(cyl)) %>% group_by(cyl) %>%
  layer_freqpolys(width = 2)
```

---

layer_lines *Layer lines on a plot.*

Description

layer_lines differs from layer_paths in that layer_lines sorts the data on the x variable, so the line will always proceed from left to right, whereas layer_paths will draw a line in whatever order appears in the data.

Usage

```
layer_lines(vis, ...)
```

Arguments

vis	Visualisation to modify.
...	Visual properties.

See Also

layer_paths

Examples

```
mtcars %>% ggvis(~wt, ~mpg, stroke = ~factor(cyl)) %>% layer_lines()

# Equivalent to
mtcars %>% ggvis(~wt, ~mpg, stroke = ~factor(cyl)) %>%
  group_by(cyl) %>% dplyr::arrange(wt) %>% layer_paths()
```

`layer_model_predictions` *Overlay model predictions or a smooth curve.*

Description

`layer_model_predictions` fits a model to the data and draw it with `layer_paths` and, optionally, `layer_ribbons`. `layer_smooths` is a special case of layering model predictions where the model is a smooth loess curve whose smoothness is controlled by the `span` parameter.

Usage

```
layer_model_predictions(
  vis,
  ...,
  model,
  formula = NULL,
  model_args = NULL,
  se = FALSE,
  domain = NULL
)

layer_smooths(vis, ..., span = 0.75, se = FALSE)
```

Arguments

`vis`	Visualisation to modify
`...`	Visual properties. Stroke properties control only affect line, fill properties only affect standard error band.
`model`	Name of the model as a string, e.g. `"loess"`, `"lm"`, or `"MASS::rlm"`. Must be the name of a function that produces a standard model object with a `predict` method. For `layer_smooth` this is always "loess".
`formula`	Model formula. If not supplied, guessed from the visual properties, constructing `y ~ x`.
`model_args`	A list of additional arguments passed on to the `model` function.
`se`	Also display a point-wise standard error band? Defaults to `FALSE` because interpretation is non-trivial.
`domain`	If `NULL` (the default), the domain of the predicted values will be the same as the domain of the prediction variable in the data. It can also be a two-element numeric vector specifying the min and max.
`span`	For `layer_smooth`, the span of the loess smoother.

Examples

```
mtcars %>% ggvis(~wt, ~mpg) %>% layer_smooths()
mtcars %>% ggvis(~wt, ~mpg) %>% layer_smooths(se = TRUE)

# Use group by to display multiple smoothes
mtcars %>% ggvis(~wt, ~mpg) %>% group_by(cyl) %>% layer_smooths()

# Control appearance with props
mtcars %>% ggvis(~wt, ~mpg) %>%
  layer_smooths(se = TRUE, stroke := "red", fill := "red", strokeWidth := 5)

# Control the wiggliness with span. Default is 0.75
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points() %>%
  layer_smooths(span = 0.2)
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points() %>%
  layer_smooths(span = 1)
# Map to an input to modify interactively
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points() %>%
  layer_smooths(span = input_slider(0.2, 1))

# Use other modelling functions with layer_model_predictions
mtcars %>% ggvis(~wt, ~mpg) %>%
  layer_points() %>%
  layer_model_predictions(model = "lm") %>%
  layer_model_predictions(model = "MASS::rlm", stroke := "red")

# Custom domain for predictions
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points() %>%
  layer_model_predictions(model = "lm", domain = c(0, 8))
mtcars %>% ggvis(~wt, ~mpg) %>% layer_points() %>%
  layer_model_predictions(model = "lm",
    domain = input_slider(0, 10, value = c(1, 4)))

# layer_smooths() is just compute_smooth() + layer_paths()
# Run loess or other model outside of a visualisation to see what variables
# you get.
mtcars %>% compute_smooth(mpg ~ wt)
mtcars %>% compute_model_prediction(mpg ~ wt, model = "lm")

mtcars %>%
  ggvis(~wt, ~mpg) %>%
  layer_points() %>%
  compute_smooth(mpg ~ wt) %>%
  layer_paths(~pred_, ~resp_, strokeWidth := 2)
```

---

left_right *Interactive inputs bound to arrow keys.*

Description

Interactive inputs bound to arrow keys.

Usage

```
left_right(min, max, value = (min + max)/2, step = (max - min)/40)

up_down(min, max, value = (min + max)/2, step = (max - min)/40)
```

Arguments

min	A minimum value.
max	A maximum value.
value	The initial value before any keys are pressed. Defaults to half-way between min and max.
step	How much each key press changes value. Defaults to 40 steps along range

Examples

```
size <- left_right(1, 801, value = 51, step = 50)
opacity <- up_down(0, 1, value = 0.9, step = 0.05)

mtcars %>% ggvis(~mpg, ~wt, size := size, opacity := opacity) %>%
  layer_points()
```

---

legend_props *Create an axis_props object for controlling legend properties.*

Description

The items in each of the lists can be a literal value, like 5 or "blue", or they can be a scaled_value object.

Usage

```
legend_props(
  title = NULL,
  labels = NULL,
  symbols = NULL,
  gradient = NULL,
  legend = NULL
)
```

Arguments

title	A named list of text properties for the legend title.
labels	A named list of text properties for legend labels.
symbols	A named list of line properties for symbols (for discrete legend items).
gradient	A named list of line properties a continuous color gradient.
legend	A named list of line properties for the overall legend. The x and y position can be set here, which will override automatic positioning.

`linked_brush` *Create a linked brush object.*

Description

A linked brush has two sides: input and output

Usage

```
linked_brush(keys, fill = "red")
```

Arguments

`keys`	vector of all possible keys, if known.
`fill`	brush colour

Value

A list with components:

`input`	A function that takes a visualisation as an argument and adds an input brush to that plot
`selected`	A reactive providing a logical vector that describes which points are under the brush
`fill`	A reactive that gives the fill colour of points under the brush

Note

`linked_brush` is very new and is likely to change substantially in the future

Examples

```
lb <- linked_brush(keys = 1:nrow(mtcars), "red")

# Change the colour of the points
mtcars %>%
 ggvis(~disp, ~mpg) %>%
 layer_points(fill := lb$fill, size.brush := 400) %>%
 lb$input()

# Display one layer with all points and another layer with selected points
library(shiny)
mtcars %>%
 ggvis(~disp, ~mpg) %>%
 layer_points(size.brush := 400) %>%
 lb$input() %>%
 layer_points(fill := "red", data = reactive(mtcars[lb$selected(), ]))
```

marks *Vega marks.*

Description

These functions create mark objects, corresponding to vega marks. Marks are leaves in the plot tree, and control the details of the final rendering. Marks are equivalent to the basic geoms in ggplot2 (e.g. point, line, polygon), where ggvis layers correspond to combinations of geoms and statistical transforms.

Usage

```
emit_points(vis, props)

layer_points(vis, ..., data = NULL)

emit_images(vis, props)

layer_images(vis, ..., data = NULL)

emit_arcs(vis, props)

layer_arcs(vis, ..., data = NULL)

emit_ribbons(vis, props)

layer_ribbons(vis, ..., data = NULL)

emit_paths(vis, props)

layer_paths(vis, ..., data = NULL)

emit_rects(vis, props)

layer_rects(vis, ..., data = NULL)

emit_text(vis, props)

layer_text(vis, ..., data = NULL)
```

Arguments

vis	Visualisation to modify
props, ...	A props object, named according to the properties listed below.
data	An optional dataset, if you want to override the usual data inheritance for this mark.

Details

Note that by supplying a fill property to `mark_path` will produce a filled property. `mark_point` is an alias to `mark_symbol`.

Properties

You can set the following mark properties:

- x The first (typically left-most) x-coordinate.
- x2 The second (typically right-most) x-coordinate.
- width The width of the mark (if supported).
- y The first (typically top-most) y-coordinate.
- y2 The second (typically bottom-most) y-coordinate.
- height The height of the mark (if supported).
- opacity The overall opacity.
- fill The fill color.
- fillOpacity The fill opacity
- stroke The stroke color.
- strokeWidth The stroke width, in pixels.
- strokeOpacity The stroke opacity.
- size [symbol] The pixel area of the symbol. For example in the case of circles, the radius is determined in part by the square root of the size value.
- shape [symbol] The symbol shape to use. One of circle (default), square, cross, diamond, triangle-up, or triangle-down (symbol only)
- innerRadius [arc] The inner radius of the arc, in pixels.
- outerRadius [arc] The outer radius of the arc, in pixels.
- startAngle [arc] The start angle of the arc, in radians.
- endAngle [arc] The end angle of the arc, in radians.
- interpolate [area, line] The line interpolation method to use. One of linear, step-before, step-after, basis, basis-open, cardinal, cardinal-open, monotone.
- tension [area, line] Depending on the interpolation type, sets the tension parameter.
- url [image] The URL from which to retrieve the image.
- align [image, text] The horizontal alignment of the object. One of left, right, center.
- baseline [image, text] The vertical alignment of the object. One of top, middle, bottom.
- text [text] The text to display.
- dx [text] The horizontal margin, in pixels, between the text label and its anchor point. The value is ignored if the align property is center.
- dy [text] The vertical margin, in pixels, between the text label and its anchor point. The value is ignored if the baseline property is middle.
- angle [text] The rotation angle of the text, in degrees.

- font [text] The typeface to set the text in (e.g., Helvetica Neue).
- fontSize [text] The font size, in pixels.
- fontWeight [text] The font weight (e.g., bold).
- fontStyle [text] The font style (e.g., italic).

To each property, you can assign any property object (prop) either locally (i.e. in the mark), or in a parent layer.

---

padding *Define padding.*

Description

Define padding.

Usage

```
padding(top = NULL, right = NULL, bottom = NULL, left = NULL)
```

Arguments

top, right, bottom, left
: Amount of padding on each border. Can either be a single number, "auto", or "strict"

Examples

```
p <- mtcars %>% ggvis(~wt, ~mpg) %>% layer_points()
p %>% set_options(padding = padding())
p %>% set_options(padding = padding(10, 10, 10, 10))
```

---

prop *Create a property.*

Description

Properties are used to describe the visual properties of marks. You create a single property defintion with prop, and manage sets of named properties with props (which also provides shortcuts for creating the most common kind of properties)

Usage

```
prop(
  property,
  x,
  scale = NULL,
  offset = NULL,
  mult = NULL,
  env = parent.frame(),
  event = NULL,
  label = NULL
)

is.prop(x)

is.prop_constant(x)

is.prop_variable(x)

is.prop_reactive(x)
```

Arguments

property	A property, like "x", "x2", "y", "fill", and so on.
x	The value of the property. This can be an atomic vector (a constant), a name or quoted call (a variable), a single-sided formula (a constant or variable depending on its contents), or a delayed reactive (which can be either variable or constant).
scale	If NULL, automatically determine behavior by the kind of value (constant, variable, or reactive). If TRUE use the default scale associated with property. If FALSE, do not scale the value. Otherwise supply a string to select a custom scale. If x is an interactive input, then this defaults to the scale parameter of the input.
offset, mult	Additive and multiplicate pixel offset used to adjust scaled values. These are useful if you want to place labels offset from points.
env	If x is a quoted call this provides the environment in which to look for variables not in the data. You should not need this in ordinary operation.
event	An event to which this property applies. One of "update", "enter", "exit", "hover", "brush".
label	A label for this prop to use for reporting errors.

See Also

props to manage multiple properties and to succintly create the most common types.

Examples

```
prop("x", 1)
prop("x", ~1)
```

```
prop("fill", quote(cyl))
prop("fill", ~cyl)
prop("x", input_slider(0, 100))

# If you have a variable name as a string
var <- "cyl"
prop("x", as.name(var))

# Use a custom scale
prop("y", quote(cyl), scale = "y-2")

# Don't scale variable (i.e. it already makes sense in the visual space)
prop("fill", ~colour, scale = FALSE)

# Use a constant, but scaled
prop("x", 5, scale = TRUE)

# Use other events
prop("y", quote(cyl), scale = "y-2")
```

`props` *Manage a list of properties.*

Description

`props()` provides a tool for concise creation of `prop` objects using a set of conventions designed to capture the most common use cases. If you need something less common, you'll need to use `prop` to access all possible options.

Usage

```
props(..., .props = NULL, inherit = TRUE, env = parent.frame())

is.ggvis_props(x)
```

Arguments

`...`	A set of name-value pairs. The name should be a valid vega property. The first two unnamed components are taken to be `x` and `y`. Any additional unnamed components will raise an error.
`.props`	When calling `props` from other functions, you'll often have a list of quoted function functions. You can pass that function to the `.props` argument instead of messing around with substitute. In other words, `.props` lets you opt out of the non-standard evaluation that `props` does.
`inherit`	If `TRUE`, the defaults, will inherit from properties from the parent layer If `FALSE`, it will start from nothing.
`env`	The environment in which to evaluate variable properties.
`x`	an object to test for props-ness.

Heuristics

If the values are not already objects of class `prop`, `props` uses the following heuristics to when creating the prop:

- atomic vectors, e.g. `x = 1`: scaled = FALSE
- an interative input, e.g. `x = input_slider`: scaled = FALSE
- a formula containing a single value, e.g. `x ~ 1`: scaled = TRUE
- a formula containing a name or expression, `x ~ mpg`: scaled = TRUE

Non-standard evaluation

`props` uses non-standard evaluation in a slightly unusual way: if you provide a formula input, the LHS of the formula will provide the name of the component. In otherwise, `props(x = y ~ 1)` is the same as `props(y ~ 1)`.

You can combine variables from the dataset and variables defined in the local environment: expressions will be evaluated in the environment which the formula was defined.

If you have the name of a variable in a string, see the props vignette for how to create the needed property mapping.

Enter, exit, hover, and update events

There are four different property events that the marks can use. These can, for example, be used to change the appearance of a mark when the mouse cursor is hovering over it: when the mark is hovered over, it uses the hover event, and when the mark isn't hovered over, it uses the update event

- enter: This event is used by marks when they are added to a plot.
- update: This event is used by marks after they have entered, and also after they have been hovered over.
- exit: This event is used by marks as they are removed from a plot.
- hover: This event is used when the mouse cursor is over the mark.

You can specify the event for a property, by putting a period and the event after the property name. For example, `props(fill.update := "black",fill.hover := "red")` will make a mark have a black fill normally, and red fill when it is hovered over.

The default event is update, so if you run `props(fill := "red")`, this is equivalent to `props(fill.update := "red")`.

In practice, the enter and exit events are useful only when the update has a duration (and is therefore not instantaneous). The update event can be thought of as the "default" state.

Key property

In addition to the standard properties, there is a special optional property called `key`. This is useful for plots with dynamic data and smooth transitions: as the data changes, the key is used to tell the plot how the new data rows should be matched to the old data rows. Note that the key must be an unscaled value. Additionally, the key property doesn't have a event, since it is independent of enter, update, exit, and hover events.

Properties

You can set the following mark properties:

- x The first (typically left-most) x-coordinate.
- x2 The second (typically right-most) x-coordinate.
- width The width of the mark (if supported).
- y The first (typically top-most) y-coordinate.
- y2 The second (typically bottom-most) y-coordinate.
- height The height of the mark (if supported).
- opacity The overall opacity.
- fill The fill color.
- fillOpacity The fill opacity
- stroke The stroke color.
- strokeWidth The stroke width, in pixels.
- strokeOpacity The stroke opacity.
- size [symbol] The pixel area of the symbol. For example in the case of circles, the radius is determined in part by the square root of the size value.
- shape [symbol] The symbol shape to use. One of circle (default), square, cross, diamond, triangle-up, or triangle-down (symbol only)
- innerRadius [arc] The inner radius of the arc, in pixels.
- outerRadius [arc] The outer radius of the arc, in pixels.
- startAngle [arc] The start angle of the arc, in radians.
- endAngle [arc] The end angle of the arc, in radians.
- interpolate [area, line] The line interpolation method to use. One of linear, step-before, step-after, basis, basis-open, cardinal, cardinal-open, monotone.
- tension [area, line] Depending on the interpolation type, sets the tension parameter.
- url [image] The URL from which to retrieve the image.
- align [image, text] The horizontal alignment of the object. One of left, right, center.
- baseline [image, text] The vertical alignment of the object. One of top, middle, bottom.
- text [text] The text to display.
- dx [text] The horizontal margin, in pixels, between the text label and its anchor point. The value is ignored if the align property is center.
- dy [text] The vertical margin, in pixels, between the text label and its anchor point. The value is ignored if the baseline property is middle.
- angle [text] The rotation angle of the text, in degrees.
- font [text] The typeface to set the text in (e.g., Helvetica Neue).
- fontSize [text] The font size, in pixels.
- fontWeight [text] The font weight (e.g., bold).
- fontStyle [text] The font style (e.g., italic).

To each property, you can assign any property object (`prop`) either locally (i.e. in the mark), or in a parent `layer`.

Examples

```
# Set to constant values
props(x := 1, y := 2)
# Map to variables in the dataset
props(x = ~mpg, y = ~cyl)
# Set to a constant value in the data space
props(x = 1, y = 1)
# Use an interactive slider
props(opacity := input_slider(0, 1))

# To control other settings (like custom scales, mult and offset)
# use a prop object
props(prop("x", "old", scale = "x", offset = -1))

# Red when hovered over, black otherwise (these are equivalent)
props(fill := "black", fill.hover := "red")
props(fill.update := "black", fill.hover := "red")

# Use a column called id as the key (for dynamic data)
props(key := ~id)

# Explicitly create prop objects. The following are equivalent:
props(fill = ~cyl)
props(fill.update = ~cyl)
props(prop("fill", ~cyl))
props(prop("fill", ~cyl, scale = "fill", event = "update"))

# Prop objects can be programmatically created and added:
property <- "fill"
expr <- parse(text = "wt/mpg")[[1]]
p <- prop(property, expr)
props(p)

# Using .props
props(.props = list(x = 1, y = 2))
props(.props = list(x = ~mpg, y = ~cyl))
props(.props = list(quote(x := ~mpg)))
```

`prop_domain` *Property domain.*

Description

Property domain.

Usage

```
prop_domain(x, data)
```

Arguments

x	property to dispatch on
data	name of data set

---

resolution *Compute the "resolution" of a data vector.*

Description

The resolution is is the smallest non-zero distance between adjacent values. If there is only one unique value, then the resolution is defined to be one.

Usage

```
resolution(x, zero = TRUE)
```

Arguments

x	numeric vector
zero	should a zero value be automatically included in the computation of resolution

Details

If x is an integer vector, then it is assumed to represent a discrete variable, and the resolution is 1.

Examples

```
resolution(1:10)
resolution((1:10) - 0.5)
resolution((1:10) - 0.5, FALSE)
resolution(c(1,2, 10, 20, 50))
resolution(as.integer(c(1, 10, 20, 50)))  # Returns 1
```

---

scaled_value *Create a scaled_value object*

Description

These are for use with legends and axes.

Usage

```
scaled_value(scale, value)
```

Arguments

scale	The name of a scale, e.g., "x", "fill".
value	A value which will be transformed using the scale.

`scales` *Add a scale to a ggvis plot*

Description

This creates a scale object for a given scale and variable type, and adds it to a ggvis plot. The scale object is populated with default settings, which depend on the scale (e.g. fill, x, opacity) and the type of variable (e.g. numeric, nominal, ordinal). Any settings that are passed in as arguments will override the defaults.

Arguments

`vis`	A ggvis object.
`scale`	The name of a scale, such as "x", "y", "fill", "stroke", etc.
`type`	A variable type. One of "numeric", "nominal", "ordinal", "logical", "datetime".
`...`	other arguments passed to the scale function. See the help for `scale_numeric`, `scale_ordinal` and `scale_datetime` for more details. For example, you might supply `trans = "log"` to create a log scale.
`name`	If `NULL`, the default, the scale name is the same as `scale`. Set this to a custom name to create multiple scales for stroke or fill, or (god forbid) a secondary y scale.

Scale selection

ggvis supports the following types of scales. Typical uses for each scale type are listed below:

- numeric For continuous numeric values.
- nominal For character vectors and factors.
- ordinal For ordered factors (these presently behave the same as nominal).
- logical For logical (TRUE/FALSE) values.
- datetime For dates and date-times.

Each type has a corresponding function: `scale_numeric`, `scale_nominal`, and so on.

The scale types for ggvis are mapped to scale types for Vega, which include "ordinal", "quantitative", and "time". See `ggvis_scale` for more details.

Given a scale and type, the range is selected based on the combination of the `scale` and `type`. For example, you get a different range of colours depending on whether the data is numeric, ordinal, or nominal. Some scales also set other properties. For example, nominal/ordinal position scales also add some padding so that points are spaced away from plot edges.

Not all combinations have an existing default scale. If you use a combination that does not have an existing combination, it may suggest you're displaying the data in a suboptimal way. For example, there is no default for a numeric shape scale, because there's no obvious way to map continuous values to discrete shapes.

Examples

```
p <- mtcars %>%
  ggvis(x = ~wt, y = ~mpg, fill = ~factor(cyl), stroke = ~hp) %>%
  layer_points()

p %>% scale_numeric("x")
p %>% scale_numeric("stroke")
p %>% scale_nominal("fill")

# You can also supply additional arguments or override the defaults
p %>% scale_numeric("x", trans = "log")
p %>% scale_numeric("stroke", range = c("red", "blue"))
```

---

scale_datetime *Add a date-time scale to a ggvis object.*

Description

A date/time scale controls the mapping of date and time variables to visual properties.

Usage

```
scale_datetime(
  vis,
  property,
  domain = NULL,
  range = NULL,
  reverse = NULL,
  round = NULL,
  utc = NULL,
  clamp = NULL,
  nice = NULL,
  expand = NULL,
  name = property,
  label = NULL,
  override = NULL
)
```

Arguments

vis	A ggvis object.
property	The name of a property, such as "x", "y", "fill", "stroke", etc.
domain	The domain of the scale, representing the set of data values. For ordinal scales, a character vector; for quantitative scales, a numeric vector of length two. Either value (but not both) may be NA, in which case domainMin or domainMax is set. For dynamic scales, this can also be a reactive which returns the appropriate type of vector.

range	The range of the scale, representing the set of visual values. For numeric values, the range can take the form of a two-element array with minimum and maximum values. For ordinal data, the range may by an array of desired output values, which are mapped to elements in the specified domain. The following range literals are also available: "width", "height", "shapes", "category10", "category20".
reverse	If true, flips the scale range.
round	If true, rounds numeric output values to integers. This can be helpful for snapping to the pixel grid.
utc	if TRUE, uses UTC times. Default is FALSE.
clamp	If TRUE, values that exceed the data domain are clamped to either the minimum or maximum range value. Default is FALSE.
nice	If specified, modifies the scale domain to use a more human-friendly value range. Should be a string indicating the desired time interval; legal values are "second", "minute", "hour", "day", "week", "month", or "year".
expand	A multiplier for how much the scale should be expanded beyond the domain of the data. For example, if the data goes from 10 to 110, and expand is 0.05, then the resulting domain of the scale is 5 to 115. Set to 0 and use nice=FALSE if you want exact control over the domain.
name	Name of the scale, such as "x", "y", "fill", etc. Can also be an arbitrary name like "foo".
label	Label for the scale. Used for axis or legend titles.
override	Should the domain specified by this ggvis_scale object override other ggvis_scale objects for the same scale? Useful when domain is manually specified. For example, by default, the domain of the scale will contain the range of the data, but when this is TRUE, the specified domain will override, and the domain can be smaller than the range of the data. If FALSE, the domain will not behave this way. If left NULL, then it will be treated as TRUE whenever domain is non-NULL.

See Also

scales, scale_numeric, https://github.com/trifacta/vega/wiki/Scales#time-scale-properties

Other scales: scale_numeric(), scale_ordinal()

Examples

```
set.seed(2934)
dat <- data.frame(
  time = as.Date("2013-07-01") + 1:100,
  value = seq(1, 10, length.out = 100) + rnorm(100)
)
p <- dat %>% ggvis(~time, ~value) %>% layer_points()

# Start and end on month boundaries
p %>% scale_datetime("x", nice = "month")
```

```
dist <- data.frame(times = as.POSIXct("2013-07-01", tz = "GMT") +
                           rnorm(200) * 60 * 60 * 24 * 7)
p <- dist %>% ggvis(x = ~times) %>% layer_histograms()
p

# Start and end on month boundaries
p %>% scale_datetime("x", nice = "month")

p %>% scale_datetime("x", utc = TRUE)
```

`scale_numeric` *Add a numeric scale to a ggvis object.*

Description

A numeric (quantitative) scale controls the mapping of continuous variables to visual properties.

Usage

```
scale_numeric(
  vis,
  property,
  domain = NULL,
  range = NULL,
  reverse = NULL,
  round = NULL,
  trans = NULL,
  clamp = NULL,
  exponent = NULL,
  nice = NULL,
  zero = NULL,
  expand = NULL,
  name = property,
  label = NULL,
  override = NULL
)
```

Arguments

`vis`	A ggvis object.
`property`	The name of a visual property, such as "x", "y", "fill", "stroke". Note both x and x2 use the "x" scale (similarly for y and y2). fillOpacity, opacity and strokeOpacity use the "opacity" scale.
`domain`	The domain of the scale, representing the set of data values. For ordinal scales, a character vector; for quantitative scales, a numeric vector of length two. Either value (but not both) may be NA, in which case `domainMin` or `domainMax` is set. For dynamic scales, this can also be a reactive which returns the appropriate type of vector.

range	The range of the scale, representing the set of visual values. For numeric values, the range can take the form of a two-element array with minimum and maximum values. For ordinal data, the range may by an array of desired output values, which are mapped to elements in the specified domain. The following range literals are also available: "width", "height", "shapes", "category10", "category20".
reverse	If true, flips the scale range.
round	If true, rounds numeric output values to integers. This can be helpful for snapping to the pixel grid.
trans	A scale transformation: one of "linear", "log", "pow", "sqrt", "quantile", "quantize", "threshold". Default is "linear".
clamp	If TRUE, values that exceed the data domain are clamped to either the minimum or maximum range value. Default is FALSE.
exponent	Sets the exponent of the scale transformation. For pow transform only.
nice	If TRUE, modifies the scale domain to use a more human-friendly number range (e.g., 7 instead of 6.96). Default is FALSE.
zero	If TRUE, ensures that a zero baseline value is included in the scale domain. This option is ignored for non-quantitative scales. Default is FALSE.
expand	A multiplier for how much the scale should be expanded beyond the domain of the data. For example, if the data goes from 10 to 110, and expand is 0.05, then the resulting domain of the scale is 5 to 115. Set to 0 and use nice=FALSE if you want exact control over the domain. If left NULL, behavior will depend on the scale type. For positional scales (x and y), expand will default to 0.05. For other scales, it will default to 0.
name	Name of the scale, such as "x", "y", "fill", etc. Can also be an arbitrary name like "foo".
label	Label for the scale. Used for axis or legend titles.
override	Should the domain specified by this ggvis_scale object override other ggvis_scale objects for the same scale? Useful when domain is manually specified. For example, by default, the domain of the scale will contain the range of the data, but when this is TRUE, the specified domain will override, and the domain can be smaller than the range of the data. If FALSE, the domain will not behave this way. If left NULL, then it will be treated as TRUE whenever domain is non-NULL.

Details

The default values for most of the arguments is NULL. When the plot is created, these NULL values will be replaced with default values, as indicated below.

See Also

scales, scale_ordinal, https://github.com/trifacta/vega/wiki/Scales#quantitative-scale-properties

Other scales: scale_datetime(), scale_ordinal()

Examples

```
p <- mtcars %>% ggvis(~wt, ~mpg, fill = ~hp) %>% layer_points()

p %>% scale_numeric("y")

p %>% scale_numeric("y", trans = "pow", exponent = 0.5)

p %>% scale_numeric("y", trans = "log")

# Can control other properties other than x and y
p %>% scale_numeric("fill", domain = c(0, 120), clamp = TRUE)

# Set range of data from 0 to 3
p %>% scale_numeric("x", domain = c(0, 3), clamp = TRUE, expand = 0,
                    nice = FALSE)

# Lower bound is set to lower limit of data, upper bound set to 3.
p %>% scale_numeric("x", domain = c(NA, 3), clamp = TRUE, nice = FALSE)
```

---

scale_ordinal *Add a ordinal, nominal, or logical scale to a ggvis object.*

Description

Ordinal, nominal, and logical scales are all categorical, and are treated similarly by ggvis.

Usage

```
scale_ordinal(
  vis,
  property,
  domain = NULL,
  range = NULL,
  reverse = NULL,
  round = NULL,
  points = NULL,
  padding = NULL,
  sort = NULL,
  name = property,
  label = NULL,
  override = NULL
)

scale_nominal(
  vis,
  property,
  domain = NULL,
  range = NULL,
```

```
  reverse = NULL,
  round = NULL,
  points = NULL,
  padding = NULL,
  sort = NULL,
  name = property,
  label = NULL,
  override = NULL
)

scale_logical(
  vis,
  property,
  domain = NULL,
  range = NULL,
  reverse = NULL,
  round = NULL,
  points = NULL,
  padding = NULL,
  sort = NULL,
  name = property,
  label = NULL,
  override = NULL
)
```

Arguments

vis	A ggvis object.
property	The name of a property, such as "x", "y", "fill", "stroke", etc.
domain	The domain of the scale, representing the set of data values. For ordinal scales, a character vector; for quantitative scales, a numeric vector of length two. Either value (but not both) may be NA, in which case domainMin or domainMax is set. For dynamic scales, this can also be a reactive which returns the appropriate type of vector.
range	The range of the scale, representing the set of visual values. For numeric values, the range can take the form of a two-element array with minimum and maximum values. For ordinal data, the range may by an array of desired output values, which are mapped to elements in the specified domain. The following range literals are also available: "width", "height", "shapes", "category10", "category20".
reverse	If true, flips the scale range.
round	If true, rounds numeric output values to integers. This can be helpful for snapping to the pixel grid.
points	If TRUE (default), distributes the ordinal values over a quantitative range at uniformly spaced points. The spacing of the points can be adjusted using the padding property. If FALSE, the ordinal scale will construct evenly-spaced bands, rather than points. Note that if any mark is added with a band() prop, then the scale for that prop will automatically have points set to FALSE.

padding	Applies spacing among ordinal elements in the scale range. The actual effect depends on how the scale is configured. If the points parameter is true, the padding value is interpreted as a multiple of the spacing between points. A reasonable value is 1.0, such that the first and last point will be offset from the minimum and maximum value by half the distance between points. Otherwise, padding is typically in the range [0, 1] and corresponds to the fraction of space in the range interval to allocate to padding. A value of 0.5 means that the range band width will be equal to the padding width. For positional (x and y) scales, the default padding is 0.1. For other scales, the default padding is 0.5.
sort	If TRUE, the values in the scale domain will be sorted according to their natural order. Default is FALSE.
name	Name of the scale, such as "x", "y", "fill", etc. Can also be an arbitrary name like "foo".
label	Label for the scale. Used for axis or legend titles.
override	Should the domain specified by this ggvis_scale object override other ggvis_scale objects for the same scale? Useful when domain is manually specified. For example, by default, the domain of the scale will contain the range of the data, but when this is TRUE, the specified domain will override, and the domain can be smaller than the range of the data. If FALSE, the domain will not behave this way. If left NULL, then it will be treated as TRUE whenever domain is non-NULL.

See Also

scales, scale_numeric, https://github.com/trifacta/vega/wiki/Scales#ordinal-scale-properties, https://github.com/d3/d3/wiki/Ordinal-Scales

Other scales: scale_datetime(), scale_numeric()

Examples

```
p <- PlantGrowth %>% ggvis(~group, ~weight) %>% layer_points()

p
p %>% scale_nominal("x", padding = 0)
p %>% scale_nominal("x", padding = 1)

p %>% scale_nominal("x", reverse = TRUE)

p <- ToothGrowth %>% group_by(supp) %>%
  ggvis(~len, fill = ~supp) %>%
  layer_histograms(width = 4, stack = TRUE)

# Control range of fill scale
p %>% scale_nominal("fill", range = c("pink", "lightblue"))

# There's no default range when the data is categorical but the output range
# is continuous, as in the case of opacity. In these cases, you can
# manually specify the range for the scale.
mtcars %>% ggvis(x = ~wt, y = ~mpg, opacity = ~factor(cyl)) %>%
  layer_points() %>%
  scale_nominal("opacity", range = c(0.2, 1))
```

set_options *Set options for a ggvis plot*

Description

Set options for a ggvis plot

Usage

```
set_options(
  vis,
  width = NULL,
  height = NULL,
  keep_aspect = NULL,
  resizable = NULL,
  padding = NULL,
  duration = NULL,
  renderer = NULL,
  hover_duration = NULL
)
```

Arguments

vis	Visualisation to modify
width, height	Width and height of plot, in pixels. Default is 600x400. width or height can also be "auto", in which case the plot will size to fit in the containing div. This is useful only in a Shiny app or custom HTML output. Note that height="auto" should only be used when the plot is placed within a div that has a fixed height; if not, automatic height will not work, due to the way that web browsers do vertical layout.
keep_aspect	Should the aspect ratio be preserved? The default value is FALSE, or the value of getOption("ggvis.keep_aspect"), if it is set.
resizable	If TRUE, allow the user to resize the plot. The default value is TRUE, or the value of getOption("ggvis.resizable"), if it is set. Not compatible when width or height is "auto".
padding	A padding object specifying padding on the top, right, left, and bottom. See padding.
duration	Duration of transitions, in milliseconds.
renderer	The renderer to use in the browser. Can be "canvas" or "svg" (the default).
hover_duration	The amount of time for hover transitions, in milliseconds.

See Also

getOption and options, for getting and setting global options.

default_options to see the default options.

Examples

```
mtcars %>%
  ggvis(~wt, ~mpg) %>%
  layer_points() %>%
  set_options(width = 300, height = 200, padding = padding(10, 10, 10, 10))

# Display the default options
str(default_options())
```

`set_scale_label` *Set the label for a scale*

Description

Set the label for a scale

Usage

```
set_scale_label(vis, scale, label)
```

Arguments

`vis`	A ggvis object.
`scale`	The name of a scale, like "x".
`label`	Text to use for the label.

`shiny-ggvis` *Connect a ggvis graphic to a shiny app.*

Description

Embedding ggvis in a shiny app is easy. You need to make a place for it in your `ui.r` with `ggvisOutput`, and tell your `server.r` where to draw it with `bind_shiny`. It's easiest to learn by example: there are many shiny apps in `demo/apps/` that you can learn from.

Usage

```
bind_shiny(
  vis,
  plot_id,
  controls_id = NULL,
  ...,
  session = shiny::getDefaultReactiveDomain()
)
```

```
bind_shiny_ui(vis, controls_id, session = shiny::getDefaultReactiveDomain())

ggvisOutput(plot_id = rand_id("plot_id"))
```

Arguments

vis	A ggvis object, or a reactive expression that returns a ggvis object.
plot_id	unique identifier to use for the div containing the ggvis plot.
controls_id	Unique identifier for controls div.
...	Other arguments passed to as.vega.
session	A Shiny session object.

Client-side

In your UI, use ggvisOutput() in ui.r to insert an html placeholder for the plot.

If you're going to be using interactive controls generated by ggvis, use renderUI() to add a place holder. By convention, if the id of plot placehold is called "plot", call the controls placeholder "plot_ui".

Server-side

When you run ggvis plot interactively, it is automatically plotted because it triggers the default print method. In shiny apps, you need to explicitly render the plot to a specific placeholder with bind_shiny:

```
p %>% bind_shiny("plot")
```

If the plot has controls, and you've reserved space for them in the UI, supply the name of the placeholder as the third argument:

```
p %>% bind_shiny("plot","plot_ui")
```

Examples

```
## Run these examples only in interactive R sessions
if (interactive()) {

# Simplest possible app:
library(shiny)
runApp(list(
  ui = bootstrapPage(
    ggvisOutput("p"),
    uiOutput("p_ui")
  ),
  server = function(..., session) {
    mtcars %>%
      ggvis(~wt, ~mpg) %>%
      layer_points() %>%
      layer_smooths(span = input_slider(0, 1)) %>%
      bind_shiny("p", "p_ui")
```

```
  }
))

}
```

---

show_spec *Print out the vega plot specification*

Description

Print out the vega plot specification

Usage

```
show_spec(vis, pieces = NULL)
```

Arguments

vis	Visualisation to print
pieces	Optional, a character or numeric vector used to pull out selected pieces of the spec

Examples

```
base <- mtcars %>% ggvis(~mpg, ~wt) %>% layer_points()
base %>% show_spec()
base %>% show_spec("scales")
```

---

show_tooltip *Send a message to the client to show or hide a tooltip*

Description

Send a message to the client to show or hide a tooltip

Usage

```
show_tooltip(session, l = 0, t = 0, html = "")

hide_tooltip(session)
```

Arguments

session	A Shiny session object.
l	Pixel location of left edge of tooltip (relative to page)
t	Pixel location of top edge of tooltip (relative to page)
html	HTML to display in the tooltip box.

sidebarBottomPage *Create a page with a sidebar*

Description

This creates a page with a sidebar, where the sidebar moves to the bottom when the width goes below a particular value.

Usage

```
sidebarBottomPage(sidebarPanel, mainPanel, shiny_headers = TRUE)

sidebarBottomPanel(...)

mainTopPanel(...)
```

Arguments

sidebarPanel	The sidebarBottomPanel containing input controls.
mainPanel	The mainTopPanel containing the main content.
shiny_headers	Should Shiny headers be embedded in the page? This should be TRUE for interactive/dynamic pages, FALSE for static pages.
...	Additional tags.

Examples

```
sidebarBottomPage(sidebarBottomPanel(), mainTopPanel())
```

singular *singular.*

Description

Use singular when you want constant x or y position.

Usage

```
singular()

scale_singular(
  vis,
  property,
  name = property,
  label = name,
  points = TRUE,
```

```
  domain = NULL,
  override = NULL
)
```

Arguments

vis	A ggvis object.
property	The name of a property, such as "x", "y", "fill", "stroke", etc.
name	Name of the scale, such as "x", "y", "fill", etc. Can also be an arbitrary name like "foo".
label	Label for the scale. Used for axis or legend titles.
points	If TRUE (default), distributes the ordinal values over a quantitative range at uniformly spaced points. The spacing of the points can be adjusted using the padding property. If FALSE, the ordinal scale will construct evenly-spaced bands, rather than points. Note that if any mark is added with a band() prop, then the scale for that prop will automatically have points set to FALSE.
domain	The domain of the scale, representing the set of data values. For ordinal scales, a character vector; for quantitative scales, a numeric vector of length two. Either value (but not both) may be NA, in which case domainMin or domainMax is set. For dynamic scales, this can also be a reactive which returns the appropriate type of vector.
override	Should the domain specified by this ggvis_scale object override other ggvis_scale objects for the same scale? Useful when domain is manually specified. For example, by default, the domain of the scale will contain the range of the data, but when this is TRUE, the specified domain will override, and the domain can be smaller than the range of the data. If FALSE, the domain will not behave this way. If left NULL, then it will be treated as TRUE whenever domain is non-NULL.

Examples

```
mtcars %>% ggvis("", ~mpg) %>%
  layer_points() %>%
  scale_nominal("x") %>%
  add_axis("x", title = "", tick_size_major = 0)

# OR
mtcars %>% ggvis("", ~mpg) %>%
  layer_points() %>%
  scale_singular("x")

# OR, even simpler
mtcars %>% ggvis(singular(), ~mpg) %>% layer_points()

# In the other direction:
mtcars %>% ggvis(~mpg, singular()) %>% layer_points()
```

vector_type *Determine the "type" of a vector*

Description

The vector_type collapses down the class of base vectors into something useful more for visualisation, yielding one of "datetime", "numeric", "ordinal", "nominal" or "logical".

Usage

```
vector_type(x)
```

Arguments

x	a vector

See Also

default_scale, which uses this when picking the default scale.

vega_data_parser *Determine the vega data type for a vector*

Description

This is used to specify the data type so that the appropriate parser is used when Vega receives the data.

Usage

```
vega_data_parser(x)
```

Arguments

x	A vector.

waggle *Waggle back and forth between two numbers*

Description

Waggle back and forth between two numbers

Usage

```
waggle(min, max, value = (min + max)/2, step = (max - min)/50, fps = 10)
```

Arguments

min	A minimum value.
max	A maximum value.
value	Starting value. Defaults to half-way between min and max.
step	How much value changes at each frame. Defaults to 50 steps between min and max so it takes 5 seconds to waggle once.
fps	number of frames per second.

Examples

```
span <- waggle(0.2, 1)
mtcars %>% ggvis(~mpg, ~wt) %>%
 layer_points() %>%
 layer_smooths(span = span)
```

zero_range *Determine if range of vector is close to zero, with a specified tolerance*

Description

The machine epsilon is the difference between 1.0 and the next number that can be represented by the machine. By default, this function uses epsilon * 100 as the tolerance. First it scales the values so that they have a mean of 1, and then it checks if the difference between them is larger than the tolerance.

Usage

```
zero_range(x, tol = .Machine$double.eps * 100)
```

Arguments

x	numeric range: vector of length 2
tol	A value specifying the tolerance. Defaults to .Machine$double.eps * 100.

Value

logical TRUE if the relative difference of the endpoints of the range are not distinguishable from 0.

Examples

```
eps <- .Machine$double.eps
zero_range(c(1, 1 + eps))       # TRUE
zero_range(c(1, 1 + 99 * eps))  # TRUE
zero_range(c(1, 1 + 101 * eps)) # FALSE - Crossed the tol threshold
zero_range(c(1, 1 + 2 * eps), tol = eps) # FALSE - Changed tol

# Scaling up or down all the values has no effect since the values
# are rescaled to 1 before checking against tol
zero_range(100000 * c(1, 1 + eps))        # TRUE
zero_range(100000 * c(1, 1 + 200 * eps))  # FALSE
zero_range(.00001 * c(1, 1 + eps))        # TRUE
zero_range(.00001 * c(1, 1 + 200 * eps))  # FALSE

# NA values
zero_range(c(1, NA))   # NA
zero_range(c(1, NaN))  # NA

# Infinite values
zero_range(c(1, Inf))     # FALSE
zero_range(c(-Inf, Inf))  # FALSE
zero_range(c(Inf, Inf))   # TRUE
```

%>% *Pipe graphics*

Description

Like dplyr, ggvis also uses the pipe function, %>% to turn function composition into a series of imperative statements.

Arguments

lhs, rhs	A visualisation and a function to apply to it

Examples

```
# Instead of
layer_points(ggvis(mtcars, ~mpg, ~wt))
# you can write
mtcars %>% ggvis(~mpg, ~wt) %>% layer_points()
```

Index