Package ‘ggplot2’

December 18, 2015

Version 2.0.0
Title An Implementation of the Grammar of Graphics
Description An implementation of the grammar of graphics in R. It combines the advantages of both base and lattice graphics: conditioning and shared axes are handled automatically, and you can still build up a plot step by step from multiple data sources. It also implements a sophisticated multidimensional conditioning system and a consistent interface to map data to aesthetic attributes. See http://ggplot2.org for more information, documentation and examples.
Depends R (>= 2.14)
Imports digest, grid, gtable (>= 0.1.1), MASS, plyr (>= 1.7.1), reshape2, scales (>= 0.3.0), stats
Suggests ggplot2movies, hexbin, Hmisc, mapproj, maps, maptools, mgcv, multcomp, nlme, testthat, quantreg, knitr
Enhances sp
License GPL-2
BugReports https://github.com/hadley/ggplot2/issues
LazyData true
Collate 'ggproto.r' 'aaa-.r' 'aes-calculated.r'
'aes-colour-fill-alpha.r' 'aes-group-order.r'
'aes-linetype-size-shape.r' 'aes-position.r' 'utilities.r'
'aes.r' 'legend-draw.r' 'geom-.r' 'annotation-custom.r'
'annotation-logticks.r' 'geom-polygon.r' 'geom-map.r'
'annotation-map.r' 'geom-raster.r' 'annotation-raster.r'
'annotation.r' 'autoplot.r' 'bench.r' 'coord-.r'
'coord-cartesian-.r' 'coord-fixed.r' 'coord-flip.r'
'coord-map.r' 'coord-munch.r' 'coord-polar.r'
'coord-quickmap.R' 'coord-transform.r' 'data.R' 'facet-.r'
'facet-grid-.r' 'facet-labels.r' 'facet-layout.r'
'facet-locate.r' 'facet-null.r' 'facet-viewport.r'
'facet-wrap.r' 'fortify-lm.r' 'fortify-map.r'
VignetteBuilder knitr
RoxygenNote 5.0.0

NeedsCompilation no

Author Hadley Wickham [aut, cre],
Winston Chang [aut],
RStudio [cph]

Maintainer Hadley Wickham <hadley@rstudio.com>

Repository CRAN

Date/Publication 2015-12-18 10:45:15
R topics documented:

- gg ................................................................. 6
- add_theme ....................................................... 7
- aes ............................................................... 8
- aes_ ............................................................. 9
- aes_colour_fill_alpha ........................................ 10
- aes_group_order ............................................... 11
- aes_linetype_size_shape .................................... 12
- aes_position .................................................... 14
- annotate ......................................................... 15
- annotation_custom ........................................... 16
- annotation_logticks ......................................... 17
- annotation_map ............................................... 19
- annotation_raster ............................................ 19
- as.list.ggproto ............................................... 20
- as_labeller ....................................................... 21
- autoplot ........................................................ 22
- borders ........................................................ 22
- calc_element ................................................... 23
- coord_cartesian ............................................... 24
- coord_fixed ..................................................... 25
- coord_flip ....................................................... 26
- coord_map ........................................................ 27
- coord_polar ..................................................... 29
- coord_trans ..................................................... 30
- cut_interval .................................................... 32
- diamonds ....................................................... 33
- economics ....................................................... 34
- element_blank .................................................. 34
- element_line .................................................... 35
- element_rect .................................................... 35
- element_text .................................................... 36
- expand_limits ................................................... 36
- facet_grid ......................................................... 37
- facet_null ....................................................... 41
- facet_wrap ....................................................... 41
- faithful .......................................................... 43
- format.ggproto ............................................... 44
- fortify .......................................................... 44
- fortify-multcomp .............................................. 45
- fortify.lm ....................................................... 46
- fortify.map ...................................................... 47
- fortify.sp ....................................................... 48
- geom_abline ..................................................... 49
- geom_bar ......................................................... 51
- geom_bin2d ...................................................... 54
- geom_blank ...................................................... 56
R topics documented:

geom_boxplot ................................................................. 57
geom_contour ................................................................. 60
geom_count ................................................................. 62
geom_crossbar ................................................................. 64
geom_density ................................................................. 66
geom_density_2d ............................................................... 68
geom_dotplot ................................................................. 71
geom_errorbarh ................................................................. 74
geom_freqpoly ................................................................. 75
geom_hex ................................................................. 78
geom_jitter ................................................................. 80
geom_label ................................................................. 82
geom_map ................................................................. 85
geom_path ................................................................. 87
geom_point ................................................................. 90
geom_polygon ................................................................. 93
geom_quantile ................................................................. 95
geom_raster ................................................................. 97
geom_ribbon ................................................................. 99
geom_rug ................................................................. 101
geom_segment ............................................................... 102
geom_smooth ............................................................... 105
geom_spoke ............................................................... 108
geom_violin ............................................................... 109
ggplot ................................................................. 112
ggproto ................................................................. 114
ggsave ................................................................. 114
ggtheme ................................................................. 116
graphical-units ............................................................ 117
guides ................................................................. 118
guide_colourbar ............................................................... 119
guide_legend ............................................................... 122
hmisc ................................................................. 125
is.ggproto ................................................................. 125
is.rel ................................................................. 126
is.theme ................................................................. 126
labeller ................................................................. 126
labellers ................................................................. 128
label_bquote ............................................................... 130
labs ................................................................. 131
last_plot ................................................................. 132
layer ................................................................. 132
lims ................................................................. 133
luv_colours ................................................................. 134
map_data ................................................................. 135
margin ................................................................. 136
mean_se ................................................................. 136
midwest ................................................................. 137
Modify a ggplot or theme object by adding on new components.

Description

This operator allows you to add objects to a ggplot or theme object.

Usage

```r
## S3 method for class 'gg'
e1 + e2

e1 %+% e2

e1 %+replace% e2
```

Arguments

- `e1`: An object of class `ggplot` or `theme`
- `e2`: A component to add to `e1`

Details

If the first object is an object of class `ggplot`, you can add the following types of objects, and it will return a modified ggplot object.

- `data.frame`: replace current data.frame (must use `%+%`)
- `uneval`: replace current aesthetics
- `layer`: add new layer
- `theme`: update plot theme
- `scale`: replace current scale
- `coord`: override current coordinate system
- `facet`: override current coordinate faceting

If the first object is an object of class `theme`, you can add another theme object. This will return a modified theme object.

For theme objects, the `+` operator and the `%+replace%` can be used to modify elements in themes.

The `+` operator updates the elements of `e1` that differ from elements specified (not NULL) in `e2`. Thus this operator can be used to incrementally add or modify attributes of a ggplot theme.

In contrast, the `%+replace%` operator replaces the entire element; any element of a theme not specified in `e2` will not be present in the resulting theme (i.e. NULL). Thus this operator can be used to overwrite an entire theme.
**add_theme**

Modify properties of an element in a theme object

**Description**

Modify properties of an element in a theme object

**Usage**

```
add_theme(t1, t2, t2name)
```

**Arguments**

- **t1**: A theme object
- **t2**: A theme object that is to be added to t1
- **t2name**: A name of the t2 object. This is used for printing informative error messages.

**See Also**

`+gg`
Define aesthetic mappings.

Description

Generate aesthetic mappings that describe how variables in the data are mapped to visual properties (aesthetics) of geoms. This function also standardise aesthetic names by performs partial name matching, converting color to colour, and old style R names to ggplot names (eg. pch to shape, cex to size).

Usage

```r
aes(x, y, ...)
```

Arguments

- `x, y, ...` List of name value pairs giving aesthetics to map to variables. The names for x and y aesthetics can be omitted (because they are so common); all other aesthetics must be named.

See Also

See `aes_q/aes_string` for standard evaluation versions of `aes`.

See `aes_colour_fill_alpha, aes_group_order, aes_linetype_size_shape` and `aes_position` for more specific examples with different aesthetics.

Examples

```r
aes(x = mpg, y = wt)
```

```r
aes(mpg, wt)
```

# You can also map aesthetics to functions of variables
```r
eaes(x = mpg ^ 2, y = wt / cyl)
```n
# Aesthetic names are automatically standardised
```r
aes(col = x)
aes(fg = x)
aes(color = x)
aes(colour = x)
```

# aes is almost always used with `ggplot()` or a layer
```r
ggplot(mpg, aes(displ, hwy)) + geom_point()
ggplot(mpg) + geom_point(aes(displ, hwy))
```n
# Aesthetics supplied to `ggplot()` are used as defaults for every layer
# you can override them, or supply different aesthetics for each layer
**aes_**  
*Define aesthetic mappings from strings, or quoted calls and formulas.*

**Description**

Aesthetic mappings describe how variables in the data are mapped to visual properties (aesthetics) of geoms. `aes` uses non-standard evaluation to capture the variable names. `aes_` and `aes_string` require you to explicitly quote the inputs either with "" for `aes_string()`, or with `quote` or ~ for `aes_()`. (`aes_q` is an alias to `aeq_`)

**Usage**

```r
aes_(x, y, ...)
aes_string(x, y, ...)
aes_q(x, y, ...)
```

**Arguments**

`x, y, ...`  
List of name value pairs. Elements must be either quoted calls, strings, one-sided formulas or constants.

**Details**

It’s better to use `aes_q()`, because there’s no easy way to create the equivalent to `aes(colour = "my colour")` or `aes(x = `X$1`)` with `aes_string()`.

`aes_string` and `aes_` are particularly useful when writing functions that create plots because you can use strings or quoted names/calls to define the aesthetic mappings, rather than having to use `substitute` to generate a call to `aes()`.

**See Also**

`aes`

**Examples**

```r
# Three ways of generating the same aesthetics
aes(mpg, wt, col = cyl)
aes_(quote(mpg), quote(wt), col = quote(cyl))
aes_(~mpg, ~wt, col = ~cyl)
aes_string("mpg", "wt", col = "cyl")

# You can’t easily mimic these calls with aes_string
aes("$100", colour = "smooth")
aes_(~ "$100", colour = "smooth")
# Ok, you can, but it requires a _lot_ of quotes
aes_string(""$100"", colour = "smooth")
```
# Convert strings to names with as.name
var <- "cyl"
aes(col = x)
aes_(col = as.name(var))

---

**aes_colour_fill_alpha  Colour related aesthetics: colour, fill and alpha**

**Description**

This page demonstrates the usage of a sub-group of aesthetics; colour, fill and alpha.

**Examples**

```r
# Bar chart example
c <- ggplot(mtcars, aes(factor(cyl)))
# Default plotting
c + geom_bar()
# To change the interior colouring use fill aesthetic
c + geom_bar(fill = "red")
# Compare with the colour aesthetic which changes just the bar outline
c + geom_bar(colour = "red")
# Combining both, you can see the changes more clearly
c + geom_bar(fill = "white", colour = "red")

# The aesthetic fill also takes different colouring scales
# setting fill equal to a factor variable uses a discrete colour scale
k <- ggplot(mtcars, aes(factor(cyl), fill = factor(vs)))
k + geom_bar()

# Fill aesthetic can also be used with a continuous variable
m <- ggplot(faithful, aes(waiting, eruptions))
m + geom_raster()
m + geom_raster(aes(fill = density))

# Some geoms don't use both aesthetics (i.e. geom_point or geom_line)
b <- ggplot(economics, aes(x = date, y = unemploy))
b + geom_line()
b + geom_line(colour = "green")
b + geom_point()
b + geom_point(colour = "red")

# For large datasets with overplotting the alpha
# aesthetic will make the points more transparent
df <- data.frame(x = rnorm(5000), y = rnorm(5000))
h <- ggplot(df, aes(x,y))
h + geom_point()
```
h + geom_point(alpha = 0.5)
h + geom_point(alpha = 1/10)

# Alpha can also be used to add shading
j <- b + geom_line()
j
yrng <- range(economics$unemploy)
j <- j + geom_rect(aes(NULL, NULL, xmin = start, xmax = end, fill = party),
ymin = yrng[1], ymax = yrng[2], data = presidential)
j
j + scale_fill_manual(values = alpha(c("blue", "red"), .3))

--------

aes_group_order  Aesthetics: group

Description

Aesthetics: group

Examples

# By default, the group is set to the interaction of all discrete variables in the
# plot. This often partitions the data correctly, but when it does not, or when
# no discrete variable is used in the plot, you will need to explicitly define the
# grouping structure, by mapping group to a variable that has a different value
# for each group.

# For most applications you can simply specify the grouping with
# various aesthetics (colour, shape, fill, linetype) or with facets.

p <- ggplot(mtcars, aes(wt, mpg))
# A basic scatter plot
p + geom_point(size = 4)
# The colour aesthetic
p + geom_point(aes(colour = factor(cyl)), size = 4)
# Or you can use shape to distinguish the data
p + geom_point(aes(shape = factor(cyl)), size = 4)

# Using fill
a <- ggplot(mtcars, aes(factor(cyl)))
a + geom_bar()
a + geom_bar(aes(fill = factor(cyl)))
a + geom_bar(aes(fill = factor(vs)))

# Using linetypes
rescalePQ <- function(x) (x - min(x)) / diff(range(x))
ec_scaled <- data.frame(
date = economics$date,
plyr::colwise(rescale)(economics[, -(1:2)])
ecm <- reshape2::melt(ec_scaled, id.vars = "date")
f <- ggplot(ecm, aes(date, value))
f + geom_line(aes(linetype = variable))

# Using facets
k <- ggplot(diamonds, aes(carat, ..density..)) + geom_histogram(binwidth = 0.2)
k + facet_grid(~ cut)

# There are three common cases where the default is not enough, and we
# will consider each one below. In the following examples, we will use a simple
# longitudinal dataset, Oxboys, from the nlme package. It records the heights
# (height) and centered ages (age) of 26 boys (Subject), measured on nine
# occasions (Occasion).

# Multiple groups with one aesthetic
h <- ggplot(nlme::Oxboys, aes(age, height))
# A single line tries to connect all the observations
h + geom_line()
# The group aesthetic maps a different line for each subject
h + geom_line(aes(group = Subject))

# Different groups on different layers
h <- h + geom_line(aes(group = Subject))
# Using the group aesthetic with both geom_line() and geom_smooth()
# groups the data the same way for both layers
h + geom_smooth(aes(group = Subject), method = "lm", se = FALSE)
# Changing the group aesthetic for the smoother layer
# fits a single line of best fit across all boys
h + geom_smooth(aes(group = 1), size = 2, method = "lm", se = FALSE)

# Overriding the default grouping
# The plot has a discrete scale but you want to draw lines that connect across
# groups. This is the strategy used in interaction plots, profile plots, and parallel
# coordinate plots, among others. For example, we draw boxplots of height at
# each measurement occasion
boysbox <- ggplot(nlme::Oxboys, aes(Occasion, height))
boysbox + geom_boxplot()
# There is no need to specify the group aesthetic here; the default grouping
# works because occasion is a discrete variable. To overlay individual trajectories
# we again need to override the default grouping for that layer with aes(group = Subject)
boysbox <- boysbox + geom_boxplot()
boysbox + geom_line(aes(group = Subject), colour = "blue")

Differentiation related aesthetics: linetype, size, shape
Description

This page demonstrates the usage of a sub-group of aesthetics; linetype, size and shape.

Examples

# Line types should be specified with either an integer, a name, or with a string of
# an even number (up to eight) of hexadecimal digits which give the lengths in
# consecutive positions in the string.
# 0 = blank, 1 = solid, 2 = dashed, 3 = dotted, 4 = dotdash, 5 = longdash, 6 = twodash

# Data
df <- data.frame(x = 1:10, y = 1:10)
f <- ggplot(df, aes(x, y))
f + geom_line(linetype = 2)
f + geom_line(linetype = "dotdash")

# An example with hex strings, the string "33" specifies three units on followed
# by three off and "3313" specifies three units on followed by three off followed
# by one on and finally three off.
f + geom_line(linetype = "3313")

# Mapping line type from a variable
ggplot(economics_long, aes(date, value)) +
  geom_line(aes(linetype = variable))

# Size examples
# Should be specified with a numerical value (in millimetres),
# or from a variable source
p <- ggplot(mtcars, aes(wt, mpg))
p + geom_point(size = 4)
p + geom_point(aes(size = qsec))
p + geom_point(size = 2.5) +
  geom_hline(yintercept = 25, size = 3.5)

# Shape examples
# Shape takes four types of values: an integer in [0, 25],
# a single character---which uses that character as the plotting symbol,
# a . to draw the smallest rectangle that is visible (i.e., about one pixel)
# an NA to draw nothing
p + geom_point()
p + geom_point(shape = 5)
p + geom_point(shape = "k", size = 3)
p + geom_point(shape = ".")
p + geom_point(shape = NA)

# Shape can also be mapped from a variable
p + geom_point(aes(shape = factor(cyl)))

# A look at all 25 symbols
df2 <- data.frame(x = 1:5, y = 1:25, z = 1:25)
s <- ggplot(df2, aes(x, y))
This page demonstrates the usage of a sub-group of aesthetics: x, y, xmin, xmax, ymin, ymax, xend, and yend.

Examples

# Generate data: means and standard errors of means for prices
# for each type of cut
dmod <- lm(price ~ cut, data = diamonds)
cuts <- data.frame(cut = unique(diamonds$cut), predict(dmod, data.frame(cut = unique(diamonds$cut)), se = TRUE)[c("fit", "se.fit")])
se <- ggplot(cuts, aes(x = cut, y = fit, ymin = fit - se.fit, ymax = fit + se.fit, colour = cut))
se + geom_pointrange()

# Using annotate
p <- ggplot(mtcars, aes(wt, mpg)) + geom_point()
p + annotate("rect", xmin = 2, xmax = 3.5, ymin = 2, ymax = 25, fill = "dark grey", alpha = .5)

# Geom_segment examples
p + geom_segment(aes(x = 2, y = 15, xend = 2, yend = 25),
                 arrow = arrow(length = unit(0.5, "cm")))
p + geom_segment(aes(x = 2, y = 15, xend = 3, yend = 15),
                 arrow = arrow(length = unit(0.5, "cm")))
p + geom_segment(aes(x = 5, y = 30, xend = 3.5, yend = 25),
                 arrow = arrow(length = unit(0.5, "cm")))

# You can also use geom_segment to recreate plot(type = "h"):
counts <- as.data.frame(table(x = rpois(100, 5)))
counts$x <- as.numeric(as.character(counts$x))
with(counts, plot(x, Freq, type = "h", lwd = 10))
ggplot(counts, aes(x, Freq)) +
   geom_segment(aes(yend = 0, xend = x), size = 10)
annotate

Create an annotation layer.

Description

This function adds geoms to a plot. Unlike typical a geom function, the properties of the geoms are not mapped from variables of a data frame, but are instead passed in as vectors. This is useful for adding small annotations (such as text labels) or if you have your data in vectors, and for some reason don’t want to put them in a data frame.

Usage

annotate(geom, x = NULL, y = NULL, xmin = NULL, xmax = NULL, ymin = NULL, ymax = NULL, xend = NULL, yend = NULL, ..., na.rm = FALSE)

Arguments

geom

name of geom to use for annotation

x, y, xmin, ymin, xmax, ymax, xend, yend

positioning aesthetics - you must specify at least one of these.

... other aesthetics. These are not scaled so you can do (e.g.) colour = "red" to get a red point.

na.rm If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

Details

Note that all position aesthetics are scaled (i.e. they will expand the limits of the plot so they are visible), but all other aesthetics are set. This means that layers created with this function will never affect the legend.

Examples

p <- ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point()
p + annotate("text", x = 4, y = 25, label = "Some text")
p + annotate("text", x = 2:5, y = 25, label = "Some text")
p + annotate("rect", xmin = 3, xmax = 4.2, ymin = 12, ymax = 21, alpha = .2)
p + annotate("segment", x = 2.5, xend = 4, y = 15, yend = 25, colour = "blue")
p + annotate("pointrange", x = 3.5, y = 20, ymin = 12, ymax = 28, colour = "red", size = 1.5)
p + annotate("text", x = 2:3, y = 20:21, label = c("my label", "label 2"))
annotation_custom  Ann\textit{notation: Custom grob.}

**Description**

This is a special geom intended for use as static annotations that are the same in every panel. These annotations will not affect scales \(\text{i.e.}\) the x and y axes will not grow to cover the range of the grob, and the grob will not be modified by any ggplot settings or mappings.

**Usage**

```r
annotation_custom(grob, xmin = -Inf, xmax = Inf, ymin = -Inf, ymax = Inf)
```

**Arguments**

- `grob` grob to display
- `xmin, xmax` x location (in data coordinates) giving horizontal location of raster
- `ymin, ymax` y location (in data coordinates) giving vertical location of raster

**Details**

Most useful for adding tables, inset plots, and other grid-based decorations.

**Note**

`annotation_custom` expects the grob to fill the entire viewport defined by xmin, xmax, ymin, ymax. Grobs with a different (absolute) size will be center-justified in that region. Inf values can be used to fill the full plot panel (see examples).

**Examples**

```r
# Dummy plot
df <- data.frame(x = 1:10, y = 1:10)
base <- ggplot(df, aes(x, y)) +
  geom_blank() +
  theme_bw()
# Adding a table
## Not run:
if (require(gridExtra)) {
  base + annotation_custom(grob = tableGrob(head(iris[,1:3]),
                                          xmin = 3, xmax = 6, ymin = 2, ymax = 8))
}
# full panel
base + annotation_custom(grob = roundrectGrob(),
                         xmin = -Inf, xmax = Inf, ymin = -Inf, ymax = Inf)
}
## End(Not run)
```
# Inset plot

```r
df2 <- data.frame(x = 1, y = 1)
g <- ggplotGrob(ggplot(df2, aes(x, y)) +
  geom_point() +
  theme(plot.background = element_rect(colour = "black")))
base +
  annotation_custom(grob = g, xmin = 1, xmax = 10, ymin = 8, ymax = 10)
```
See Also

`scale_y_continuous`, `scale_y_log10` for log scale transformations.
`coord_trans` for log coordinate transformations.

Examples

```r
# Make a log-log plot (without log ticks)
a <- ggplot(msleep, aes(bodywt, brainwt)) +
  geom_point(na.rm = TRUE) +
  scale_x_log10(
    breaks = scales::trans_breaks("log10", function(x) 10^x),
    labels = scales::trans_format("log10", scales::math_format(10^x))
  ) +
  scale_y_log10(
    breaks = scales::trans_breaks("log10", function(x) 10^x),
    labels = scales::trans_format("log10", scales::math_format(10^x))
  ) +
  theme_bw()
a + annotation_logticks()  # Default: log ticks on bottom and left
a + annotation_logticks(sides = "lr")  # Log ticks for y, on left and right
a + annotation_logticks(sides = "trbl")  # All four sides

# Hide the minor grid lines because they don't align with the ticks
a + annotation_logticks(sides = "trbl") + theme(panel.grid.minor = element_blank())

# Another way to get the same results as 'a' above: log-transform the data before plotting it. Also hide the minor grid lines.
b <- ggplot(msleep, aes(log10(bodywt), log10(brainwt))) +
  geom_point(na.rm = TRUE) +
  scale_x_continuous(name = "body", labels = scales::math_format(10^x)) +
  scale_y_continuous(name = "brain", labels = scales::math_format(10^x)) +
  theme_bw() + theme(panel.grid.minor = element_blank())
b + annotation_logticks()

# Using a coordinate transform requires scaled = FALSE
t <- ggplot(msleep, aes(bodywt, brainwt)) +
  geom_point() +
  coord_trans(x = "log10", y = "log10") +
  theme_bw()
t + annotation_logticks(scaled = FALSE)

# Change the length of the ticks
a + annotation_logticks(
  short = unit(.5,"mm"),
  mid = unit(3,"mm"),
  long = unit(4,"mm")
)
```

**annotation_map**

Annotation: maps.

**Description**

Annotation: maps.

**Usage**

```r
annotation_map(map, ...)```

**Arguments**

- `map`: data frame representing a map. Most map objects can be converted into the right format by using `fortify`
- `...`: other arguments used to modify aesthetics

**Examples**

```r
if (require("maps")) {
  usamap <- map_data("state")

  seal.sub <- subset(seals, long > -130 & lat < 45 & lat > 40)
  ggplot(seal.sub, aes(x = long, y = lat)) +
    annotation_map(usamap, fill = "NA", colour = "grey50") +
    geom_segment(aes(xend = long + delta_long, yend = lat + delta_lat))

  seal2 <- transform(seal.sub,
    latr = cut(lat, 2),
    longr = cut(long, 2))

  ggplot(seal2, aes(x = long, y = lat)) +
    annotation_map(usamap, fill = "NA", colour = "grey50") +
    geom_segment(aes(xend = long + delta_long, yend = lat + delta_lat)) +
    facet_grid(latr ~ longr, scales = "free", space = "free")
}
```

**annotation_raster**

Annotation: High-performance rectangular tiling.

**Description**

This is a special version of `geom_raster` optimised for static annotations that are the same in every panel. These annotations will not affect scales (i.e. the x and y axes will not grow to cover the range of the raster, and the raster must already have its own colours).
Usage

```
annotation_raster(raster, xmin, xmax, ymin, ymax, interpolate = FALSE)
```

Arguments

- `raster`  
  - raster object to display
- `xmin`, `xmax`  
  - x location (in data coordinates) giving horizontal location of raster
- `ymin`, `ymax`  
  - y location (in data coordinates) giving vertical location of raster
- `interpolate`  
  - If TRUE interpolate linearly, if FALSE (the default) don’t interpolate.

Details

Most useful for adding bitmap images.

Examples

```r
# Generate data
rainbow <- matrix(hcl(seq(0, 360, length.out = 50 * 50), 80, 70), nrow = 50)
ggplot(mtcars, aes(mpg, wt)) +
  geom_point() +
  annotation_raster(rainbow, 15, 20, 3, 4)
# To fill up whole plot
ggplot(mtcars, aes(mpg, wt)) +
  annotation_raster(rainbow, -Inf, Inf, -Inf, Inf) +
  geom_point()
```

---

**as.list.ggproto**  
*Convert a ggproto object to a list*

Description

This will not include the object’s super member.

Usage

```r
## S3 method for class 'ggproto'
as.list(x, inherit = TRUE, ...)
```
as_labeller

Arguments

- **x**: A ggproto object to convert to a list.
- **inherit**: If `TRUE` (the default), flatten all inherited items into the returned list. If `FALSE`, do not include any inherited items.
- **...**: Further arguments to pass to `as.list.environment`.

Coerce to labeller function

Description

This transforms objects to labeller functions. Used internally by `labeller()`.

Usage

`as_labeller(x, default = label_value, multi_line = TRUE)`

Arguments

- **x**: Object to coerce to a labeller function. If a named character vector, it is used as a lookup table before being passed on to `default`. If a non-labeller function, it is assumed it takes and returns character vectors and is applied to the labels. If a labeller, it is simply applied to the labels.
- **default**: Default labeller to process the labels produced by lookup tables or modified by non-labeller functions.
- **multi_line**: Whether to display the labels of multiple factors on separate lines. This is passed to the labeller function.

See Also

`labeller()`, `labellers`

Examples

```r
p <- ggplot(mtcars, aes(disp, drat)) + geom_point()
p + facet_wrap(~am)

# Rename labels on the fly with a lookup character vector
to_string <- as_labeller(c('0' = "Zero", '1' = "One"))
p + facet_wrap(~am, labeller = to_string)

# Quickly transform a function operating on character vectors to a labeller function:
appender <- function(string, suffix = "-foo") paste0(string, suffix)
p + facet_wrap(~am, labeller = as_labeller(appender))

# If you have more than one facetting variable, be sure to dispatch your labeller to the right variable with labeller()
p + facet_grid(cyl ~ am, labeller = labeller(am = to_string))
```
**autoplot**  
*Create a complete ggplot appropriate to a particular data type*

**Description**

autoplot uses ggplot2 to draw a particular plot for an object of a particular class in a single command. This defines the S3 generic that other classes and packages can extend.

**Usage**

`autoplot(object, ...)`

**Arguments**

- `object` an object, whose class will determine the behaviour of autoplot
- `...` other arguments passed to specific methods

**Value**

a ggplot object

**See Also**

ggplot and fortify

---

**borders**  
*Create a layer of map borders.*

**Description**

Create a layer of map borders.

**Usage**

`borders(database = "world", regions = ".", fill = NA, colour = "grey50", xlim = NULL, ylim = NULL, ...)`

**Arguments**

- `database` map data, see `map` for details
- `regions` map region
- `fill` fill colour
- `colour` border colour
- `xlim, ylim` latitudinal and logistical range for extracting map polygons, see `map` for details.
- `...` other arguments passed onto `geom_polygon`
calc_element

Examples

```r
if (require("maps")) {

  ia <- map_data("county", "iowa")
  mid_range <- function(x) mean(range(x))
  seats <- plyr::ddply(ia, "subregion", plyr::colwise(mid_range, c("lat", "long")))
  ggplot(ia, aes(long, lat)) +
    geom_polygon(aes(group = group), fill = NA, colour = "grey60") +
    geom_text(aes(label = subregion), data = seats, size = 2, angle = 45)

  data(us.cities)
  capitals <- subset(us.cities, capital == 2)
  ggplot(capitals, aes(long, lat)) +
    borders("state") +
    geom_point(aes(size = pop)) +
    scale_size_area() +
    coord_quickmap()

  # Same map, with some world context
  ggplot(capitals, aes(long, lat)) +
    borders("world", xlim = c(-130, -60), ylim = c(20, 50)) +
    geom_point(aes(size = pop)) +
    scale_size_area() +
    coord_quickmap()
}
```

calc_element

**Calculate the element properties, by inheriting properties from its parents**

Description

Calculate the element properties, by inheriting properties from its parents

Usage

calc_element(element, theme, verbose = FALSE)

Arguments

element The name of the theme element to calculate
theme A theme object (like theme_grey())
verbose If TRUE, print out which elements this one inherits from
Examples

```r
t <- theme_grey()
calc_element('text', t)

# Compare the "raw" element definition to the element with calculated inheritance
t$axis.text.x
calc_element('axis.text.x', t, verbose = TRUE)

# This reports that axis.text.x inherits from axis.text,
# which inherits from text. You can view each of them with:
t$axis.text.x
t$axis.text
t$text
```

---

**coord_cartesian**

*Cartesian coordinates.*

Description

The Cartesian coordinate system is the most familiar, and common, type of coordinate system. Setting limits on the coordinate system will zoom the plot (like you’re looking at it with a magnifying glass), and will not change the underlying data like setting limits on a scale will.

Usage

```r
coord_cartesian(xlim = NULL, ylim = NULL, expand = TRUE)
```

Arguments

- `xlim`, `ylim` Limits for the x and y axes.
- `expand` If TRUE, the default, adds a small expansion factor to the limits to ensure that data and axes don’t overlap. If FALSE, limits are taken exactly from the data or xlim/ylim.

Examples

```r
# There are two ways of zooming the plot display: with scales or
# with coordinate systems. They work in two rather different ways.

p <- ggplot(mtcars, aes(disp, wt)) +
  geom_point() +
  geom_smooth()
p

# Setting the limits on a scale converts all values outside the range to NA.
p + scale_x_continuous(limits = c(325, 500))

# Setting the limits on the coordinate system performs a visual zoom.
```
# The data is unchanged, and we just view a small portion of the original plot. Note how smooth continues past the points visible on this plot.
p + coord_cartesian(xlim = c(325, 500))

# By default, the same expansion factor is applied as when setting scale limits. You can set the limits precisely by setting expand = FALSE
p + coord_cartesian(xlim = c(325, 500), expand = FALSE)

# Similarly, we can use expand = FALSE to turn off expansion with the default limits
p + coord_cartesian(expand = FALSE)

# You can see the same thing with this 2d histogram
d <- ggplot(diamonds, aes(carat, price)) +
  stat_bin2d(bins = 25, colour = "white")
d
# When zooming the scale, the we get 25 new bins that are the same size on the plot, but represent smaller regions of the data space
d + scale_x_continuous(limits = c(0, 1))

# When zooming the coordinate system, we see a subset of original 50 bins, displayed bigger
d + coord_cartesian(xlim = c(0, 1))

---

**coord_fixed**

Cartesian coordinates with fixed relationship between x and y scales.

### Description

A fixed scale coordinate system forces a specified ratio between the physical representation of data units on the axes. The ratio represents the number of units on the y-axis equivalent to one unit on the x-axis. The default, ratio = 1, ensures that one unit on the x-axis is the same length as one unit on the y-axis. Ratios higher than one make units on the y-axis longer than units on the x-axis, and vice versa. This is similar to eqscplot, but it works for all types of graphics.

### Usage

coord_fixed(ratio = 1, xlim = NULL, ylim = NULL, expand = TRUE)

### Arguments

- **ratio**: aspect ratio, expressed as \( \frac{y}{x} \)
- **xlim**: Limits for the x and y axes.
- **ylim**: Limits for the x and y axes.
- **expand**: If TRUE, the default, adds a small expansion factor to the limits to ensure that data and axes don’t overlap. If FALSE, limits are taken exactly from the data or xlim/ylim.
Examples

# ensures that the ranges of axes are equal to the specified ratio by
# adjusting the plot aspect ratio

p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
p + coord_fixed(ratio = 1)
p + coord_fixed(ratio = 5)
p + coord_fixed(ratio = 1/5)

# Resize the plot to see that the specified aspect ratio is maintained

---

coord_flip  

Flipped cartesian coordinates.

Description

Flipped cartesian coordinates so that horizontal becomes vertical, and vertical, horizontal. This is primarily useful for converting geoms and statistics which display y conditional on x, to x conditional on y.

Usage

coord_flip(xlim = NULL, ylim = NULL, expand = TRUE)

Arguments

xlim  

Limits for the x and y axes.

ylim  

Limits for the x and y axes.

expand  

If TRUE, the default, adds a small expansion factor to the limits to ensure that data and axes don’t overlap. If FALSE, limits are taken exactly from the data or xlim/ylim.

Examples

# Very useful for creating boxplots, and other interval
# geoms in the horizontal instead of vertical position.

ggplot(diamonds, aes(cut, price)) +
  geom_boxplot() +
  coord_flip()

h <- ggplot(diamonds, aes(carat)) +
  geom_histogram()

h + coord_flip()

# You can also use it to flip line and area plots:
coord_map

The representation of a portion of the earth, which is approximately spherical, onto a flat 2D plane requires a projection. This is what `coord_map` does. These projections account for the fact that the actual length (in km) of one degree of longitude varies between the equator and the pole. Near the equator, the ratio between the lengths of one degree of latitude and one degree of longitude is approximately 1. Near the pole, it is tends towards infinity because the length of one degree of longitude tends towards 0. For regions that span only a few degrees and are not too close to the poles, setting the aspect ratio of the plot to the appropriate lat/lon ratio approximates the usual mercator projection. This is what `coord_quickmap` does. With `coord_map` all elements of the graphic have to be projected which is not the case here. So `coord_quickmap` has the advantage of being much faster, in particular for complex plots such as those using with `geom_tile`, at the expense of correctness in the projection. This coordinate system provides the full range of map projections available in the mapproj package.

Usage

```r
coord_map(projection = "mercator", ..., orientation = NULL, xlim = NULL, ylim = NULL)
coord_quickmap(xlim = NULL, ylim = NULL, expand = TRUE)
```

Arguments

- **projection**: projection to use, see `mapproject` for list
- **...**: other arguments passed on to `mapproject`
- **orientation**: projection orientation, which defaults to `c(90, 0, mean(range(x)))`. This is not optimal for many projections, so you will have to supply your own. See `mapproject` for more information.
- **xlim**: manually specific x limits (in degrees of longitude)
- **ylim**: manually specific y limits (in degrees of latitude)
- **expand**: If TRUE, the default, adds a small expansion factor to the limits to ensure that data and axes don’t overlap. If FALSE, limits are taken exactly from the data or `xlim/ylim`. 
Examples

if (require("maps")) {
  nz <- map_data("nz")
  # Prepare a map of NZ
  nzmap <- ggplot(nz, aes(x = long, y = lat, group = group)) +
           geom_polygon(fill = "white", colour = "black")
  # Plot it in cartesian coordinates
  nzmap
  # With correct mercator projection
  nzmap + coord_map()
  # With the aspect ratio approximation
  nzmap + coord_quickmap()
  # Other projections
  nzmap + coord_map("cylindrical")
  nzmap + coord_map("azequalarea", orientation = c(-36.92, 174.6, 0))
  states <- map_data("state")
  usamap <- ggplot(states, aes(long, lat, group = group)) +
           geom_polygon(fill = "white", colour = "black")
  # Use cartesian coordinates
  usamap
  # With mercator projection
  usamap + coord_map()
  usamap + coord_quickmap()
  # See ?mapproject for coordinate systems and their parameters
  usamap + coord_map("gilbert")
  usamap + coord_map("lagrange")
  # For most projections, you'll need to set the orientation yourself
  # as the automatic selection done by mapproject is not available to
  # ggplot
  usamap + coord_map("orthographic")
  usamap + coord_map("stereographic")
  usamap + coord_map("conic", lat0 = 30)
  usamap + coord_map("bonne", lat0 = 50)
  # World map, using geom_path instead of geom_polygon
  world <- map_data("world")
  worldmap <- ggplot(world, aes(x = long, y = lat, group = group)) +
              geom_path() +
              scale_y_continuous(breaks = (-2:2) * 30) +
              scale_x_continuous(breaks = (-4:4) * 45)
  # Orthographic projection with default orientation (looking down at North pole)
  worldmap + coord_map("ortho")
  # Looking up up at South Pole
  worldmap + coord_map("ortho", orientation = c(-90, 0, 0))
  # Centered on New York (currently has issues with closing polygons)
  worldmap + coord_map("ortho", orientation = c(41, -74, 0))
}
Description

The polar coordinate system is most commonly used for pie charts, which are a stacked bar chart in polar coordinates.

Usage

coord_polar(theta = "x", start = 0, direction = 1)

Arguments

theta variable to map angle to (x or y)
start offset of starting point from 12 o’clock in radians
direction 1, clockwise; -1, anticlockwise

Examples

# NOTE: Use these plots with caution – polar coordinates has
# major perceptual problems. The main point of these examples is
# to demonstrate how these common plots can be described in the
# grammar. Use with EXTREME caution.

#' # A pie chart = stacked bar chart + polar coordinates
pie <- ggplot(mtcars, aes(x = factor(1), fill = factor(cyl))) +
  geom_bar(width = 1)
pie + coord_polar(theta = "y")

# A coxcomb plot = bar chart + polar coordinates
cxc <- ggplot(mtcars, aes(x = factor(cyl))) +
  geom_bar(width = 1, colour = "black")
cxc + coord_polar()
# A new type of plot?
cxc + coord_polar(theta = "y")

# The bullseye chart
pie + coord_polar()

# Hadley’s favourite pie chart
df <- data.frame(
  variable = c("does not resemble", "resembles"),
  value = c(20, 80))
)
coord_trans

Transformed cartesian coordinate system.

Description

coord_trans is different to scale transformations in that it occurs after statistical transformation and will affect the visual appearance of geoms - there is no guarantee that straight lines will continue to be straight.

Usage

coord_trans(x = "identity", y = "identity", limx = NULL, limy = NULL, xtrans, ytrans)

Arguments

x, y
transformers for x and y axes

limx, limy
limits for x and y axes. (Named so for backward compatibility)

xtrans, ytrans
Deprecated; use x and y instead.

Details

All current transformations only work with continuous values - see trans_new for list of transformations, and instructions on how to create your own.
Examples

# See ?geom_boxplot for other examples

# Three ways of doing transformation in ggplot:
# * by transforming the data
ggplot(diamonds, aes(log10(carat), log10(price))) +
  geom_point()
# * by transforming the scales
ggplot(diamonds, aes(carat, price)) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10()
# * by transforming the coordinate system:
ggplot(diamonds, aes(carat, price)) +
  geom_point() +
  coord_trans(x = "log10", y = "log10")

# The difference between transforming the scales and
# transforming the coordinate system is that scale
# transformation occurs BEFORE statistics, and coordinate
# transformation afterwards. Coordinate transformation also
# changes the shape of geoms:

d <- subset(diamonds, carat > 0.5)

ggplot(d, aes(carat, price)) +
  geom_point() +
  geom_smooth(method = "lm") +
  scale_x_log10() +
  scale_y_log10()

ggplot(d, aes(carat, price)) +
  geom_point() +
  geom_smooth(method = "lm") +
  coord_trans(x = "log10", y = "log10")

# Here I used a subset of diamonds so that the smoothed line didn't
# drop below zero, which obviously causes problems on the log-transformed
# scale

# With a combination of scale and coordinate transformation, it's
# possible to do back-transformations:

ggplot(diamonds, aes(carat, price)) +
  geom_point() +
  geom_smooth(method = "lm") +
  scale_x_log10() +
  scale_y_log10() +
  coord_trans(x = scales::exp_trans(10), y = scales::exp_trans(10))

# cf.
ggplot(diamonds, aes(carat, price)) +
```r
geom_point() +
geom_smooth(method = "lm")

# Also works with discrete scales
df <- data.frame(a = abs(rnorm(26)), letters)
plot <- ggplot(df, aes(a, letters)) + geom_point()
plot + coord_trans(x = "log10")
plot + coord_trans(x = "sqrt")
```

---

**cut_interval**

```r
cut_interval Help
```  

**Description**

cut_interval makes \( n \) groups with equal range, cut_number makes \( n \) groups with (approximately) equal numbers of observations; cut_width makes groups of width width.

**Usage**

cut_interval(x, n = NULL, length = NULL, ...)

cut_number(x, n = NULL, ...)

cut_width(x, width, center = NULL, boundary = NULL, closed = c("right", "left"))

**Arguments**

- \( x \) numeric vector
- \( n \) number of intervals to create, OR length
- \( \ldots \) other arguments passed on to `cut`
- width The bin width.
- center, boundary

Specify either the position of edge or the center of a bin. Since all bins are aligned, specifying the position of a single bin (which doesn’t need to be in the range of the data) affects the location of all bins. If not specified, uses the "tile layers algorithm", and sets the boundary to half of the binwidth. To center on integers, width = 1 and center = 0, boundary = 0.5.

- closed One of "right" or "left" indicating whether right or left edges of bins are included in the bin.

**Author(s)**

Randall Prium contributed most of the implementation of cut_width.
**diamonds**

**See Also**

cut_number

**Examples**

```r
table(cut_interval(1:100, 10))
table(cut_interval(1:100, 11))

table(cut_number(runif(1000), 10))

table(cut_width(runif(1000), 0.1))
table(cut_width(runif(1000), 0.1, boundary = 0))
table(cut_width(runif(1000), 0.1, center = 0))
```

---

**Description**

A dataset containing the prices and other attributes of almost 54,000 diamonds. The variables are as follows:

**Usage**

diamonds

**Format**

A data frame with 53940 rows and 10 variables:

- price: price in US dollars ($326–$18,823)
- carat: weight of the diamond (0.2–5.01)
- cut: quality of the cut (Fair, Good, Very Good, Premium, Ideal)
- color: diamond colour, from J (worst) to D (best)
- clarity: a measurement of how clear the diamond is (I1 (worst), SI1, SI2, VS1, VS2, VVS1, VVS2, IF (best))
- x: length in mm (0–10.74)
- y: width in mm (0–58.9)
- z: depth in mm (0–31.8)
- depth: total depth percentage = \( z / \text{mean}(x, y) = 2 \times z / (x + y) \) (43–79)
- table: width of top of diamond relative to widest point (43–95)
Description

This dataset was produced from US economic time series data available from http://research.stlouisfed.org/fred2. economics is in "wide" format, economics_long is in "long" format.

Usage

economics

economics_long

Format

A data frame with 478 rows and 6 variables

- date. Month of data collection
- psavert, personal savings rate, http://research.stlouisfed.org/fred2/series/PSAVER/
- pce, personal consumption expenditures, in billions of dollars, http://research.stlouisfed.org/fred2/series/PCE
- unemploy, number of unemployed in thousands, http://research.stlouisfed.org/fred2/series/UNEMPLOY
- uempmed, median duration of unemployment, in week, http://research.stlouisfed.org/fred2/series/UEMPMED

Description

Theme element: blank. This theme element draws nothing, and assigns no space

Usage

element_blank()
element_line

| Description | Theme element: line.
|-------------|---------------------
| Usage       | `element_line(colour = NULL, size = NULL, linetype = NULL, lineend = NULL, color = NULL)`
| Arguments   | colour: line colour  
size: line size  
linetype: line type  
lineend: line end  
color: an alias for colour

element_rect

| Description | Theme element: rectangle.
|-------------|---------------------
| Usage       | `element_rect(fill = NULL, colour = NULL, size = NULL, linetype = NULL, color = NULL)`
| Arguments   | fill: fill colour  
colour: border colour  
size: border size  
linetype: border linetype  
color: an alias for colour
element_text  Theme element: text.

Description
Theme element: text.

Usage

```r
element_text(family = NULL, face = NULL, colour = NULL, size = NULL, 
hjust = NULL, vjust = NULL, angle = NULL, lineheight = NULL, 
color = NULL, margin = NULL, debug = NULL)
```

Arguments

- **family**: font family
- **face**: font face ("plain", "italic", "bold", "bold.italic")
- **colour**/**color**: text colour
- **size**: text size (in pts)
- **hjust**: horizontal justification (in [0, 1])
- **vjust**: vertical justification (in [0, 1])
- **angle**: angle (in [0, 360])
- **lineheight**: line height
- **margin**: margins around the text. See `margin` for more details. When creating a theme, the margins should be placed on the side of the text facing towards the center of the plot.
- **debug**: If TRUE, aids visual debugging by drawing a solid rectangle behind the complete text area, and a point where each label is anchored.

expand_limits  Expand the plot limits with data.

Description
panels or all plots. This function is a thin wrapper around `geom_blank` that makes it easy to add such values.

Usage

```r
expand_limits(...)```

**Arguments**

... named list of aesthetics specifying the value (or values) that should be included in each scale.

**Examples**

```r
p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
p + expand_limits(x = 0)
p + expand_limits(y = c(1, 9))
p + expand_limits(x = 0, y = 0)

ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(colour = cyl)) +
  expand_limits(colour = seq(2, 10, by = 2))
ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(colour = factor(cyl))) +
  expand_limits(colour = factor(seq(2, 10, by = 2)))
```

---

**facet_grid**  
* Lay out panels in a grid.

**Description**

Lay out panels in a grid.

**Usage**

```r
facet_grid(facets, margins = FALSE, scales = "fixed", space = "fixed",
shrink = TRUE, labeller = "label_value", as.table = TRUE,
switch = NULL, drop = TRUE)
```

**Arguments**

- **facets**  
  a formula with the rows (of the tabular display) on the LHS and the columns (of the tabular display) on the RHS; the dot in the formula is used to indicate there should be no faceting on this dimension (either row or column). The formula can also be provided as a string instead of a classical formula object.

- **margins**  
  either a logical value or a character vector. Margins are additional facets which contain all the data for each of the possible values of the faceting variables. If FALSE, no additional facets are included (the default). If TRUE, margins are included for all faceting variables. If specified as a character vector, it is the names of variables for which margins are to be created.

- **scales**  
  Are scales shared across all facets (the default, "fixed"), or do they vary across rows ("free_x"), columns ("free_y"), or both rows and columns ("free")?

- **space**  
  If "fixed", the default, all panels have the same size. If "free_y" their height will be proportional to the length of the y scale; if "free_x" their width will be proportional to the length of the x scale; or if "free" both height and width will vary. This setting has no effect unless the appropriate scales also vary.
facet_grid

- **shrink**: If TRUE, will shrink scales to fit output of statistics, not raw data. If FALSE, will be range of raw data before statistical summary.
- **labeller**: A function that takes one data frame of labels and returns a list or data frame of character vectors. Each input column corresponds to one factor. Thus there will be more than one with formulae of the type ~cyl + am. Each output column gets displayed as one separate line in the strip label. This function should inherit from the "labeller" S3 class for compatibility with `labeller()`. See `label_value` for more details and pointers to other options.
- **as.table**: If TRUE, the default, the facets are laid out like a table with highest values at the bottom-right. If FALSE, the facets are laid out like a plot with the highest value at the top-right.
- **switch**: By default, the labels are displayed on the top and right of the plot. If "x", the top labels will be displayed to the bottom. If "y", the right-hand side labels will be displayed to the left. Can also be set to "both".
- **drop**: If TRUE, the default, all factor levels not used in the data will automatically be dropped. If FALSE, all factor levels will be shown, regardless of whether or not they appear in the data.

**Examples**

```r
p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
# With one variable
p + facet_grid(. ~ cyl)
p + facet_grid(cyl ~ .)

# With two variables
p + facet_grid(vs ~ am)
p + facet_grid(am ~ vs)
p + facet_grid(vs ~ am, margins=TRUE)

# To change plot order of facet grid,
# change the order of variable levels with factor()

set.seed(6809)
diamonds <- diamonds[sample(nrow(diamonds), 1000), ]
diamonds$cut <- factor(diamonds$cut,
    levels = c("Ideal", "Very Good", "Fair", "Good", "Premium"))

# Repeat first example with new order
p <- ggplot(diamonds, aes(carat, ..density..)) +
    geom_histogram(binwidth = 1)
p + facet_grid(. ~ cut)

g <- ggplot(mtcars, aes(mpg, wt)) +
    geom_point()
g + facet_grid(. ~ vs + am)
g + facet_grid(vs + am ~ .)

# You can also use strings, which makes it a little easier
```
When writing functions that generate faceting specifications:

```r
p + facet_grid("cut ~ .")
```

See also `?plotmatrix` for the scatterplot matrix.

If there isn’t any data for a given combination, that panel will be empty:

```r
g + facet_grid(cyl ~ vs)
```

If you combine a facetted dataset with a dataset that lacks those facetting variables, the data will be repeated across the missing combinations:

```r
g + facet_grid(vs ~ cyl)
```

```r
df <- data.frame(mpg = 22, wt = 3)
g + facet_grid(vs ~ cyl) + geom_point(data = df, colour = "red", size = 2)
```

```r
df2 <- data.frame(mpg = c(19, 22), wt = c(2,4), vs = c(0, 1))
g + facet_grid(vs ~ cyl) + geom_point(data = df2, colour = "red", size = 2)
```

```r
df3 <- data.frame(mpg = c(19, 22), wt = c(2,4), vs = c(1, 1))
g + facet_grid(vs ~ cyl) + geom_point(data = df3, colour = "red", size = 2)
```

You can also choose whether the scales should be constant across all panels (the default), or whether they should be allowed to vary:

```r
mt <- ggplot(mtcars, aes(mpg, wt, colour = factor(cyl))) + geom_point()
```

```r
mt + facet_grid(. ~ cyl, scales = "free")
```

If scales and space are free, then the mapping between position and values in the data will be the same across all panels:

```r
mt + facet_grid(. ~ cyl, scales = "free", space = "free")
```

```r
mt + facet_grid(vs ~ am, scales = "free")
```

```r
mt + facet_grid(vs ~ am, scales = "free_x")
```

```r
mt + facet_grid(vs ~ am, scales = "free_y")
```

```r
mt + facet_grid(vs ~ am, scales = "free", space = "free")
```

```r
mt + facet_grid(vs ~ am, scales = "free", space = "free_x")
```

```r
mt + facet_grid(vs ~ am, scales = "free", space = "free_y")
```

You may need to set your own breaks for consistent display:

```r
mt + facet_grid(. ~ cyl, scales = "free_x", space = "free") + scale_x_continuous(breaks = seq(10, 36, by = 2))
```

Adding scale limits override free scales:

```r
last_plot() + xlim(10, 15)
```
# Free scales are particularly useful for categorical variables
ggplot(mpg, aes(cty, model)) +
  geom_point() +
  facet_grid(manufacturer ~ ., scales = "free", space = "free")
# particularly when you reorder factor levels
mpg$model <- reorder(mpg$model, mpg$cty)
manufacturer <- reorder(mpg$manufacturer, mpg$cty)
last_plot() %>% mpg + theme(strip.text.y = element_text())

# Use as_table to control direction of horizontal facets, TRUE by default
h <- ggplot(mtcars, aes(x = mpg, y = wt)) + geom_point()
  h + facet_grid(cyl ~ vs)
  h + facet_grid(cyl ~ vs, as.table = FALSE)

# Use labeller to control facet labels, label_value is default
h + facet_grid(cyl ~ vs, labeller = label_both)
# Using label_parsed, see plotmath for more options
mtcars$cyl2 <- factor(mtcars$cyl, labels = c("alpha", "beta", "sqrt(x, y)"))
k <- ggplot(mtcars, aes(wt, mpg)) + geom_point()
  k + facet_grid(cyl ~ vs, labeller = label_parsed)
  # For label_bquote the label value is x.
  p <- ggplot(mtcars, aes(wt, mpg)) + geom_point()
  p + facet_grid(cyl ~ vs, labeller = label_bquote(alpha ~ .(x))
  p + facet_grid(cyl ~ vs, labeller = label_bquote(alpha ~ .(x))

# Margins can be specified by logically (all yes or all no) or by specific
# variables as (character) variable names
mg <- ggplot(mtcars, aes(x = mpg, y = wt)) + geom_point()
  mg + facet_grid(vs ~ am ~ gear)
  mg + facet_grid(vs ~ am ~ gear, margins = TRUE)
  mg + facet_grid(vs ~ am ~ gear, margins = "am")
  # when margins are made over "vs", since the facets for "am" vary
  # within the values of "vs", the marginal facet for "vs" is also
  # a margin over "am".
  mg + facet_grid(vs ~ am ~ gear, margins = "vs")
  mg + facet_grid(vs ~ am ~ gear, margins = "gear")
  mg + facet_grid(vs ~ am ~ gear, margins = c("gear", "am"))

# The facet strips can be displayed near the axes with switch
data <- transform(mtcars,
  am = factor(am, levels = 0:1, c("Automatic", "Manual")),
  gear = factor(gear, levels = 3:5, labels = c("Three", "Four", "Five"))
)
p <- ggplot(data, aes(mpg, disp)) + geom_point()
  p + facet_grid(am ~ gear, switch = "both") + theme_light()

# It may be more aesthetic to use a theme without boxes around
# around the strips.
facet_null

Facet specification: a single panel.

Description

Facet specification: a single panel.

Usage

facet_null(shrink = TRUE)

Arguments

shrink

If TRUE, will shrink scales to fit output of statistics, not raw data. If FALSE, will be range of raw data before statistical summary.

Examples

# facet_null is the default facetting specification if you
# don't override it with facet_grid or facet_wrap
ggplot(mtcars, aes(mpg, wt)) + geom_point()

facet_wrap

Wrap a 1d ribbon of panels into 2d.

Description

Most displays are roughly rectangular, so if you have a categorical variable with many levels, it doesn't make sense to try and display them all in one row (or one column). To solve this dilemma, facet_wrap wraps a 1d sequence of panels into 2d, making best use of screen real estate.

Usage

facet_wrap(facets, nrow = NULL, ncol = NULL, scales = "fixed",
shrink = TRUE, labeller = "label_value", as.table = TRUE,
switch = NULL, drop = TRUE, dir = "h")
Arguments

facets
Either a formula or character vector. Use either a one sided formula, ~a + b, or a character vector, c("a", "b").
nrow, ncol
Number of rows and columns.
scales
Should scales be fixed ("fixed", the default), free ("free"), or free in one dimension ("free_x", "free_y").
shrink
If TRUE, will shrink scales to fit output of statistics, not raw data. If FALSE, will be range of raw data before statistical summary.
labeller
A function that takes one data frame of labels and returns a list or data frame of character vectors. Each input column corresponds to one factor. Thus there will be more than one with formulae of the type ~cyl + am. Each output column gets displayed as one separate line in the strip label. This function should inherit from the "labeller" S3 class for compatibility with `labeller()`. See `label_value` for more details and pointers to other options.

as.table
If TRUE, the default, the facets are laid out like a table with highest values at the bottom-right. If FALSE, the facets are laid out like a plot with the highest value at the top-right.

switch
By default, the labels are displayed on the top of the plot. If switch is "x", they will be displayed to the bottom. If "y", they will be displayed to the left, near the y axis.

drop
If TRUE, the default, all factor levels not used in the data will automatically be dropped. If FALSE, all factor levels will be shown, regardless of whether or not they appear in the data.

dir
Direction: either "h" for horizontal, the default, or "v", for vertical.

Examples

```r
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(~class)

# Control the number of rows and columns with nrow and ncol
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(~class, nrow = 4)

# You can facet by multiple variables
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(~cyl + drv)
# Or use a character vector:
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(c("cyl", "drv"))

# Use the 'labeller' option to control how labels are printed:
```
faithful

A 2d density estimate of the waiting and eruptions variables data faithful.

Usage

faithful

Format

A data frame with 5,625 observations and 3 variables.
format.ggproto  
*Format a ggproto object*

**Description**

Format a ggproto object

**Usage**

```r
## S3 method for class 'ggproto'
format(x, ..., flat = TRUE)
```

**Arguments**

- `x`  
  A ggproto object to print.
- `...`  
  If the ggproto object has a `print` method, further arguments will be passed to it. Otherwise, these arguments are unused.
- `flat`  
  If TRUE (the default), show a flattened list of all local and inherited members. If FALSE, show the inheritance hierarchy.

---

fortify  
*Fortify a model with data.*

**Description**

Rather than using this function, I now recommend using the `broom` package, which implements a much wider range of methods. `fortify` may be deprecated in the future.

**Usage**

```r
fortify(model, data, ...)
```

**Arguments**

- `model`  
  model or other R object to convert to data frame
- `data`  
  original dataset, if needed
- `...`  
  other arguments passed to methods

**See Also**

`fortify.lm`
Description
Fortify methods for objects produced by `multcomp`

Usage

```r
## S3 method for class 'glht'
fortify(model, data, ...)

## S3 method for class 'confint.glht'
fortify(model, data, ...)

## S3 method for class 'summary.glht'
fortify(model, data, ...)

## S3 method for class 'cld'
fortify(model, data, ...)
```

Arguments

- `model`: an object of class `glht`, `confint.glht`, `summary.glht` or `cld`
- `data, ...`: other arguments to the generic ignored in this method.

Examples

```r
if (require("multcomp")) {
  amod <- aov(breaks ~ wool + tension, data = warpbreaks)
  wht <- glht(amod, linfct = mcp(tension = "Tukey"))

  fortify(wht)
  ggplot(wht, aes(lhs, estimate)) + geom_point()

  CI <- confint(wht)
  fortify(CI)
  ggplot(CI, aes(lhs, estimate, ymin = lwr, ymax = upr)) +
  geom_linerange() +
  geom_point(size = p, data = summary(wht)) +
  scale_size(trans = "reverse")

  cld <- cld(wht)
  fortify(cld)
}
```
fortify.lm

Supplement the data fitted to a linear model with model fit statistics.

Description

If you have missing values in your model data, you may need to refit the model with na.action = na.exclude.

Usage

## S3 method for class 'lm'
fortify(model, data = model$y, ...)  

Arguments

- model: linear model
- data: data set, defaults to data used to fit model
- ...: not used by this method

Value

The original data with extra columns:

- .hat: Diagonal of the hat matrix
- .sigma: Estimate of residual standard deviation when corresponding observation is dropped from model
- .cooksd: Cooks distance, cooks.distance
- .fitted: Fitted values of model
- .resid: Residuals
- .stdresid: Standardised residuals

Examples

```r
mod <- lm(mpg ~ wt, data = mtcars)
head(fortify(mod))
head(fortify(mod, mtcars))

plot(mod, which = 1)

ggplot(mod, aes(.fitted, .resid)) +  
  geom_point() +  
  geom_hline(yintercept = 0) +  
  geom_smooth(se = FALSE)

ggplot(mod, aes(.fitted, .stdresid)) +  
  geom_point() +  
  geom_hline(yintercept = 0) +  
  geom_smooth(se = FALSE)
```
fortify.map

Fortify method for map objects.

Description

This function turns a map into a data frame that can more easily be plotted with ggplot2.

ggplot(fortify(mod, mtcars), aes(.fitted, .stdresid)) +
  geom_point(aes(colour = factor(cyl)))

ggplot(fortify(mod, mtcars), aes(mpg, .stdresid)) +
  geom_point(aes(colour = factor(cyl)))

plot(mod, which = 2)
ggplot(mod) +
  stat_qq(aes(sample = .stdresid)) +
  geom_abline()

plot(mod, which = 3)
ggplot(mod, aes(.fitted, sqrt(abs(.stdresid)))) +
  geom_point() +
  geom_smooth(se = FALSE)

plot(mod, which = 4)
ggplot(mod, aes(seq_along(.cooks(.cooks)))) +
  geom_bar(stat = "identity")

plot(mod, which = 5)
ggplot(mod, aes(.hat, .stdresid)) +
  geom_vline(size = 2, colour = "white", xintercept = 0) +
  geom_hline(size = 2, colour = "white", yintercept = 0) +
  geom_point() + geom_smooth(se = FALSE)

ggplot(mod, aes(.hat, .stdresid)) +
  geom_point(aes(size = .cooks)) +
  geom_smooth(se = FALSE, size = 0.5)

plot(mod, which = 6)
ggplot(mod, aes(.hat, .cooks)) +
  geom_vline(xintercept = 0, colour = NA) +
  geom_abline(slope = seq(0, 3, by = 0.5), colour = "white") +
  geom_smooth(se = FALSE) +
  geom_point()

ggplot(mod, aes(.hat, .cooks)) +
  geom_point(aes(size = .cooks / .hat)) +
  scale_size_area()
Usage

## S3 method for class 'map'
fortify(model, data, ...)

Arguments

- **model**: map object
- **data**: not used by this method
- **...**: not used by this method

See Also

map_data and borders

Examples

```r
if (require("maps")) {
  ca <- map("county", "ca", plot = FALSE, fill = TRUE)
  head(fortify(ca))
  ggplot(ca, aes(long, lat)) +
      geom_polygon(aes(group = group))

  tx <- map("county", "texas", plot = FALSE, fill = TRUE)
  head(fortify(tx))
  ggplot(tx, aes(long, lat)) +
      geom_polygon(aes(group = group), colour = "white")
}
```

fortify.sp  
Fortify method for classes from the sp package.

Description

To figure out the correct variable name for region, inspect as.data.frame(model).

Usage

- ## S3 method for class 'SpatialPolygonsDataFrame'
  fortify(model, data, region = NULL, ...)

- ## S3 method for class 'SpatialPolygons'
  fortify(model, data, ...)

- ## S3 method for class 'Polygons'
  fortify(model, data, ...)

- ## S3 method for class 'Polygon'
fortify(model, data, ...)

## S3 method for class 'SpatialLinesDataFrame'
fortify(model, data, ...)

## S3 method for class 'Lines'
fortify(model, data, ...)

## S3 method for class 'Line'
fortify(model, data, ...)

Arguments

- **model**: SpatialPolygonsDataFrame to convert into a dataframe.
- **data**: not used by this method
- **region**: name of variable used to split up regions
- **...**: not used by this method

Examples

```r
if (require("maptools")) {
  sids <- system.file("shapes/sids.shp", package="maptools")
  nc1 <- readShapePoly(sids,
    proj4string = CRS("+proj=longlat +datum=NAD27"))
  nc1_df <- fortify(nc1)
}
```

---

**geom_abline**

*Lines: horizontal, vertical, and specified by slope and intercept.*

Description

These paired geoms and stats add straight lines to a plot, either horizontal, vertical or specified by slope and intercept. These are useful for annotating plots.

Usage

```r
geom_abline(mapping = NULL, data = NULL, ..., slope, intercept,
  na.rm = FALSE, show.legend = NA)

geom_hline(mapping = NULL, data = NULL, ..., yintercept, na.rm = FALSE,
  show.legend = NA)

geom_vline(mapping = NULL, data = NULL, ..., xintercept, na.rm = FALSE,
  show.legend = NA)
```
Arguments

mapping Set of aesthetic mappings created by \texttt{aes} or \texttt{aes}. If specified and \texttt{inherit.aes = TRUE} (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn't a mapping defined for the plot.

data A data frame. If specified, overrides the default data frame defined at the top level of the plot.

\ldots other arguments passed on to \texttt{layer}. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like \texttt{color = "red"} or \texttt{size = 3}.
- Other arguments to the layer, for example you override the default \texttt{stat} associated with the layer.
- Other arguments passed on to the \texttt{stat}.

\texttt{na.rm} If \texttt{FALSE} (the default), removes missing values with a warning. If \texttt{TRUE} silently removes missing values.

\texttt{show.legend} logical. Should this layer be included in the legends? \texttt{NA}, the default, includes if any aesthetics are mapped. \texttt{FALSE} never includes, and \texttt{TRUE} always includes.

\texttt{xintercept, yintercept, slope, intercept} Parameters that control the position of the line. If these are set, \texttt{data, mapping} and \texttt{show.legend} are overridden

Details

These geoms act slightly different to other geoms. You can supply the parameters in two ways: either as arguments to the layer function, or via aesthetics. If you use arguments, e.g. \texttt{geom_abline(intercept = 0, slope = QI)}, then behind the scenes the geom makes a new data frame containing just the data you’ve supplied. That means that the lines will be the same in all facets; if you want them to vary across facets, construct the data frame yourself and use aesthetics.

Unlike most other geoms, these geoms do not inherit aesthetics from the plot default, because they do not understand \texttt{x} and \texttt{y} aesthetics which are commonly set in the plot. They also do not affect the \texttt{x} and \texttt{y} scales.

Aesthetics

These geoms are drawn using with \texttt{geom_line} so support the same aesthetics: alpha, colour, linetype and size. They also each have aesthetics that control the position of the line:

- \texttt{geom_vline: xintercept}
- \texttt{geom_hline: yintercept}
- \texttt{geom_abline: slope and intercept}

See Also

See \texttt{geom_segment} for a more general approach to adding straight line segments to a plot.
Examples

```r
p <- ggplot(mtcars, aes(wt, mpg)) + geom_point()

# Fixed values
p + geom_vline(xintercept = 5)
p + geom_vline(xintercept = 1:5)
p + geom_hline(yintercept = 20)

p + geom_abline() # Can't see it - outside the range of the data
p + geom_abline(intercept = 20)

# Calculate slope and intercept of line of best fit
coef(lm(mpg ~ wt, data = mtcars))
p + geom_abline(intercept = 37, slope = -5)
# But this is easier to do with geom_smooth:
p + geom_smooth(method = "lm", se = FALSE)

# To show different lines in different facets, use aesthetics
p <- ggplot(mtcars, aes(mpg, wt)) +
   geom_point() +
   facet_wrap(~ cyl)

mean_wt <- data.frame(cyl = c(4, 6, 8), wt = c(2.28, 3.11, 4.00))
p + geom_hline(aes(yintercept = wt), mean_wt)

# You can also control other aesthetics
ggplot(mtcars, aes(mpg, wt, colour = wt)) +
   geom_point() +
   geom_hline(aes(yintercept = wt, colour = wt), mean_wt) +
   facet_wrap(~ cyl)
```

---

**geom_bar**  
*Bars, rectangles with bases on x-axis*

**Description**

There are two types of bar charts, determined by what is mapped to bar height. By default, `geom_bar` uses `stat="count"` which makes the height of the bar proportional to the number of cases in each group (or if the weight aesthetic is supplied, the sum of the weights). If you want the heights of the bars to represent values in the data, use `stat="identity"` and map a variable to the `y` aesthetic. `stat_count` counts the number of cases at each x position. If you want to bin the data in ranges, you should use `stat_bin` instead.

**Usage**

```r
geom_bar(mapping = NULL, data = NULL, stat = "count",
         position = "stack", width = NULL, binwidth = NULL, ..., na.rm = FALSE,
         show.legend = NA, inherit.aes = TRUE)
```
stat_count(mapping = NULL, data = NULL, geom = "bar",
position = "stack", width = NULL, ..., na.rm = FALSE,
show.legend = NA, inherit.aes = TRUE)

Arguments

mapping Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.
data A data frame. If specified, overrides the default data frame defined at the top level of the plot.
position Position adjustment, either as a string, or the result of a call to a position adjustment function.
width Bar width. By default, set to 90% of the resolution of the data.
binwidth geom_bar no longer has a binwidth argument - if you use it you'll get an warning telling to you use geom_histogram instead.
... other arguments passed on to layer. There are three types of arguments you can use here:
  • Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
  • Other arguments to the layer, for example you override the default stat associated with the layer.
  • Other arguments passed on to the stat.
na.rm If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
show.legend logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
inherit.aes If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.
geom, stat Override the default connection between geom_bar and stat_count.

Details

A bar chart maps the height of the bar to a variable, and so the base of the bar must always be shown to produce a valid visual comparison. Naomi Robbins has a nice article on this topic. This is why it doesn’t make sense to use a log-scaled y axis with a bar chart.

By default, multiple x’s occurring in the same place will be stacked atop one another by position_stack. If you want them to be dodged side-to-side, see position_dodge. Finally, position_fill shows relative proportions at each x by stacking the bars and then stretching or squashing to the same height.
Aesthetics

`geom_bar` understands the following aesthetics (required aesthetics are in bold):

- x
- alpha
- colour
- fill
- linetype
- size

Computed variables

- `count` number of points in bin
- `prop` groupwise proportion

See Also

- `geom_histogram` for continuous data, `position_dodge` for creating side-by-side barcharts.
- `stat_bin`, which bins data in ranges and counts the cases in each range. It differs from `stat_count`, which counts the number of cases at each x position (without binning into ranges). `stat_bin` requires continuous x data, whereas `stat_count` can be used for both discrete and continuous x data.

Examples

```r
# geom_bar is designed to make it easy to create bar charts that show counts (or sums of weights)
g <- ggplot(mpg, aes(class))
# Number of cars in each class:
g + geom_bar()
# Total engine displacement of each class
g + geom_bar(aes(weight = displ))

# To show (e.g.) means, you need stat = "identity"
df <- data.frame(trt = c("a", "b", "c"), outcome = c(2.3, 1.9, 3.2))
ggplot(df, aes(trt, outcome)) +
  geom_bar(stat = "identity")
# But geom_point() display exactly the same information and doesn't require the y-axis to touch zero.
ggplot(df, aes(trt, outcome)) +
  geom_point()

# You can also use geom_bar() with continuous data, in which case it will show counts at unique locations
df <- data.frame(x = rep(c(2.9, 3.1, 4.5), c(5, 10, 4)))
ggplot(df, aes(x)) + geom_bar()
# cf. a histogram of the same data
ggplot(df, aes(x)) + geom_histogram(binwidth = 0.5)
```
# Bar charts are automatically stacked when multiple bars are placed
# at the same location
g + geom_bar(aes(fill = drv))

# You can instead dodge, or fill them
g + geom_bar(aes(fill = drv), position = "dodge")
g + geom_bar(aes(fill = drv), position = "fill")

# To change plot order of bars, change levels in underlying factor
reorder_size <- function(x) {
  factor(x, levels = names(sort(table(x))))
}
ggplot(mpg, aes(reorder_size(class))) + geom_bar()

---

**geom_bin2d**

*Add heatmap of 2d bin counts.*

**Description**

Add heatmap of 2d bin counts.

**Usage**

```r
geom_bin2d(mapping = NULL, data = NULL, stat = "bin2d",
    position = "identity", na.rm = FALSE, show.legend = NA,
    inherit.aes = TRUE, ...)

stat_bin_2d(mapping = NULL, data = NULL, geom = "tile",
    position = "identity", bins = 30, binwidth = NULL, drop = TRUE,
    na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
```

**Arguments**

- **mapping**
  Set of aesthetic mappings created by `aes` or `aes_.` If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.

- **data**
  A data frame. If specified, overrides the default data frame defined at the top level of the plot.

- **position**
  Position adjustment, either as a string, or the result of a call to a position adjustment function.

- **na.rm**
  If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

- **show.legend**
  Logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
geom_bin2d

inherit.aes  If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.

...  other arguments passed on to layer. There are three types of arguments you can use here:

  • Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
  • Other arguments to the layer, for example you override the default stat associated with the layer.
  • Other arguments passed on to the stat.

geom, stat  Use to override the default connection between geom_bin2d and stat_bin2d.

bins numeric vector giving number of bins in both vertical and horizontal directions. Set to 30 by default.

binwidth Numeric vector giving bin width in both vertical and horizontal directions. Overrides bins if both set.

drop if TRUE removes all cells with 0 counts.

Aesthetics

stat_bin2d understands the following aesthetics (required aesthetics are in bold):

  • x
  • y
  • fill

See Also

  stat_binhex for hexagonal binning

Examples

d <- ggplot(diamonds, aes(x, y)) + xlim(4, 10) + ylim(4, 10)
d + geom_bin2d()

# You can control the size of the bins by specifying the number of
# bins in each direction:
d + geom_bin2d(bins = 10)
d + geom_bin2d(bins = 30)

# Or by specifying the width of the bins
d + geom_bin2d(binwidth = c(0.1, 0.1))
Description

The blank geom draws nothing, but can be a useful way of ensuring common scales between different plots.

Usage

```r
geom_blank(mapping = NULL, data = NULL, stat = "identity",
    position = "identity", show.legend = NA, inherit.aes = TRUE, ...)
```

Arguments

- **mapping**: Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.
- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.
- **stat**: The statistical transformation to use on the data for this layer, as a string.
- **position**: Position adjustment, either as a string, or the result of a call to a position adjustment function.
- **show.legend**: logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.
- **inherit.aes**: If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.
- ... other arguments passed on to `layer`. There are three types of arguments you can use here:
  - Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
  - Other arguments to the layer, for example you override the default `stat` associated with the layer.
  - Other arguments passed on to the stat.

Examples

```r
ggplot(mtcars, aes(x = wt, y = mpg)) + geom_blank()
# Nothing to see here!

# Take the following scatter plot
a <- gggplot(mtcars, aes(x = wt, y = mpg), . ~ cyl) + geom_point()
# Add to that some lines with geom_abline()
df <- data.frame(a = rnorm(10, 25), b = rnorm(10, 0))
```
geom_boxplot

Box and whiskers plot.

Description
The lower and upper "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.

Usage
geom_boxplot(mapping = NULL, data = NULL, stat = "boxplot",
              position = "dodge", outlier.colour = NULL, outlier.shape = 19,
              outlier.size = 1.5, outlier.stroke = 0.5, notch = FALSE,
              notchwidth = 0.5, varwidth = FALSE, na.rm = FALSE, show.legend = NA,
              inherit.aes = FALSE, ...)

stat_boxplot(mapping = NULL, data = NULL, geom = "boxplot",
              position = "dodge", coef = 1.5, na.rm = FALSE, show.legend = NA,
              inherit.aes = TRUE, ...)

Arguments

mapping  Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn't a mapping defined for the plot.
data  A data frame. If specified, overrides the default data frame defined at the top level of the plot.
position  Position adjustment, either as a string, or the result of a call to a position adjustment function.
outlier.colour, outlier.shape, outlier.size, outlier.stroke  Default aesthetics for outliers. Set to NULL to inherit from the aesthetics used for the box.
notch  if FALSE (default) make a standard box plot. If TRUE, make a notched box plot. Notches are used to compare groups; if the notches of two boxes do not overlap, this suggests that the medians are significantly different.
notchwidth  for a notched box plot, width of the notch relative to the body (default 0.5)
**geom_boxplot**

- **varwidth**: If FALSE (default) make a standard box plot. If TRUE, boxes are drawn with widths proportional to the square-roots of the number of observations in the groups (possibly weighted, using the weight aesthetic).

- **na.rm**: If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

- **show.legend**: logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

- **inherit.aes**: If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. *Borders*.

- **...**: other arguments passed on to *layer*. There are three types of arguments you can use here:
  - Aesthetics: to set an aesthetic to a fixed value, like *color = "red"* or *size = 3*.
  - Other arguments to the layer, for example you override the default *stat* associated with the layer.
  - Other arguments passed on to the *stat*.

- **geom, stat**: Use to override the default connection between *geom_boxplot* and *stat_boxplot*.

- **coef**: length of the whiskers as multiple of IQR. Defaults to 1.5

**Details**

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).

In a notched box plot, the notches extend 1.58 * IQR / sqrt(n). This gives a roughly 95% confidence interval (See McGill et al. (1978) for more details).

**Aesthetics**

*geom_boxplot* understands the following aesthetics (required aesthetics are in bold):

- **lower**
- **middle**
- **upper**
- **x**
- **ymax**
- **ymin**
- **alpha**
- **colour**
- **fill**
- **linetype**
- **shape**
- **size**
- **weight**
Computed variables

- **width**  width of boxplot
- **ymin**  lower whisker = smallest observation greater than or equal to lower hinge - 1.5 * IQR
- **lower**  lower hinge, 25% quantile
- **notchlower**  lower edge of notch = median - 1.58 * IQR / sqrt(n)
- **middle**  median, 50% quantile
- **notchupper**  upper edge of notch = median + 1.58 * IQR / sqrt(n)
- **upper**  upper hinge, 75% quantile
- **ymax**  upper whisker = largest observation less than or equal to upper hinge + 1.5 * IQR

References


See Also

- **stat_quantile** to view quantiles conditioned on a continuous variable, **geom_jitter** for another way to look at conditional distributions.

Examples

```r
p <- ggplot(mpg, aes(class, hwy))
p + geom_boxplot()
p + geom_boxplot() + geom_jitter(width = 0.2)
p + geom_boxplot() + coord_flip()

p + geom_boxplot(notch = TRUE)
p + geom_boxplot(varwidth = TRUE)
p + geom_boxplot(fill = "white", colour = "#3366FF")
# By default, outlier points match the colour of the box. Use
# outlier.colour to override
p + geom_boxplot(outlier.colour = "red", outlier.shape = 1)

# Boxplots are automatically dodged when any aesthetic is a factor
p + geom_boxplot(aes(colour = drv))

# You can also use boxplots with continuous x, as long as you supply
# a grouping variable. cut_width is particularly useful
ggplot(diamonds, aes(carat, price)) +
geom_boxplot()
ggplot(diamonds, aes(carat, price)) +
geom_boxplot(aes(group = cut_width(carat, 0.25)))

# It's possible to draw a boxplot with your own computations if you
# use stat = "identity";
y <- rnorm(100)
df <- data.frame(
```
geom_contour

Display contours of a 3d surface in 2d.

Description

Display contours of a 3d surface in 2d.

Usage

geom_contour(mapping = NULL, data = NULL, stat = "contour",
position = "identity", lineend = "butt", linejoin = "round",
linemitre = 1, na.rm = FALSE, show.legend = NA, inherit.aes = TRUE,
...
)

stat_contour(mapping = NULL, data = NULL, geom = "contour",
position = "identity", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)

Arguments

mapping Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.
data A data frame. If specified, overrides the default data frame defined at the top level of the plot.
stat The statistical transformation to use on the data for this layer, as a string.
position Position adjustment, either as a string, or the result of a call to a position adjustment function.
lineend Line end style (round, butt, square)
linejoin Line join style (round, mitre, bevel)
linemitre Line mitre limit (number greater than 1)
na.rm  If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

show.legend logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

inherit.aes If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

... other arguments passed on to `layer`. There are three types of arguments you can use here:

- **Aesthetics**: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- **Other arguments to the layer**, for example you override the default `stat` associated with the layer.
- **Other arguments passed on to the stat**.

**geom** The geometric object to use display the data

**Aesthetics**

`geom_contour` understands the following aesthetics (required aesthetics are in bold):

- `x`
- `y`
- `alpha`
- `colour`
- `linetype`
- `size`
- `weight`

**Computed variables**

- `level` height of contour

**See Also**

- `geom_density_2d`: 2d density contours

**Examples**

```r
# Basic plot
v <- ggplot(faithful, aes(waiting, eruptions, z = density))
v + geom_contour()

# Or compute from raw data
ggplot(faithful, aes(waiting, eruptions)) +
    geom_density_2d()
```
geom_count

Count the number of observations at each location.

Description

This is a variant of geom_point that counts the number of observations at each location, then maps the count to point size. It is useful when you have discrete data.

Usage

geom_count(mapping = NULL, data = NULL, stat = "sum",
position = "identity", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)

stat_sum(mapping = NULL, data = NULL, geom = "point",
position = "identity", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)

Arguments

mapping Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.
data A data frame. If specified, overrides the default data frame defined at the top level of the plot.
position Position adjustment, either as a string, or the result of a call to a position adjustment function.
na.rm If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
show.legend logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
**geom_count**

*inherit.aes*  
If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

...  
other arguments passed on to *layer*. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- Other arguments to the layer, for example you override the default *stat* associated with the layer.
- Other arguments passed on to the *stat*.

*geom*, *stat*  
Use to override the default connection between `geom_count` and `stat_sum`.

**Aesthetics**

*geom_point* understands the following aesthetics (required aesthetics are in bold):

- *x*
- *y*
- *alpha*
- *colour*
- *fill*
- *shape*
- *size*
- *stroke*

**Computed variables**

- *n*  number of observations at position
- *prop*  percent of points in that panel at that position

**Examples**

```r
ggplot(mpg, aes(cty, hwy)) + geom_point()

ggplot(mpg, aes(cty, hwy)) + geom_count()

# Best used in conjunction with `scale_size_area` which ensures that counts of zero would be given size 0. Doesn't make much different here because the smallest count is already close to 0.
ggplot(mpg, aes(cty, hwy)) + geom_count() + scale_size_area()

# Display proportions instead of counts ----------------------------------
# By default, all categorical variables in the plot form the groups.
```
# Specifying `geom_count` without a group identifier leads to a plot which is not useful:

d <- ggplot(diamonds, aes(x = cut, y = clarity))
d + geom_count(aes(size = ..prop..))

# To correct this problem and achieve a more desirable plot, we need to specify which group the proportion is to be calculated over.
d + geom_count(aes(size = ..prop.., group = 1)) + scale_size_area(max_size = 10)

# Or group by x/y variables to have rows/columns sum to 1.
d + geom_count(aes(size = ..prop.., group = cut)) + scale_size_area(max_size = 10)
d + geom_count(aes(size = ..prop.., group = clarity)) + scale_size_area(max_size = 10)

---

**geom_crossbar**

**Vertical intervals: lines, crossbars & errorbars.**

**Description**

Various ways of representing a vertical interval defined by x, ymin and ymax.

**Usage**

gem_crossbar(mapping = NULL, data = NULL, stat = "identity", position = "identity", fatten = 2.5, na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
gem_errorbar(mapping = NULL, data = NULL, stat = "identity", position = "identity", na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
gem_linerange(mapping = NULL, data = NULL, stat = "identity", position = "identity", na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
gem_pointrange(mapping = NULL, data = NULL, stat = "identity", position = "identity", ..., fatten = 4, na.rm = FALSE, show.legend = NA, inherit.aes = TRUE)

**Arguments**

- **mapping**
  Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.

- **data**
  A data frame. If specified, overrides the default data frame defined at the top level of the plot.

- **stat**
  The statistical transformation to use on the data for this layer, as a string.
Position adjustment, either as a string, or the result of a call to a position adjustment function.

A multiplicative factor used to increase the size of the middle bar in `geom_crossbar()` and the middle point in `geom_pointrange()`.

If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

Logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

other arguments passed on to `layer`. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- Other arguments to the layer, for example you override the default `stat` associated with the layer.
- Other arguments passed on to the stat.

**Aesthetics**

`geom_linerange` understands the following aesthetics (required aesthetics are in bold):

- `x`
- `ymax`
- `ymin`
- `alpha`
- `colour`
- `linetype`
- `size`

**See Also**

`stat_summary` for examples of these guys in use, `geom_smooth` for continuous analog

**Examples**

```r
# Create a simple example dataset
def <- data.frame(
  trt = factor(c(1, 1, 2, 2)),
  resp = c(1, 5, 3, 4),
  group = factor(c(1, 2, 1, 2)),
  upper = c(1.1, 5.3, 3.3, 4.2),
  lower = c(0.8, 4.6, 2.4, 3.6)
)
```
p <- ggplot(df, aes(trt, resp, colour = group))
p + geom_linerange(aes(ymin = lower, ymax = upper))
p + geom_pointrange(aes(ymin = lower, ymax = upper))
p + geom_crossbar(aes(ymin = lower, ymax = upper), width = 0.2)
p + geom_errorbar(aes(ymin = lower, ymax = upper), width = 0.2)

# Draw lines connecting group means
p +
gem_line(aes(group = group)) +
gem_errorbar(aes(ymin = lower, ymax = upper), width = 0.2)

# If you want to dodge bars and errorbars, you need to manually
# specify the dodge width
p <- ggplot(df, aes(trt, resp, fill = group))
p +
gem_bar(position = "dodge", stat = "identity") +
gem_errorbar(aes(ymin = lower, ymax = upper), position = "dodge", width = 0.25)

# Because the bars and errorbars have different widths
# we need to specify how wide the objects we are dodging are
dodge <- position_dodge(width=0.9)
p +
gem_bar(position = dodge, stat = "identity") +
gem_errorbar(aes(ymin = lower, ymax = upper), position = dodge, width = 0.25)

---

gem_density

Display a smooth density estimate.

Description

A kernel density estimate, useful for display the distribution of variables with underlying smoothness.

Usage

gem_density(mapping = NULL, data = NULL, stat = "density",
position = "identity", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)

stat_density(mapping = NULL, data = NULL, geom = "area",
position = "stack", adjust = 1, kernel = "gaussian", trim = FALSE,
na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)

Arguments

mapping Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE
(the default), is combined with the default mapping at the top level of the plot.
You only need to supply mapping if there isn't a mapping defined for the plot.
**geom_density**

- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.
- **position**: Position adjustment, either as a string, or the result of a call to a position adjustment function.
- **na.rm**: If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
- **show.legend**: logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
- **inherit.aes**: If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

... other arguments passed on to `layer`. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- Other arguments to the layer, for example you override the default `stat` associated with the layer.
- Other arguments passed on to the `stat`.

- **geom**, **stat**: Use to override the default connection between `geom_density` and `stat_density`.
- **adjust**: see `density` for details
- **kernel**: kernel used for density estimation, see `density` for details
- **trim**: This parameter only matters if you are displaying multiple densities in one plot. If FALSE, the default, each density is computed on the full range of the data. If TRUE, each density is computed over the range of that group: this typically means the estimated x values will not line-up, and hence you won’t be able to stack density values.

**Aesthetics**

`geom_density` understands the following aesthetics (required aesthetics are in bold):

- **x**
- **y**
- **alpha**
- **colour**
- **fill**
- **linetype**
- **size**
- **weight**

**Computed variables**

- **density**: density estimate
- **count**: density * number of points - useful for stacked density plots
- **scaled**: density estimate, scaled to maximum of 1
See Also

See `geom_histogram`, `geom_freqpoly` for other methods of displaying continuous distribution. See `geom_violin` for a compact density display.

Examples

```r
ggplot(diamonds, aes(carat)) +
  geom_density()

ggplot(diamonds, aes(carat)) +
  geom_density(adjust = 1/5)

ggplot(diamonds, aes(carat)) +
  geom_density(adjust = 5)

ggplot(diamonds, aes(depth, colour = cut)) +
  geom_density() +
  xlim(55, 70)

ggplot(diamonds, aes(depth, fill = cut, colour = cut)) +
  geom_density(alpha = 0.1) +
  xlim(55, 70)
```

# Stacked density plots: if you want to create a stacked density plot, you # probably want to 'count' (density * n) variable instead of the default # density

```r
# Loses marginal densities
ggplot(diamonds, aes(carat, fill = cut)) +
  geom_density(position = "stack")

# Preserves marginal densities
ggplot(diamonds, aes(carat, ..count.., fill = cut)) +
  geom_density(position = "stack")
```

# You can use position="fill" to produce a conditional density estimate
```r
ggplot(diamonds, aes(carat, ..count.., fill = cut)) +
  geom_density(position = "fill")
```

---

**geom_density_2d**  
*Contours from a 2d density estimate.*

### Description

Perform a 2D kernel density estimation using kde2d and display the results with contours. This can be useful for dealing with overplotting.
**Usage**

```r
gem_density_2d(mapping = NULL, data = NULL, stat = "density2d",
position = "identity", lineend = "butt", linejoin = "round",
linemitre = 1, na.rm = FALSE, show.legend = NA, inherit.aes = TRUE,
...)
```

```r
stat_density_2d(mapping = NULL, data = NULL, geom = "density_2d",
position = "identity", contour = TRUE, n = 100, h = NULL,
na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
```

**Arguments**

- **mapping**
  Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.

- **data**
  A data frame. If specified, overrides the default data frame defined at the top level of the plot.

- **position**
  Position adjustment, either as a string, or the result of a call to a position adjustment function.

- **lineend**
  Line end style (round, butt, square)

- **linejoin**
  Line join style (round, mitre, bevel)

- **linemitre**
  Line mitre limit (number greater than 1)

- **na.rm**
  If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

- **show.legend**
  logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

- **inherit.aes**
  If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

- **...**
  other arguments passed on to `layer`. There are three types of arguments you can use here:
  
  - Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
  
  - Other arguments to the layer, for example you override the default `stat` associated with the layer.

  - Other arguments passed on to the `stat`.

- **geom, stat**
  Use to override the default connection between `geom_density_2d` and `stat_density_2d`.

- **contour**
  If TRUE, contour the results of the 2d density estimation

- **n**
  number of grid points in each direction

- **h**
  Bandwidth (vector of length two). If NULL, estimated using `bandwidth.nrd`. 

geom_density_2d

Aesthetics

geom_density_2d understands the following aesthetics (required aesthetics are in bold):

• x
• y
• alpha
• colour
• linetype
• size

Computed variables

Same as stat_contour

See Also

geom_contour for contour drawing geom, stat_sum for another way of dealing with overplotting

Examples

```r
m <- ggplot(faithful, aes(x = eruptions, y = waiting)) +
geom_point() +
  xlim(0.5, 6) +
  ylim(40, 110)
m + geom_density_2d()

m + stat_density_2d(aes(fill = ..level..), geom = "polygon")

set.seed(4393)
dsmall <- diamonds[sample(nrow(diamonds), 1000), ]
d <- ggplot(dsmall, aes(x, y))
# If you map an aesthetic to a categorical variable, you will get a
# set of contours for each value of that variable
d + geom_density_2d(aes(colour = cut))

# If we turn contouring off, we can use use geoms like tiles:
d + stat_density_2d(geom = "raster", aes(fill = ..density..), contour = FALSE)
# Or points:
d + stat_density_2d(geom = "point", aes(size = ..density..), n = 20, contour = FALSE)
```
Description

In a dot plot, the width of a dot corresponds to the bin width (or maximum width, depending on the binning algorithm), and dots are stacked, with each dot representing one observation.

Usage

geom_dotplot(mapping = NULL, data = NULL, position = "identity",
binwidth = NULL, binaxis = "x", method = "dotdensity",
binpositions = "bygroup", stackdir = "up", stackratio = 1,
dotsize = 1, stackgroups = FALSE, origin = NULL, right = TRUE,
width = 0.9, drop = FALSE, na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)

Arguments

mapping Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.
data A data frame. If specified, overrides the default data frame defined at the top level of the plot.
position Position adjustment, either as a string, or the result of a call to a position adjustment function.
binwidth When method is "dotdensity", this specifies maximum bin width. When method is "histodot", this specifies bin width. Defaults to 1/30 of the range of the data
binaxis The axis to bin along, "x" (default) or "y"
method "dotdensity" (default) for dot-density binning, or "histodot" for fixed bin widths (like stat_bin)
binpositions When method is "dotdensity", "bygroup" (default) determines positions of the bins for each group separately. "all" determines positions of the bins with all the data taken together; this is used for aligning dot stacks across multiple groups.
stackdir which direction to stack the dots. "up" (default), "down", "center", "centerhole" (centered, but with dots aligned)
stackratio how close to stack the dots. Default is 1, where dots just just touch. Use smaller values for closer, overlapping dots.
dotsize The diameter of the dots relative to binwidth, default 1.
stackgroups should dots be stacked across groups? This has the effect that position = "stack" should have, but can’t (because this geom has some odd properties).
origin When method is "histodot", origin of first bin
right When method is "histodot", should intervals be closed on the right (a, b], or not [a, b)
When binaxis is "y", the spacing of the dot stacks for dodging.

If TRUE, remove all bins with zero counts

If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn't inherit behaviour from the default plot specification, e.g. borders.

other arguments passed on to layer. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
- Other arguments to the layer, for example you override the default stat associated with the layer.
- Other arguments passed on to the stat.

Details

With dot-density binning, the bin positions are determined by the data and binwidth, which is the maximum width of each bin. See Wilkinson (1999) for details on the dot-density binning algorithm.

With histodot binning, the bins have fixed positions and fixed widths, much like a histogram.

When binning along the x axis and stacking along the y axis, the numbers on y axis are not meaningful, due to technical limitations of ggplot2. You can hide the y axis, as in one of the examples, or manually scale it to match the number of dots.

Aesthetics

gem_dotplot understands the following aesthetics (required aesthetics are in bold):

- x
- y
- alpha
- colour
- fill

Computed variables

x center of each bin, if binaxis is "x"
y center of each bin, if binaxis is "x"
binwidth max width of each bin if method is "dotdensity"; width of each bin if method is "histodot"
count number of points in bin
ncount count, scaled to maximum of 1
density density of points in bin, scaled to integrate to 1, if method is "histodot"
ndensity density, scaled to maximum of 1, if method is "histodot"
References

Examples

```r
geom_dotplot
```

```r
ggplot(mtcars, aes(x = mpg)) + geom_dotplot()
ggplot(mtcars, aes(x = mpg)) + geom_dotplot(binwidth = 1.5)

# Use fixed-width bins
ggplot(mtcars, aes(x = mpg)) +
  geom_dotplot(method="histodot", binwidth = 1.5)

# Some other stacking methods
ggplot(mtcars, aes(x = mpg)) +
  geom_dotplot(binwidth = 1.5, stackdir = "center")

```r
ggplot(mtcars, aes(x = mpg)) +
  geom_dotplot(binwidth = 1.5, stackdir = "centerwhole")
```

# y axis isn’t really meaningful, so hide it
```r
ggplot(mtcars, aes(x = mpg)) + geom_dotplot(binwidth = 1.5) +
  scale_y_continuous(NULL, breaks = NULL)
```

# Overlap dots vertically
```r
ggplot(mtcars, aes(x = mpg)) + geom_dotplot(binwidth = 1.5, stackratio = .7)

# Expand dot diameter
```r
ggplot(mtcars, aes(x = mpg)) + geom_dotplot(binwidth = 1.5, dotsize = 1.25)
```

# Examples with stacking along y axis instead of x
```r
ggplot(mtcars, aes(x = 1, y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center")
```

```r
ggplot(mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center")
```

```r
ggplot(mtcars, aes(x = factor(cyl), y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "centerwhole")
```

```r
ggplot(mtcars, aes(x = factor(vs), fill = factor(cyl), y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center", position = "dodge")
```

# binpositions="all" ensures that the bins are aligned between groups
```r
ggplot(mtcars, aes(x = factor(am), y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center", binpositions = "all")
```

# Stacking multiple groups, with different fill
```r
ggplot(mtcars, aes(x = mpg, fill = factor(cyl))) +
  geom_dotplot(stackgroups = TRUE, binwidth = 1, binpositions = "all")
ggplot(mtcars, aes(x = mpg, fill = factor(cyl))) +
  geom_dotplot(stackgroups = TRUE, binwidth = 1, method = "histodot")
```
geom_errorbarh

```
geom_errorbarh(mtcars, aes(x = 1, y = mpg, fill = factor(cyl))) +
geom_dotplot(binaxis = "y", stackgroups = TRUE, binwidth = 1, method = "histodot")
```

description

Horizontal error bars

Usage

```
geom_errorbarh(mapping = NULL, data = NULL, stat = "identity",
position = "identity", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)
```

Arguments

- **mapping**: Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.
- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.
- **stat**: The statistical transformation to use on the data for this layer, as a string.
- **position**: Position adjustment, either as a string, or the result of a call to a position adjustment function.
- **na.rm**: If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
- **show.legend**: logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.
- **inherit.aes**: If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.
- **...**: other arguments passed on to layer. There are three types of arguments you can use here:
  - Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
  - Other arguments to the layer, for example you override the default stat associated with the layer.
  - Other arguments passed on to the stat.
geom_freqpoly

**Aesthetics**

geom_errorbarh understands the following aesthetics (required aesthetics are in bold):

- x
- xmax
- xmin
- y
- alpha
- colour
- height
- linetype
- size

**See Also**

geom_errorbar: vertical error bars

**Examples**

```r
df <- data.frame(
    trt = factor(c(1, 1, 2, 2)),
    resp = c(1, 5, 3, 4),
    group = factor(c(1, 2, 1, 2)),
    se = c(0.1, 0.3, 0.3, 0.2)
)

# Define the top and bottom of the errorbars
p <- ggplot(df, aes(resp, trt, colour = group))
p + geom_point() +
  geom_errorbarh(aes(xmax = resp + se, xmin = resp - se))
p + geom_point() +
  geom_errorbarh(aes(xmax = resp + se, xmin = resp - se, height = .2))
```

---

**geom_freqpoly**

*Histograms and frequency polygons.*

**Description**

Display a 1d distribution by dividing into bins and counting the number of observations in each bin. Histograms use bars; frequency polygons use lines.

*stat_bin* is suitable only for continuous x data. If your x data is discrete, you probably want to use *stat_count*. 
Usage

g geom_freqpoly(mapping = NULL, data = NULL, stat = "bin",
          position = "identity", na.rm = FALSE, show.legend = NA,
          inherit.aes = TRUE, ...)

g geom_histogram(mapping = NULL, data = NULL, stat = "bin",
          binwidth = NULL, bins = NULL, origin = NULL, right = FALSE,
          position = "stack", na.rm = FALSE, show.legend = NA,
          inherit.aes = TRUE, ...)

stat_bin(mapping = NULL, data = NULL, geom = "bar", position = "stack",
          width = 0.9, drop = FALSE, right = FALSE, binwidth = NULL,
          bins = NULL, origin = NULL, breaks = NULL, na.rm = FALSE,
          show.legend = NA, inherit.aes = TRUE, ...)

Arguments

mapping Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.

data A data frame. If specified, overrides the default data frame defined at the top level of the plot.

position Position adjustment, either as a string, or the result of a call to a position adjustment function.

na.rm If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

show.legend logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

inherit.aes If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.

... other arguments passed on to layer. There are three types of arguments you can use here:

• Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
• Other arguments to the layer, for example you override the default stat associated with the layer.
• Other arguments passed on to the stat.

binwidth Bin width to use. Defaults to 1/bins of the range of the data

bins Number of bins. Overridden by binwidth or breaks. Defaults to 30

origin Origin of first bin

right If TRUE, right-closed, left-open, if FALSE, the default, right-open, left-closed.

geom, stat Use to override the default connection between geom_histogram/geom_freqpoly and stat_bin.
**geom_freqpoly**

- **width** Width of bars when used with categorical data
- **drop** If TRUE, remove all bins with zero counts
- **breaks** Actual breaks to use. Overrides bin width, bin number and origin

**Details**

By default, stat_bin uses 30 bins - this is not a good default, but the idea is to get you experimenting with different binwidths. You may need to look at a few to uncover the full story behind your data.

**Aesthetics**

geom_histogram uses the same aesthetics as geom_bar; geom_freqpoly uses the same aesthetics as geom_line.

**Computed variables**

- **count** number of points in bin
- **density** density of points in bin, scaled to integrate to 1
- **ncount** count, scaled to maximum of 1
- **ndensity** density, scaled to maximum of 1

**See Also**

stat_count, which counts the number of cases at each x position, without binning. It is suitable for both discrete and continuous x data, whereas stat_bin is suitable only for continuous x data.

**Examples**

```r
ggplot(diamonds, aes(carat)) +
  geom_histogram()

# Rather than stacking histograms, it's easier to compare frequency
# polygons
ggplot(diamonds, aes(price, fill = cut)) +
  geom_histogram(binwidth = 500)

# To make it easier to compare distributions with very different counts,
# put density on the y axis instead of the default count
if (require("ggplot2movies")) {
  ggplot(diamonds, aes(price, ..density.., colour = cut)) +
  geom_freqpoly(binwidth = 500)
}
```

```r
# Often we don't want the height of the bar to represent the
```
# count of observations, but the sum of some other variable.  
# For example, the following plot shows the number of movies  
# in each rating.
m <- ggplot(movies, aes(rating))
m + geom_histogram(binwidth = 0.1)

# If, however, we want to see the number of votes cast in each  
# category, we need to weight by the votes variable
m + geom_histogram(aes(weight = votes), binwidth = 0.1) + ylab("votes")

# For transformed scales, binwidth applies to the transformed data.  
# The bins have constant width on the transformed scale.
m + geom_histogram() + scale_x_log10()
m + geom_histogram(binwidth = 0.05) + scale_x_log10()

# For transformed coordinate systems, the binwidth applies to the  
# raw data. The bins have constant width on the original scale.

# Using log scales does not work here, because the first  
# bar is anchored at zero, and so when transformed becomes negative  
# infinity. This is not a problem when transforming the scales, because  
# no observations have 0 ratings.
m + geom_histogram(origin = 0) + coord_trans(x = "log10")  
# Use origin = 0, to make sure we don't take sqrt of negative values
m + geom_histogram(origin = 0) + coord_trans(x = "sqrt")

# You can also transform the y axis. Remember that the base of the bars  
# has value 0, so log transformations are not appropriate
m <- ggplot(movies, aes(x = rating))
m + geom_histogram(binwidth = 0.5) + scale_y_sqrt()
}
rm(movies)

---

**geom_hex**  
Hexagon binning.

**Description**
Hexagon binning.

**Usage**

```
geom_hex(mapping = NULL, data = NULL, stat = "binhex",  
  position = "identity", na.rm = FALSE, show.legend = NA,  
  inherit.aes = TRUE, .)
```

```
stat_bin_hex(mapping = NULL, data = NULL, geom = "hex",  
  position = "identity", bins = 30, binwidth = NULL, na.rm = FALSE,  
  show.legend = NA, inherit.aes = TRUE, .)
```
Arguments

mapping  Set of aesthetic mappings created by `aes` or `aes_*`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.

data  A data frame. If specified, overrides the default data frame defined at the top level of the plot.

position  Position adjustment, either as a string, or the result of a call to a position adjustment function.

na.rm  If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

show.legend  logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

inherit.aes  If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

...  other arguments passed on to `layer`. There are three types of arguments you can use here:

  • Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
  • Other arguments to the layer, for example you override the default `stat` associated with the layer.
  • Other arguments passed on to the `stat`.

geom, stat  Override the default connection between `geom_hex` and `stat_binhex`.

bins  numeric vector giving number of bins in both vertical and horizontal directions. Set to 30 by default.

binwidth  Numeric vector giving bin width in both vertical and horizontal directions. Overrides bins if both set.

Aesthetics

`geom_hex` understands the following aesthetics (required aesthetics are in bold):

  • x
  • y
  • alpha
  • colour
  • fill
  • size

See Also

`stat_bin2d` for rectangular binning
Examples

```r
d <- ggplot(diamonds, aes(carat, price))
d + geom_hex()
```

# You can control the size of the bins by specifying the number of
# bins in each direction:
d + geom_hex(bins = 10)
d + geom_hex(bins = 30)

# Or by specifying the width of the bins
d + geom_hex(binwidth = c(1, 1000))
d + geom_hex(binwidth = c(.1, 500))

---

**geom_jitter**

Points, jittered to reduce overplotting.

Description

The jitter geom is a convenient default for geom_point with position = 'jitter'. It’s a useful way of handling overplotting caused by discreteness in smaller datasets.

Usage

```r
geom_jitter(mapping = NULL, data = NULL, width = NULL, height = NULL,
            stat = "identity", position = "jitter", na.rm = FALSE,
            show.legend = NA, inherit.aes = TRUE, ...)
```

Arguments

- **mapping**: Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.
- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.
- **width**: Amount of vertical and horizontal jitter. The jitter is added in both positive and negative directions, so the total spread is twice the value specified here. If omitted, defaults to 40% of the resolution of the data: this means the jitter values will occupy 80% of the implied bins. Categorical data is aligned on the integers, so a width or height of 0.5 will spread the data so it’s not possible to see the distinction between the categories.
- **height**: Amount of vertical and horizontal jitter. The jitter is added in both positive and negative directions, so the total spread is twice the value specified here. If omitted, defaults to 40% of the resolution of the data: this means the jitter values will occupy 80% of the implied bins. Categorical data is aligned on the integers, so a width or height of 0.5 will spread the data so it’s not possible to see the distinction between the categories.
The statistical transformation to use on the data for this layer, as a string.

Position adjustment, either as a string, or the result of a call to a position adjustment function.

If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

Logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.

Other arguments passed on to layer. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
- Other arguments to the layer, for example you override the default stat associated with the layer.
- Other arguments passed on to the stat.

Aesthetics

geom_point understands the following aesthetics (required aesthetics are in bold):

- x
- y
- alpha
- colour
- fill
- shape
- size
- stroke

See Also

geom_point for regular, unjittered points, geom_boxplot for another way of looking at the conditional distribution of a variable

Examples

```r
p <- ggplot(mpg, aes(cyl, hwy))
p + geom_point()
p + geom_jitter()

# Add aesthetic mappings
p + geom_jitter(aes(colour = class))

# Use smaller width/height to emphasise categories
```
`ggplot(mpg, aes(cyl, hwy)) + geom_jitter()`
`ggplot(mpg, aes(cyl, hwy)) + geom_jitter(width = 0.25)`

# Use larger width/height to completely smooth away discreteness
`ggplot(mpg, aes(cty, hwy)) + geom_jitter()`
`ggplot(mpg, aes(cty, hwy)) + geom_jitter(width = 0.5, height = 0.5)`

---

### geom_label

**Textual annotations.**

**Description**

`geom_text` adds text directly to the plot. `geom_label` draws a rectangle underneath the text, making it easier to read.

**Usage**

```
geom_label(mapping = NULL, data = NULL, stat = "identity",
  position = "identity", parse = FALSE, ..., nudge_x = 0, nudge_y = 0,
  label.padding = unit(0.25, "lines"), label.r = unit(0.15, "lines"),
  label.size = 0.25, na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE)
```

```
geom_text(mapping = NULL, data = NULL, stat = "identity",
  position = "identity", parse = FALSE, ..., nudge_x = 0, nudge_y = 0,
  check_overlap = FALSE, na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE)
```

**Arguments**

- **mapping**: Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.
- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.
- **stat**: The statistical transformation to use on the data for this layer, as a string.
- **position**: Position adjustment, either as a string, or the result of a call to a position adjustment function.
- **parse**: If TRUE, the labels will be parsed into expressions and displayed as described in ?plotmath
- **...**: other arguments passed on to `layer`. There are three types of arguments you can use here:
  - Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
  - Other arguments to the layer, for example you override the default stat associated with the layer.
• Other arguments passed on to the stat.

nudge_x, nudge_y
Horizontal and vertical adjustment to nudge labels by. Useful for offsetting text from points, particularly on discrete scales.

label.padding
Amount of padding around label. Defaults to 0.25 lines.

label.r
Radius of rounded corners. Defaults to 0.15 lines.

label.size
Size of label border, in mm.

na.rm
If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

show.legend
logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

inherit.aes
If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.

check_overlap
If TRUE, text that overlaps previous text in the same layer will not be plotted. A quick and dirty way

Details
Note the the "width" and "height" of a text element are 0, so stacking and dodging text will not work by default, and axis limits are not automatically expanded to include all text. Obviously, labels do have height and width, but they are physical units, not data units. The amount of space they occupy on that plot is not constant in data units: when you resize a plot, labels stay the same size, but the size of the axes changes.

Aesthetics

geom_text understands the following aesthetics (required aesthetics are in bold):

• label
• x
• y
• alpha
• angle
• colour
• family
• fontface
• hjust
• lineheight
• size
• vjust
Currently `geom_label` does not support the `rot` parameter and is considerably slower than `geom_text`. The `fill` aesthetic controls the background colour of the label.

**Alignment**

You can modify text alignment with the `vjust` and `hjust` aesthetics. These can either be a number between 0 (right/bottom) and 1 (top/left) or a character ("left", "middle", "right", "bottom", "center", "top"). There are two special alignments: "inward" and "outward". Inward always aligns text towards the center, and outward aligns it away from the center.

**Examples**

```r
p <- ggplot(mtcars, aes(wt, mpg, label = rownames(mtcars)))

p + geom_text()
# Avoid overlaps
p + geom_text(check_overlap = TRUE)
# Labels with background
p + geom_label()
# Change size of the label
p + geom_text(size = 10)

# Set aesthetics to fixed value
p + geom_point() + geom_text(hjust = 0, nudge_x = 0.05)
p + geom_point() + geom_text(vjust = 0, nudge_y = 0.5)
p + geom_point() + geom_text(angle = 45)
## Not run:
p + geom_text(family = "Times New Roman")

## End(Not run)

# Add aesthetic mappings
p + geom_text(aes(colour = factor(cyl)))
p + geom_text(aes(colour = factor(cyl))) +
  scale_colour_discrete(l = 40)
p + geom_label(aes(fill = factor(cyl)), colour = "white", fontface = "bold")

p + geom_text(aes(size = wt))
# Scale height of text, rather than sqrt(height)
p + geom_text(aes(size = wt)) + scale_radius(range = c(3,6))

# You can display expressions by setting parse = TRUE. The # details of the display are described in ?plotmath, but note that # geom_text uses strings, not expressions.
p + geom_text(aes(label = paste(wt, "\(\)", cyl, ",", sep = ",")),
  parse = TRUE)

# Add a text annotation
p +
  geom_text() +
  annotate("text", label = "plot mpg vs. wt", x = 2, y = 15, size = 8, colour = "red")
```
# Aligning labels and bars

```r
df <- data.frame(
    x = factor(c(1, 1, 2, 2)),
    y = c(1, 3, 2, 1),
    grp = c("a", "b", "a", "b")
)
```

```r
# ggplot2 doesn't know you want to give the labels the same virtual width as the bars:
# So tell it:
```r
ggplot(data = df, aes(x, y, fill = grp, label = y)) +
  geom_bar(stat = "identity", position = "dodge") +
  geom_text(position = "dodge")
```

```r
# Use you can't nudge and dodge text, so instead adjust the y position
```r
ggplot(data = df, aes(x, y, fill = grp, label = y)) +
  geom_bar(stat = "identity", position = "dodge") +
  geom_text(aes(y = y + 0.05), position = position_dodge(0.9), vjust = 0)
```

```r
# To place text in the middle of each bar in a stacked barplot, you need to do the computation yourself
```r
df <- transform(df, mid_y = ave(df$y, df$x, FUN = function(val) cumsum(val) - (0.5 * val)))
```

```r
ggplot(data = df, aes(x, y, fill = grp, label = y)) +
  geom_bar(stat = "identity") +
  geom_text(aes(y = mid_y))
```

```r
# Justification
```
```r
df <- data.frame(
    x = c(1, 1, 2, 2, 1.5),
    y = c(1, 2, 1, 2, 1.5),
    text = c("bottom-left", "bottom-right", "top-left", "top-right", "center")
)
```

```r
ggplot(df, aes(x, y)) +
  geom_text(aes(label = text))
```

```r
ggplot(df, aes(x, y)) +
  geom_text(aes(label = text), vjust = "inward", hjust = "inward")
```

---

**geom_map**

*Polygons from a reference map.*

**Description**

Does not affect position scales.
### Usage

```r
gem_map(mapping = NULL, data = NULL, map, stat = "identity", 
na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
```

### Arguments

- **mapping**
  - Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.

- **data**
  - A data frame. If specified, overrides the default data frame defined at the top level of the plot.

- **map**
  - Data frame that contains the map coordinates. This will typically be created using `fortify` on a spatial object. It must contain columns `x` or `long`, `y` or `lat`, and `region` or `id`.

- **stat**
  - The statistical transformation to use on the data for this layer, as a string.

- **na.rm**
  - If `FALSE` (the default), removes missing values with a warning. If `TRUE` silently removes missing values.

- **show.legend**
  - Logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.

- **inherit.aes**
  - If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

- **...**
  - Other arguments passed on to `layer`. There are three types of arguments you can use here:
    - Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
    - Other arguments to the layer, for example you override the default `stat` associated with the layer.
    - Other arguments passed on to the stat.

### Aesthetics

`geom_map` understands the following aesthetics (required aesthetics are in bold):

- **map_id**
- **alpha**
- **colour**
- **fill**
- **linetype**
- **size**
Examples

# When using geom_polygon, you will typically need two data frames:
# one contains the coordinates of each polygon (positions), and the
# other the values associated with each polygon (values). An id
# variable links the two together

ids <- factor(c("1.1", "2.1", "1.2", "2.2", "1.3", "2.3"))

values <- data.frame(
  id = ids,
  value = c(3, 3.1, 3.1, 3.2, 3.15, 3.5)
)

positions <- data.frame(
  id = rep(ids, each = 4),
  x = c(2, 1, 1.1, 2.2, 1, 0, 0.3, 1.1, 2.2, 1.1, 1.2, 2.5, 1.1, 0.3,
        0.5, 1.2, 2.5, 1.2, 1.3, 2.7, 1.2, 0.5, 0.6, 1.3),
  y = c(-0.5, 0, 1, 0.5, 0, 0.5, 1.5, 1, 0.5, 1, 2.1, 1.7, 1, 1.5,
        2.2, 2.1, 1.7, 2.1, 3.2, 2.8, 2.1, 2.2, 3.3, 3.2))
)

ggplot(values) + geom_map(aes(map_id = id), map = positions) +
  expand_limits(positions)

if (require(maps)) {
  crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)
  crimesm <- reshape2::melt(crimes, id = 1)
  last_plot()
  ggplot(crimesm, aes(map_id = state)) +
    geom_map(aes(fill = value), map = states_map) +
    expand_limits(x = states_map$long, y = states_map$lat) +
    facet_wrap(~ variable)
}
Description

gem.path() connects the observations in the order in which they appear in the data. geom.line() connects them in order of the variable on the x axis. geom.step() creates a stairstep plot, highlighting exactly when changes occur.

Usage

geom_path(mapping = NULL, data = NULL, stat = "identity", position = "identity", lineend = "butt", linejoin = "round", linemitre = 1, na.rm = FALSE, arrow = NULL, show.legend = NA, inherit.aes = TRUE, ...)

geom_line(mapping = NULL, data = NULL, stat = "identity", position = "identity", na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)

geom_step(mapping = NULL, data = NULL, stat = "identity", position = "identity", direction = "hv", na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)

Arguments

mapping Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.

data A data frame. If specified, overrides the default data frame defined at the top level of the plot.

stat The statistical transformation to use on the data for this layer, as a string.

position Position adjustment, either as a string, or the result of a call to a position adjustment function.

lineend Line end style (round, butt, square)

linejoin Line join style (round, mitre, bevel)

linemitre Line mitre limit (number greater than 1)

na.rm If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

arrow Arrow specification, as created by arrow

show.legend logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

inherit.aes If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.

... other arguments passed on to layer. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
• Other arguments to the layer, for example you override the default stat associated with the layer.
• Other arguments passed on to the stat.

direction  direction of stairs: ‘vh’ for vertical then horizontal, or ‘hv’ for horizontal then vertical

Aesthetics

geom_path understands the following aesthetics (required aesthetics are in bold):

• x
• y
• alpha
• colour
• linetype
• size

See Also

geom_polygon: Filled paths (polygons); geom_segment: Line segments

Examples

# geom_line() is suitable for time series
ggplot(economics, aes(date, unemploy)) + geom_line()

# geom_step() is useful when you want to highlight exactly when
# the y value changes
recent <- economics[economics$date > as.Date("2013-01-01"), ]
ggplot(recent, aes(date, unemploy)) + geom_line()

# geom_path lets you explore how two variables are related over time,
# e.g. unemployment and personal savings rate
m <- ggplot(economics, aes(unemploy/pop, psavert))
m + geom_path()
m + geom_path(aes(colour = as.numeric(date)))

# Changing parameters  

# Use the arrow parameter to add an arrow to the line
# See ?arrow for more details
c <- ggplot(economics, aes(x = date, y = pop))
c + geom_line(arrow = arrow())
c + geom_line(
    arrow = arrow(angle = 15, ends = "both", type = "closed"))
Description

The point geom is used to create scatterplots.

Usage

gem_point(mapping = NULL, data = NULL, stat = "identity",

# Control line join parameters
df <- data.frame(x = 1:3, y = c(4, 1, 9))
base <- ggplot(df, aes(x, y))
base + geom_path(size = 10)
base + geom_path(size = 10, lineend = "round")
base + geom_path(size = 10, linejoin = "mitre", lineend = "butt")

# NAs break the line. Use na.rm = T to suppress the warning message
df <- data.frame(
  x = 1:5,
  y1 = c(1, 2, 3, 4, NA),
  y2 = c(NA, 2, 3, 4, 5),
  y3 = c(1, 2, NA, 4, 5)
)

ggplot(df, aes(x, y1)) + geom_point() + geom_line()

# Setting line type vs colour/size
# Line type needs to be applied to a line as a whole, so it can
# not be used with colour or size that vary across a line
x <- seq(0.01, .99, length.out = 100)
df <- data.frame(  
  x = rep(x, 2),
  y = c(qlogis(x), 2 * qlogis(x)),
  group = rep(c("a","b"),
              each = 100)
)

p <- ggplot(df, aes(x=x, y=y, group=group))
# These work
p + geom_line(linetype = 2)
p + geom_line(aes(colour = group), linetype = 2)
p + geom_line(aes(colour = x))
# But this doesn't
should_stop(p + geom_line(aes(colour = x), linetype=2))
geom_point

position = "identity", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)

Arguments

mapping Set of aesthetic mappings created by aes or aes_. If specified and inherit.aes = TRUE (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.
data A data frame. If specified, overrides the default data frame defined at the top level of the plot.
stat The statistical transformation to use on the data for this layer, as a string.
position Position adjustment, either as a string, or the result of a call to a position adjustment function.
na.rm If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
show.legend logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
inherit.aes If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.
...
other arguments passed on to layer. There are three types of arguments you can use here:

• Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
• Other arguments to the layer, for example you override the default stat associated with the layer.
• Other arguments passed on to the stat.

Details

The scatterplot is useful for displaying the relationship between two continuous variables, although it can also be used with one continuous and one categorical variable, or two categorical variables. See geom_jitter for possibilities.

The bubblechart is a scatterplot with a third variable mapped to the size of points. There are no special names for scatterplots where another variable is mapped to point shape or colour, however.

The biggest potential problem with a scatterplot is overplotting: whenever you have more than a few points, points may be plotted on top of one another. This can severely distort the visual appearance of the plot. There is no one solution to this problem, but there are some techniques that can help. You can add additional information with geom_smooth, geom_quantile or geom_density_2d. If you have few unique x values, geom_boxplot may also be useful. Alternatively, you can summarise the number of points at each location and display that in some way, using stat_sum. Another technique is to use transparent points, e.g. geom_point(alpha = 0.05).
Aesthetics

geom_point understands the following aesthetics (required aesthetics are in bold):

- x
- y
- alpha
- colour
- fill
- shape
- size
- stroke

See Also

scale_size to see scale area of points, instead of radius, geom_jitter to jitter points to reduce (mild) overplotting

Examples

```r
p <- ggplot(mtcars, aes(wt, mpg))
p + geom_point()

# Add aesthetic mappings
p + geom_point(aes(colour = factor(cyl)))
p + geom_point(aes(shape = factor(cyl)))
p + geom_point(aes(size = qsec))

# Change scales
p + geom_point(aes(colour = cyl)) + scale_colour_gradient(low = "blue")
p + geom_point(aes(shape = factor(cyl))) + scale_shape(solid = FALSE)

# Set aesthetics to fixed value
ggplot(mtcars, aes(wt, mpg)) + geom_point(colour = "red", size = 3)

# Varying alpha is useful for large datasets
d <- ggplot(diamonds, aes(carat, price))
d + geom_point(alpha = 1/10)
d + geom_point(alpha = 1/20)
d + geom_point(alpha = 1/100)

# For shapes that have a border (like 21), you can colour the inside and outside separately. Use the stroke aesthetic to modify the width of the border
ggplot(mtcars, aes(wt, mpg)) +
  geom_point(shape = 21, colour = "black", fill = "white", size = 5, stroke = 5)
```
# You can create interesting shapes by layering multiple points of different sizes
p <- ggplot(mtcars, aes(mpg, wt, shape = factor(cyl)))
p + geom_point(aes(colour = factor(cyl)), size = 4) +
  geom_point(colour = "grey90", size = 1.5)
p + geom_point(colour = "black", size = 4.5) +
  geom_point(colour = "pink", size = 4) +
  geom_point(aes(shape = factor(cyl))))

# These extra layers don't usually appear in the legend, but we can force their inclusion
p + geom_point(colour = "black", size = 4.5, show.legend = TRUE) +
  geom_point(colour = "pink", size = 4, show.legend = TRUE) +
  geom_point(aes(shape = factor(cyl))))

# geom_point warns when missing values have been dropped from the data set and not plotted, you can turn this off by setting na.rm = TRUE
mtcars2 <- transform(mtcars, mpg = ifelse(runif(32) < 0.2, NA, mpg))
ggplot(mtcars2, aes(wt, mpg)) + geom_point()
ggplot(mtcars2, aes(wt, mpg)) + geom_point(na.rm = TRUE)

---

**geom_polygon**

*Polygon, a filled path.*

**Description**

Polygon, a filled path.

**Usage**

```r
gem_polygon(mapping = NULL, data = NULL, stat = "identity",
position = "identity", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)```

**Arguments**

- `mapping` Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.
- `data` A data frame. If specified, overrides the default data frame defined at the top level of the plot.
- `stat` The statistical transformation to use on the data for this layer, as a string.
- `position` Position adjustment, either as a string, or the result of a call to a position adjustment function.
- `na.rm` If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
Logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

Other arguments passed on to `layer`. There are three types of arguments you can use here:

- **Aesthetics**: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- **Other arguments to the layer**, for example you override the default `stat` associated with the layer.
- **Other arguments passed on to the stat**.

### Aesthetics

`geom_polygon` understands the following aesthetics (required aesthetics are in bold):

- `x`
- `y`
- `alpha`
- `colour`
- `fill`
- `linetype`
- `size`

### See Also

- `geom_path` for an unfilled polygon, `geom_ribbon` for a polygon anchored on the x-axis

### Examples

```r
# When using `geom_polygon`, you will typically need two data frames:
# one contains the coordinates of each polygon (positions), and the
# other the values associated with each polygon (values). An id
# variable links the two together

tmp <- data.frame(id = c("A", "B", "C", "D", "E", "F"),
                  value = c(1, 2, 3, 4, 5, 6),
                  positions = c(1, 2, 3, 4, 5, 6),
                  values = c(10, 20, 30, 40, 50, 60))

# Load the ggplot2 package
ggplot(tmp, aes(x = x, y = y)) +
  geom_polygon(aes(alpha = a, colour = c), show.legend = FALSE) +
  geom_point()
```

```r
# Use data frames to create a set of polygons:

tmp <- data.frame(id = c("A", "B", "C", "D", "E", "F"),
                  positions = c(1, 2, 3, 4, 5, 6),
                  values = c(10, 20, 30, 40, 50, 60))

# Create a ggplot object
p <- ggplot(tmp, aes(x = x, y = y))

# Add a layer for the polygons
p + geom_polygon(aes(alpha = a, colour = c), show.legend = FALSE)
```

```r
# Use data frames to create a set of polygons:

tmp <- data.frame(id = c("A", "B", "C", "D", "E", "F"),
                  positions = c(1, 2, 3, 4, 5, 6),
                  values = c(10, 20, 30, 40, 50, 60))

# Create a ggplot object
p <- ggplot(tmp, aes(x = x, y = y))

# Add a layer for the polygons
p + geom_polygon(aes(alpha = a, colour = c), show.legend = FALSE)
```

```r
# Use data frames to create a set of polygons:

tmp <- data.frame(id = c("A", "B", "C", "D", "E", "F"),
                  positions = c(1, 2, 3, 4, 5, 6),
                  values = c(10, 20, 30, 40, 50, 60))

# Create a ggplot object
p <- ggplot(tmp, aes(x = x, y = y))

# Add a layer for the polygons
p + geom_polygon(aes(alpha = a, colour = c), show.legend = FALSE)
```
Currently we need to manually merge the two together

datapoly <- merge(values, positions, by=c("id"))

(p <- ggplot(datapoly, aes(x=x, y=y)) + geom_polygon(aes(fill=value, group=id)))

# Which seems like a lot of work, but then it's easy to add on
# other features in this coordinate system, e.g.:

stream <- data.frame(
  x = cumsum(runif(50, max = 0.1)),
  y = cumsum(runif(50,max = 0.1))
)

p + geom_line(data = stream, colour="grey30", size = 5)

# And if the positions are in longitude and latitude, you can use
# coord_map to produce different map projections.

---

**geom_quantile**  
Add quantile lines from a quantile regression.

---

**Description**

This can be used as a continuous analogue of a geom_boxplot.

**Usage**

```r
geom_quantile(mapping = NULL, data = NULL, stat = "quantile",
  position = "identity", lineend = "butt", linejoin = "round",
  linemitre = 1, na.rm = FALSE, show.legend = NA, inherit.aes = TRUE,
  ...)
```

```r
stat_quantile(mapping = NULL, data = NULL, geom = "quantile",
  position = "identity", quantiles = c(0.25, 0.5, 0.75), formula = NULL,
  method = "rq", method.args = list(), na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE, ...)
```

**Arguments**

- **mapping**  
  Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn't a mapping defined for the plot.

- **data**  
  A data frame. If specified, overrides the default data frame defined at the top level of the plot.

- **position**  
  Position adjustment, either as a string, or the result of a call to a position adjustment function.
geom_quantile

lineend Line end style (round, butt, square)
linejoin Line join style (round, mitre, bevel)
linemitre Line mitre limit (number greater than 1)
na.rm If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
show.legend logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
inherit.aes If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.

... other arguments passed on to layer. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
- Other arguments to the layer, for example you override the default stat associated with the layer.
- Other arguments passed on to the stat.

geom, stat Use to override the default connection between geom_quantile and stat_quantile.
quantiles conditional quantiles of y to calculate and display
formula formula relating y variables to x variables
method Quantile regression method to use. Currently only supports rq.
method.args List of additional arguments passed on to the modelling function defined by method.

Aesthetics

geom_quantile understands the following aesthetics (required aesthetics are in bold):

- x
- y
- alpha
- colour
- linetype
- size
- weight

Computed variables

quantile quantile of distribution
Examples

```r
m <- ggplot(mpg, aes(displ, 1 / hwy)) + geom_point()
  m + geom_quantile()
  m + geom_quantile(quantiles = 0.5)
  q10 <- seq(0.05, 0.95, by = 0.05)
  m + geom_quantile(quantiles = q10)

# You can also use rqss to fit smooth quantiles
m + geom_quantile(method = "rqss")
# Note that rqss doesn't pick a smoothing constant automatically, so
# you'll need to tweak lambda yourself
m + geom_quantile(method = "rqss", lambda = 0.1)

# Set aesthetics to fixed value
m + geom_quantile(colour = "red", size = 2, alpha = 0.5)
```

draw rectangles.

Description

`geom_rect` and `geom_tile` do the same thing, but are parameterised differently. `geom_rect` uses the locations of the four corners (`xmin`, `xmax`, `ymin` and `ymax`). `geom_tile` uses the center of the tile and its size (`x`, `y`, `width`, and `height`). `geom_raster` is a high performance special case for when all the tiles are the same size.

Usage

```r
geom_raster(mapping = NULL, data = NULL, stat = "identity",
            position = "identity", hjust = 0.5, vjust = 0.5, interpolate = FALSE,
            na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
geom_rect(mapping = NULL, data = NULL, stat = "identity",
          position = "identity", na.rm = FALSE, show.legend = NA,
          inherit.aes = TRUE, ...)
geom_tile(mapping = NULL, data = NULL, stat = "identity",
          position = "identity", na.rm = FALSE, show.legend = NA,
          inherit.aes = TRUE, ...)
```

Arguments

- **mapping**: Set of aesthetic mappings created by `aes` or `aes_.` If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply mapping if there isn’t a mapping defined for the plot.
- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.
stat  The statistical transformation to use on the data for this layer, as a string.
position  Position adjustment, either as a string, or the result of a call to a position adjustment function.
hjust, vjust  horizontal and vertical justification of the grob. Each justification value should be a number between 0 and 1. Defaults to 0.5 for both, centering each pixel over its data location.
interpolate  If TRUE interpolate linearly, if FALSE (the default) don’t interpolate.
na.rm  If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
show.legend  logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
inherit.aes  If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.

...  other arguments passed on to layer. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
- Other arguments to the layer, for example you override the default stat associated with the layer.
- Other arguments passed on to the stat.

Aesthetics

geom_tile understands the following aesthetics (required aesthetics are in bold):

- x
- y
- alpha
- colour
- fill
- linetype
- size

Examples

# The most common use for rectangles is to draw a surface. You always want
to use geom_raster here because it's so much faster, and produces
# smaller output when saving to PDF
ggplot(faithful, aes(waiting, eruptions)) +
  geom_raster(aes(fill = density))

# Interpolation smooths the surface & is most helpful when rendering images.
ggplot(faithful, aes(waiting, eruptions)) +
  geom_raster(aes(fill = density), interpolate = TRUE)
# If you want to draw arbitrary rectangles, use geom_tile() or geom_rect()

df <- data.frame(
  x = rep(c(2, 5, 7, 9, 12), 2),
  y = rep(c(1, 2), each = 5),
  z = factor(rep(1:5, each = 2)),
  w = rep(diff(c(0, 4, 6, 8, 10, 14)), 2)
)

ggplot(df, aes(x, y)) +
  geom_tile(aes(fill = z))

ggplot(df, aes(x, y)) +
  geom_tile(aes(fill = z, width = w), colour = "grey50")

ggplot(df, aes(xmin = x - w / 2, xmax = x + w / 2, ymin = y, ymax = y + 1)) +
  geom_rect(aes(fill = z, width = w), colour = "grey50")

# Justification controls where the cells are anchored

df <- expand.grid(x = 0:5, y = 0:5)
df$id <- runif(nrow(df))

ggplot(df, aes(x, y, fill = id)) + geom_raster()

# zero padding

ggplot(df, aes(x, y, fill = id)) + geom_raster(hjust = 0, vjust = 0)

# Inspired by the image-density plots of Ken Knoblauch

cars <- ggplot(mtcars, aes(mpg, factor(cyl)))
cars + geom_point()
cars + stat_bin2d(aes(fill = ..count..), binwidth = c(3,1))
cars + stat_bin2d(aes(fill = ..density..), binwidth = c(3,1))

cars + stat_density(aes(fill = ..density..), geom = "raster", position = "identity")
cars + stat_density(aes(fill = ..count..), geom = "raster", position = "identity")

---

**geom_ribbon**

Ribbons and area plots.

**Description**

For each continuous x value, `geom_interval` displays a y interval. `geom_area` is a special case of `geom_ribbon`, where the minimum of the range is fixed to 0.

**Usage**

```r
geom_ribbon(mapping = NULL, data = NULL, stat = "identity",
            position = "identity", na.rm = FALSE, show.legend = NA,
            inherit.aes = TRUE, ...)
```

```r
geom_area(mapping = NULL, data = NULL, stat = "identity",
            position = "stack", na.rm = FALSE, show.legend = NA,
            inherit.aes = TRUE, ...)
```
Arguments

mapping

Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.

data

A data frame. If specified, overrides the default data frame defined at the top level of the plot.

stat

The statistical transformation to use on the data for this layer, as a string.

position

Position adjustment, either as a string, or the result of a call to a position adjustment function.

na.rm

If `FALSE` (the default), removes missing values with a warning. If `TRUE` silently removes missing values.

show.legend

logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.

inherit.aes

If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

... other arguments passed on to `layer`. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- Other arguments to the layer, for example you override the default `stat` associated with the layer.
- Other arguments passed on to the stat.

Details

An area plot is the continuous analog of a stacked bar chart (see `geom_bar`), and can be used to show how composition of the whole varies over the range of x. Choosing the order in which different components is stacked is very important, as it becomes increasing hard to see the individual pattern as you move up the stack.

Aesthetics

`geom_ribbon` understands the following aesthetics (required aesthetics are in bold):

- x
- ymax
- ymin
- alpha
- colour
- fill
- linetype
- size
**geom_rug**

**Marginal rug plots.**

**Description**

Marginal rug plots.

**Usage**

```
geom_rug(mapping = NULL, data = NULL, stat = "identity",
         position = "identity", sides = "btl", na.rm = FALSE, show.legend = NA,
         inherit.aes = TRUE, ...)```

**Arguments**

- **mapping**: Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.
- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.
- **stat**: The statistical transformation to use on the data for this layer, as a string.
- **position**: Position adjustment, either as a string, or the result of a call to a position adjustment function.
- **sides**: A string that controls which sides of the plot the rugs appear on. It can be set to a string containing any of "trbl", for top, right, bottom, and left.
- **na.rm**: If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

**Examples**

```r
# Generate data
huron <- data.frame(year = 1875:1972, level = as.vector(LakeHuron))
h <- ggplot(huron, aes(year))

h + geom.ribbon(aes(ymin=0, ymax=level))
h + geom_area(aes(y = level))

# Add aesthetic mappings
h +
  geom.ribbon(aes(ymin = level - 1, ymax = level + 1), fill = "grey70") +
  geom_line(aes(y = level))
```

**See Also**

- `geom_bar` for discrete intervals (bars), `geom_linerange` for discrete intervals (lines), `geom_polygon` for general polygons

**Marginal rug plots.**
show.legend: logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

inherit.aes: If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

... other arguments passed on to layer. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- Other arguments to the layer, for example you override the default `stat` associated with the layer.
- Other arguments passed on to the stat.

### Aesthetics

`geom_rug` understands the following aesthetics (required aesthetics are in bold):

- `alpha`
- `colour`
- `linetype`
- `size`

### Examples

```r
p <- ggplot(mtcars, aes(wt, mpg))
p + geom_point()
p + geom_point() + geom_rug()  
p + geom_point() + geom_rug(sides="b")  # Rug on bottom only
p + geom_point() + geom_rug(sides="trbl")  # All four sides
p + geom_point() + geom_rug(position='jitter')
```

### Description

`geom_segment` draws a straight line between points (x1, y1) and (x2, y2). `geom_curve` draws a curved line.

### Usage

```r
geom_segment(mapping = NULL, data = NULL, stat = "identity",
position = "identity", arrow = NULL, lineend = "butt", na.rm = FALSE,
show.legend = NA, inherit.aes = TRUE, ...)
```

```r
geom_curve(mapping = NULL, data = NULL, stat = "identity",
position = "identity", arrow = NULL, lineend = "butt", na.rm = FALSE,
show.legend = NA, inherit.aes = TRUE, ...)
```
geom_segment

position = "identity", curvature = 0.5, angle = 90, ncp = 5,
arrow = NULL, lineend = "butt", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...

Arguments

mapping  Set of aesthetic mappings created by \texttt{aes} or \texttt{aes_}. If specified and \texttt{inherit.aes} = \texttt{TRUE} (the default), is combined with the default mapping at the top level of the plot. You only need to supply \texttt{mapping} if there isn’t a mapping defined for the plot.
data    A data frame. If specified, overrides the default data frame defined at the top level of the plot.
stat    The statistical transformation to use on the data for this layer, as a string.
position  Position adjustment, either as a string, or the result of a call to a position adjustment function.
arrow    specification for arrow heads, as created by \texttt{arrow()}
lineend    Line end style (round, butt, square)
na.rm    If \texttt{FALSE} (the default), removes missing values with a warning. If \texttt{TRUE} silently removes missing values.
show.legend    logical. Should this layer be included in the legends? \texttt{NA}, the default, includes if any aesthetics are mapped. \texttt{FALSE} never includes, and \texttt{TRUE} always includes.
inherit.aes    If \texttt{FALSE}, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. \texttt{borders}.
...    other arguments passed on to \texttt{layer}. There are three types of arguments you can use here:
\begin{itemize}
  \item Aesthetics: to set an aesthetic to a fixed value, like \texttt{color = "red"} or \texttt{size = 3}.
  \item Other arguments to the layer, for example you override the default \texttt{stat} associated with the layer.
  \item Other arguments passed on to the stat.
\end{itemize}
curvature    A numeric value giving the amount of curvature. Negative values produce left-hand curves, positive values produce right-hand curves, and zero produces a straight line.
angle    A numeric value between 0 and 180, giving an amount to skew the control points of the curve. Values less than 90 skew the curve towards the start point and values greater than 90 skew the curve towards the end point.
ncp    The number of control points used to draw the curve. More control points creates a smoother curve.

Aesthetics

\texttt{geom_segment} understands the following aesthetics (required aesthetics are in bold):
\begin{itemize}
  \item \texttt{x}
• xend
• y
• yend
• alpha
• colour
• linetype
• size

See Also

geom_path and geom_line for multi-segment lines and paths.
geom_spoke for a segment parameterised by a location (x, y), and an angle and radius.

Examples

b <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point()

df <- data.frame(x1 = 2.62, x2 = 3.57, y1 = 21.0, y2 = 15.0)
b +
  geom_curve(aes(x = x1, y = y1, xend = x2, yend = y2, colour = "curve"), data = df) +
  geom_segment(aes(x = x1, y = y1, xend = x2, yend = y2, colour = "segment"), data = df)

b + geom_curve(aes(x = x1, y = y1, xend = x2, yend = y2), data = df, curvature = -0.2)
b + geom_curve(aes(x = x1, y = y1, xend = x2, yend = y2), data = df, curvature = 1)
b + geom_curve(
  aes(x = x1, y = y1, xend = x2, yend = y2),
  data = df,
  arrow = arrow(length = unit(0.03, "npc"))
)

ggplot(seals, aes(long, lat)) +
  geom_segment(aes(xend = long + delta_long, yend = lat + delta_lat),
               arrow = arrow(length = unit(0.1, "cm"))) +
  borders("state")

# You can also use geom_segment to recreate plot(type = "h") :
counts <- as.data.frame(table(x = rpois(100,5)))
counts$x <- as.numeric(as.character(counts$x))
with(counts, plot(x, Freq, type = "h", lwd = 10))

ggplot(counts, aes(x, Freq)) +
  geom_segment(aes(xend = x, yend = 0), size = 10, lineend = "butt")
**geom_smooth**

*Add a smoothed conditional mean.*

**Description**

Aids the eye in seeing patterns in the presence of overplotting. `geom_smooth` and `stat_smooth` are effectively aliases: they both use the same arguments. Use `geom_smooth` unless you want to display the results with a non-standard geom.

**Usage**

```r
geom_smooth(mapping = NULL, data = NULL, stat = "smooth",
method = "auto", formula = y ~ x, se = TRUE, position = "identity",
na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
```

```r
stat_smooth(mapping = NULL, data = NULL, geom = "smooth",
position = "identity", method = "auto", formula = y ~ x, se = TRUE,
n = 80, span = 0.75, fullrange = FALSE, level = 0.95,
method.args = list(), na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)
```

**Arguments**

- **mapping**
  - Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.

- **data**
  - A data frame. If specified, overrides the default data frame defined at the top level of the plot.

- **method**
  - Smoothing method (function) to use, eg. `lm`, `glm`, `gam`, `loess`, `rlm`. For datasets with n < 1000 default is `loess`. For datasets with 1000 or more observations defaults to `gam`, see `gam` for more details.

- **formula**
  - Formula to use in smoothing function, eg. `y ~ x`, `y ~ poly(x, 2)`, `y ~ log(x)`

- **se**
  - Display confidence interval around smooth? (TRUE by default, see level to control)

- **position**
  - Position adjustment, either as a string, or the result of a call to a position adjustment function.

- **na.rm**
  - If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.

- **show.legend**
  - Logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.

- **inherit.aes**
  - If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

- **...**
  - Other arguments passed on to `layer`. There are three types of arguments you can use here:
• Aesthetics: to set an aesthetic to a fixed value, like color = "red" or size = 3.
• Other arguments to the layer, for example you override the default stat associated with the layer.
• Other arguments passed on to the stat.

geom, stat
Use to override the default connection between geom_smooth and stat_smooth.
n
number of points to evaluate smoother at
span
Controls the amount of smoothing for the default loess smoother. Smaller numbers produce wigglier lines, larger numbers produce smoother lines.
fullrange
should the fit span the full range of the plot, or just the data
level
level of confidence interval to use (0.95 by default)
method.args
List of additional arguments passed on to the modelling function defined by method.

Details
Calculation is performed by the (currently undocumented) predictdf generic and its methods. For most methods the standard error bounds are computed using the predict method - the exceptions are loess which uses a t-based approximation, and glm where the normal confidence interval is constructed on the link scale, and then back-transformed to the response scale.

Aesthetics
geom_smooth understands the following aesthetics (required aesthetics are in bold):

• x
• y
• alpha
• colour
• fill
• linetype
• size
• weight

Computed variables
y predicted value
ymin lower pointwise confidence interval around the mean
ymax upper pointwise confidence interval around the mean
se standard error

See Also
See individual modelling functions for more details: lm for linear smooths, glm for generalised linear smooths, loess for local smooths
**Examples**

```r
ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth()

# Use span to control the "wiggliness" of the default loess smoother
# The span is the fraction of points used to fit each local regression:
# small numbers make a wigglier curve, larger numbers make a smoother curve.
ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth(span = 0.3)

# Instead of a loess smooth, you can use any other modelling function:
ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth(method = "lm", se = FALSE)

ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth(method = "lm", formula = y ~ splines::bs(x, 3), se = FALSE)

# Smoothes are automatically fit to each group (defined by categorical
# aesthetics or the group aesthetic) and for each facet

ggplot(mpg, aes(displ, hwy, colour = class)) + geom_point() + geom_smooth(se = FALSE, method = "lm")

ggplot(mpg, aes(displ, hwy)) + geom_point() + geom_smooth(span = 0.8) + facet_wrap(~drv)

## Not run:
# To fit a logistic regression, you need to coerce the values to
# a numeric vector lying between 0 and 1.
binomial_smooth <- function(...) {
  geom_smooth(method = "glm", method.args = list(family = "binomial"), ...)
}

ggplot(rpart::kyphosis, aes(Age, Kyphosis)) + geom_jitter(height = 0.05) + binomial_smooth()

ggplot(rpart::kyphosis, aes(Age, as.numeric(Kyphosis) - 1)) + geom_jitter(height = 0.05) + binomial_smooth()

ggplot(rpart::kyphosis, aes(Age, as.numeric(Kyphosis) - 1)) + geom_jitter(height = 0.05) + binomial_smooth(formula = y ~ splines::ns(x, 2))

# But in this case, it's probably better to fit the model yourself
# so you can exercise more control and see whether or not it's a good model

## End(Not run)

**geom_spoke**

A line segment parameterised by location, direction and distance.

### Description

A line segment parameterised by location, direction and distance.

### Usage

```r
gem.spoke(mapping = NULL, data = NULL, stat = "identity", position = "identity", na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)
```

### Arguments

- **mapping**: Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.
- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.
- **stat**: The statistical transformation to use on the data for this layer, as a string.
- **position**: Position adjustment, either as a string, or the result of a call to a position adjustment function.
- **na.rm**: If `FALSE` (the default), removes missing values with a warning. If `TRUE` silently removes missing values.
- **show.legend**: logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.
- **inherit.aes**: If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.
- **...**: other arguments passed on to `layer`. There are three types of arguments you can use here:
  - Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
  - Other arguments to the layer, for example you override the default `stat` associated with the layer.
  - Other arguments passed on to the stat.
Aesthetics

gem_spoke understands the following aesthetics (required aesthetics are in bold):

- angle
- radius
- x
- y
- alpha
- colour
- linetype
- size

Examples

def <- expand.grid(x = 1:10, y = 1:10)
def$angle <- runif(100, 0, 2*pi)
def$speed <- runif(100, 0, sqrt(0.1 * df$x))

```r
# Example 1
ggplot(def, aes(x, y)) +
  geom_point() +
  geom_spoke(aes(angle = angle), radius = 0.5)
```

```r
# Example 2
ggplot(def, aes(x, y)) +
  geom_point() +
  geom_spoke(aes(angle = angle, radius = speed))
```

---

**geom_violin**  
Violin plot.

Description

Violin plot.

Usage

gem_violin(mapping = NULL, data = NULL, stat = "ydensity",
  draw_quantiles = NULL, position = "dodge", trim = TRUE,
  scale = "area", na.rm = FALSE, show.legend = NA, inherit.aes = TRUE,
  ...)

stat_ydensity(mapping = NULL, data = NULL, geom = "violin",
  position = "dodge", adjust = 1, kernel = "gaussian", trim = TRUE,
  scale = "area", na.rm = FALSE, show.legend = NA, inherit.aes = TRUE,
  ...)
Arguments

- **mapping**: Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there’s not a mapping defined for the plot.

- **data**: A data frame. If specified, overrides the default data frame defined at the top level of the plot.

- **draw_quantiles**: If `not,NULL` (default), draw horizontal lines at the given quantiles of the density estimate.

- **position**: Position adjustment, either as a string, or the result of a call to a position adjustment function.

- **trim**: If `TRUE` (default), trim the tails of the violins to the range of the data. If `FALSE`, don’t trim the tails.

- **scale**: if "area" (default), all violins have the same area (before trimming the tails). If "count", areas are scaled proportionally to the number of observations. If "width", all violins have the same maximum width.

- **na.rm**: If `FALSE` (the default), removes missing values with a warning. If `TRUE` silently removes missing values.

- **show.legend**: logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.

- **inherit.aes**: If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

... other arguments passed on to `layer`. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- Other arguments to the layer, for example you override the default `stat` associated with the layer.
- Other arguments passed on to the stat.

- **geom, stat**: Use to override the default connection between `geom_violin` and `stat_ydensity`.
- **adjust**: see `density` for details
- **kernel**: kernel used for density estimation, see `density` for details

Aesthetics

gem_violin understands the following aesthetics (required aesthetics are in bold):

- **x**
- **y**
- **alpha**
- **colour**
- **fill**
- **linetype**
- **size**
- **weight**
Computed variables

- **density**: density estimate
- **scaled**: density estimate, scaled to maximum of 1
- **count**: density * number of points - probably useless for violin plots
- **violinwidth**: density scaled for the violin plot, according to area, counts or to a constant maximum width
- **n**: number of points
- **width**: width of violin bounding box

References


See Also

- `geom_violin` for examples, and `stat_density` for examples with data along the x axis.

Examples

```r
p <- ggplot(mtcars, aes(factor(cyl), mpg))
p + geom_violin()
```

```r
p + geom_violin() + geom_jitter(height = 0)
p + geom_violin() + coord_flip()
```

```
# Scale maximum width proportional to sample size:
p + geom_violin(scale = "count")

# Scale maximum width to 1 for all violins:
p + geom_violin(scale = "width")

# Default is to trim violins to the range of the data. To disable:
p + geom_violin(trim = FALSE)

# Use a smaller bandwidth for closer density fit (default is 1).
p + geom_violin(adjust = .5)

# Add aesthetic mappings
# Note that violins are automatically dodged when any aesthetic is
# a factor
p + geom_violin(aes(fill = cyl))
p + geom_violin(aes(fill = factor(cyl)))
p + geom_violin(aes(fill = factor(vs)))
p + geom_violin(aes(fill = factor(am)))

# Set aesthetics to fixed value
p + geom_violin(fill = "grey80", colour = "#3366FF")
```
# Show quartiles
p + geom_violin(draw_quantiles = c(0.25, 0.5, 0.75))

# Scales vs. coordinate transforms -------
if (require("ggplot2movies")) {
  # Scale transformations occur before the density statistics are computed.
  # Coordinate transformations occur afterwards. Observe the effect on the
  # number of outliers.
  m <- ggplot(movies, aes(y = votes, x = rating, group = cut_width(rating, 0.5)))
  m + geom_violin()
  m + geom_violin() + scale_y_log10()
  m + geom_violin() + coord_trans(y = "log10")
  m + geom_violin() + scale_y_log10() + coord_trans(y = "log10")

  # Violin plots with continuous x:
  # Use the group aesthetic to group observations in violins
  ggplot(movies, aes(year, budget)) + geom_violin()
  ggplot(movies, aes(year, budget)) +
    geom_violin(aes(group = cut_width(year, 10)), scale = "width")
}

---

ggplot

Create a new ggplot plot.

Description

ggplot() initializes a ggplot object. It can be used to declare the input data frame for a graphic and to specify the set of plot aesthetics intended to be common throughout all subsequent layers unless specifically overridden.

Usage

ggplot(data = NULL, mapping = aes(), ..., environment = parent.frame())

Arguments

data Default dataset to use for plot. If not already a data.frame, will be converted to one by fortify. If not specified, must be supplied in each layer added to the plot.

mapping Default list of aesthetic mappings to use for plot. If not specified, must be supplied in each layer added to the plot.

... Other arguments passed on to methods. Not currently used.

environment If an variable defined in the aesthetic mapping is not found in the data, ggplot will look for it in this environment. It defaults to using the environment in which ggplot() is called.
ggplot

Details

`ggplot()` is typically used to construct a plot incrementally, using the + operator to add layers to the existing ggplot object. This is advantageous in that the code is explicit about which layers are added and the order in which they are added. For complex graphics with multiple layers, initialization with `ggplot()` is recommended.

There are three common ways to invoke `ggplot`:

- `ggplot(df, aes(x, y, <other aesthetics>))`
- `ggplot(df)`
- `ggplot()`

The first method is recommended if all layers use the same data and the same set of aesthetics, although this method can also be used to add a layer using data from another data frame. See the first example below. The second method specifies the default data frame to use for the plot, but no aesthetics are defined up front. This is useful when one data frame is used predominantly as layers are added, but the aesthetics may vary from one layer to another. The third method initializes a skeleton `ggplot` object which is fleshed out as layers are added. This method is useful when multiple data frames are used to produce different layers, as is often the case in complex graphics.

Examples

df <- data.frame(gp = factor(rep(letters[1:3], each = 10)),
                 y = rnorm(30))
# Compute sample mean and standard deviation in each group
ds <- plyr::ddply(df, "gp", plyr::summarise, mean = mean(y), sd = sd(y))

# Declare the data frame and common aesthetics.
# The summary data frame ds is used to plot
# larger red points in a second geom_point() layer.
# If the data = argument is not specified, it uses the
# declared data frame from ggplot(); ditto for the aesthetics.
ggplot(df, aes(x = gp, y = y)) +
    geom_point() +
    geom_point(data = ds, aes(y = mean),
               colour = 'red', size = 3)
# Same plot as above, declaring only the data frame in ggplot().
# Note how the x and y aesthetics must now be declared in
# each geom_point() layer.
ggplot(df) +
    geom_point(aes(x = gp, y = y)) +
    geom_point(data = ds, aes(x = gp, y = mean),
               colour = 'red', size = 3)
# Set up a skeleton ggplot object and add layers:
ggplot() +
    geom_point(data = df, aes(x = gp, y = y)) +
    geom_point(data = ds, aes(x = gp, y = mean),
               colour = 'red', size = 3) +
    geom_errorbar(data = ds, aes(x = gp, y = mean,
                                 ymin = mean - sd, ymax = mean + sd),
                  colour = 'red', width = 0.4)
ggproto

Create a new ggproto object

Description

ggproto is inspired by the proto package, but it has some important differences. Notably, it cleanly supports cross-package inheritance, and has faster performance.

Usage

ggproto(`_class` = NULL, `_inherit` = NULL, ...)

ggproto_parent(parent, self)

Arguments

_class Class name to assign to the object. This is stored as the class attribute of the object. If NULL (the default), no class name will be added to the object.
_inherit ggproto object to inherit from. If NULL, don’t inherit from any object.
... A list of members in the ggproto object.
parent, self Access parent class parent of object self.

Calling ggproto methods

ggproto methods can take an optional self argument: if it is present, it is a regular method; if it’s absent, it’s a “static” method (i.e. it doesn’t use any fields).

Imagine you have a ggproto object Adder, which has a method addx = function(self, n) n + self$x. Then, to call this function, you would use Adder$addx(10) – the self is passed in automatically by the wrapper function. self be located anywhere in the function signature, although customarily it comes first.

Calling methods in a parent

To explicitly call a methods in a parent, use ggproto_parent(Parent, self).

ggsave

Save a ggplot (or other grid object) with sensible defaults

Description

ggsave() is a convenient function for saving a plot. It defaults to saving the last plot that you displayed, using the size of the current graphics device. It also guesses the type of graphics device from the extension.
Usage

ggsave(filename, plot = last_plot(), device = NULL, path = NULL,
   scale = 1, width = NA, height = NA, units = c("in", "cm", "mm"),
   dpi = 300, limitsize = TRUE, ...)

Arguments

filename File name to create on disk.
plot Plot to save, defaults to last plot displayed.
device Device to use (function or any of the recognized extensions, e.g. "pdf"). By default, extracted from filename extension. ggsave currently recognises eps/ps, tex (pictex), pdf, jpeg, tiff, png, bmp, svg and wmf (windows only).
path Path to save plot to (combined with filename).
scale Multiplicative scaling factor.
width, height Plot dimensions, defaults to size of current graphics device.
units Units for width and height when specified explicitly (in, cm, or mm)
dpi Resolution used for raster outputs.
limitsize When TRUE (the default), ggsave will not save images larger than 50x50 inches, to prevent the common error of specifying dimensions in pixels.
... Other arguments passed on to graphics device

Examples

## Not run:
ggplot(mtcars, aes(mpg, wt)) + geom_point()

ggsave("mtcars.pdf")
ggsave("mtcars.png")

ggsave("mtcars.pdf", width = 4, height = 4)
ggsave("mtcars.pdf", width = 20, height = 20, units = "cm")

unlink("mtcars.pdf")
unlink("mtcars.png")

# specify device when saving to a file with unknown extension
# (for example a server supplied temporary file)
file <- tempfile()
ggsave(file, device = "pdf")
unlink(file)

## End(Not run)
Description

Themes set the general aspect of the plot such as the colour of the background, gridlines, the size and colour of fonts.

Usage

```r
theme_grey(base_size = 11, base_family = "")
theme_gray(base_size = 11, base_family = "")
theme_bw(base_size = 12, base_family = "")
theme_linedraw(base_size = 12, base_family = "")
theme_light(base_size = 12, base_family = "")
theme_minimal(base_size = 12, base_family = "")
theme_classic(base_size = 12, base_family = "")
theme_dark(base_size = 12, base_family = "")
theme_void(base_size = 12, base_family = "")
```

Arguments

- `base_size` base font size
- `base_family` base font family

Details

- `theme_gray` The signature ggplot2 theme with a grey background and white gridlines, designed to put the data forward yet make comparisons easy.
- `theme_bw` The classic dark-on-light ggplot2 theme. May work better for presentations displayed with a projector.
- `theme_linedraw` A theme with only black lines of various widths on white backgrounds, reminiscent of a line drawings. Serves a purpose similar to `theme_bw`. Note that this theme has some very thin lines (< 1 pt) which some journals may refuse.
- `theme_light` A theme similar to `theme_linedraw` but with light grey lines and axes, to direct more attention towards the data.
- `theme_dark` The dark cousin of `theme_light`, with similar line sizes but a dark background. Useful to make thin coloured lines pop out.
theme_minimal A minimalistic theme with no background annotations.
theme_classic A classic-looking theme, with x and y axis lines and no gridlines.
theme_void A completely empty theme.

Examples

```r
p <- ggplot(mtcars) + geom_point(aes(x = wt, y = mpg,
                  colour=factor(gear))) + facet_wrap(~am)

p
p + theme_gray()
p + theme_bw()
p + theme_linedraw()
p + theme_light()
p + theme_minimal()
p + theme_classic()
p + theme_void()
```

### Description

Multiply size in mm by these constants in order to convert to the units that grid uses internally for lwd and font.size.

### Usage

- .pt

- stroke

### Format

An object of class numeric of length 1.
guides

Set guides for each scale.

Description

Guides for each scale can be set in call of scale_* with argument guide, or in guides.

Usage

guides(...)

Arguments

... List of scale guide pairs

Value

A list containing the mapping between scale and guide.

See Also

Other guides: guide_colourbar, guide_legend

Examples

# ggplot object
dat <- data.frame(x = 1:5, y = 1:5, p = 1:5, q = factor(1:5),
    r = factor(1:5))
p <- ggplot(dat, aes(x, y, colour = p, size = q, shape = r)) + geom_point()

# without guide specification
p

# Show colorbar guide for colour.
# All these examples below have a same effect.
p + guides(colour = "colorbar", size = "legend", shape = "legend")
p + guides(colour = guide_colorbar(), size = guide_legend(),
    shape = guide_legend())
p +
    scale_colour_continuous(guide = "colorbar") +
    scale_size_discrete(guide = "legend") +
    scale_shape(guide = "legend")

# Remove some guides
p + guides(colour = "none")
p + guides(colour = "colorbar", size = "none")
# Guides are integrated where possible

```r
p + guides(colour = guide_legend("title"), size = guide_legend("title"),
shape = guide_legend("title"))
# same as
g <- guide_legend("title")
p + guides(colour = g, size = g, shape = g)
```

```r
p + theme(legend.position = "bottom")
```

# position of guides

```r
colour = guide_colourbar(order = 1),
shape = guide_legend(order = 2),
size = guide_legend(order = 3)
```

---

**guide_colourbar**  
*Continuous colour bar guide.*

---

**Description**

Colour bar guide shows continuous color scales mapped onto values. Colour bar is available with `scale_fill` and `scale_colour`. For more information, see the inspiration for this function: Matlab’s colorbar function.

**Usage**

```r
guide_colourbar(title = waiver(), title.position = NULL,
title.theme = NULL, title.hjust = NULL, title.vjust = NULL,
label = TRUE, label.position = NULL, label.theme = NULL,
label.hjust = NULL, label.vjust = NULL, barwidth = NULL,
barheight = NULL, nbin = 20, raster = TRUE, ticks = TRUE,
draw.ulim = TRUE, draw.llim = TRUE, direction = NULL,
default.unit = "line", reverse = FALSE, order = 0, ...)
guide_colorbar(title = waiver(), title.position = NULL,
title.theme = NULL, title.hjust = NULL, title.vjust = NULL,
label = TRUE, label.position = NULL, label.theme = NULL,
label.hjust = NULL, label.vjust = NULL, barwidth = NULL,
barheight = NULL, nbin = 20, raster = TRUE, ticks = TRUE,
draw.ulim = TRUE, draw.llim = TRUE, direction = NULL,
default.unit = "line", reverse = FALSE, order = 0, ...)
```
Arguments

title A character string or expression indicating a title of guide. If NULL, the title is not shown. By default (waiver), the name of the scale object or the name specified in labs is used for the title.
title.position A character string indicating the position of a title. One of "top" (default for a vertical guide), "bottom", "left" (default for a horizontal guide), or "right."
title.theme A theme object for rendering the title text. Usually the object of element_text is expected. By default, the theme is specified by legend.title in theme or theme.
title.hjust A number specifying horizontal justification of the title text.
title.vjust A number specifying vertical justification of the title text.
label logical. If TRUE then the labels are drawn. If FALSE then the labels are invisible.
label.position A character string indicating the position of a label. One of "top", "bottom" (default for horizontal guide), "left", or "right" (default for vertical guide).
label.theme A theme object for rendering the label text. Usually the object of element_text is expected. By default, the theme is specified by legend.text in theme or theme.
label.hjust A numeric specifying horizontal justification of the label text.
label.vjust A numeric specifying vertical justification of the label text.
barwidth A numeric or a unit object specifying the width of the colorbar. Default value is legend.key.width or legend.key.size in theme or theme.
barheight A numeric or a unit object specifying the height of the colorbar. Default value is legend.key.height or legend.key.size in theme or theme.
nbin A numeric specifying the number of bins for drawing colorbar. A smoother colorbar for a larger value.
raster A logical. If TRUE then the colorbar is rendered as a raster object. If FALSE then the colorbar is rendered as a set of rectangles. Note that not all graphics devices are capable of rendering raster image.
ticks A logical specifying if tick marks on colorbar should be visible.
draw.ulim A logical specifying if the upper limit tick marks should be visible.
draw.llim A logical specifying if the lower limit tick marks should be visible.
direction A character string indicating the direction of the guide. One of "horizontal" or "vertical."
default.unit A character string indicating unit for barwidth and barheight.
reverse logical. If TRUE the colorbar is reversed. By default, the highest value is on the top and the lowest value is on the bottom
order positive integer less that 99 that specifies the order of this guide among multiple guides. This controls the order in which multiple guides are displayed, not the contents of the guide itself. If 0 (default), the order is determined by a secret algorithm.
... ignored.
Details

Guides can be specified in each scale_.* or in guides. guide=“legend” in scale_.* is syntactic sugar for guide=guide_legend() (e.g. scale_color_manual(guide = "legend"). As for how to specify the guide for each scale in more detail, see guides.

Value

A guide object

See Also

Other guides: guide_legend, guides

Examples

df <- reshape2::melt(outer(1:4, 1:4), varnames = c("X1", "X2"))

p1 <- ggplot(df, aes(X1, X2)) + geom_tile(aes(fill = value))
p2 <- p1 + geom_point(aes(size = value))

# Basic form
p1 + scale_fill_continuous(guide = "colorbar")
p1 + scale_fill_continuous(guide = guide_colorbar())
p1 + guides(fill = guide_colorbar())

# Control styles

# bar size
p1 + guides(fill = guide_colorbar(barwidth = 0.5, barheight = 10))

# no label
p1 + guides(fill = guide_colorbar(label = FALSE))

# no tick marks
p1 + guides(fill = guide_colorbar(ticks = FALSE))

# label position
p1 + guides(fill = guide_colorbar(label.position = "left"))

# label theme
p1 + guides(fill = guide_colorbar(label.theme = element_text(colour = "blue", angle = 0)))

# small number of bins
p1 + guides(fill = guide_colorbar(nbin = 3))

# large number of bins
p1 + guides(fill = guide_colorbar(nbin = 100))

# make top- and bottom-most ticks invisible
p1 + scale_fill_continuous(limits = c(0, 20), breaks=c(0, 5, 10, 15, 20),
guide = guide_colorbar(nbin=100, draw. ulim = FALSE, draw. llim = FALSE))
# guides can be controlled independently
p2 +
  scale_fill_continuous(guide = "colorbar") +
  scale_size(guide = "legend")
p2 + guides(fill = "colorbar", size = "legend")

p2 +
  scale_fill_continuous(guide = guide_colorbar(direction = "horizontal")) +
  scale_size(guide = guide_legend(direction = "vertical"))

---

**guide_legend**

*Legend guide.*

**Description**

Legend type guide shows key (i.e., geoms) mapped onto values. Legend guides for various scales are integrated if possible.

**Usage**

```r
guide_legend(title = waiver(), title.position = NULL, title.theme = NULL,
  title.hjust = NULL, title.vjust = NULL, label = TRUE,
  label.position = NULL, label.theme = NULL, label.hjust = NULL,
  label.vjust = NULL, keywidth = NULL, keyheight = NULL,
  direction = NULL, default.unit = "line", override.aes = list(),
  nrow = NULL, ncol = NULL, byrow = FALSE, reverse = FALSE, order = 0,
  ...)
```

**Arguments**

- **title**: A character string or expression indicating a title of guide. If NULL, the title is not shown. By default (*waiver*), the name of the scale object or the name specified in `labs` is used for the title.
- **title.position**: A character string indicating the position of a title. One of "top" (default for a vertical guide), "bottom", "left" (default for a horizontal guide), or "right."
- **title.theme**: A theme object for rendering the title text. Usually the object of `element_text` is expected. By default, the theme is specified by `legend.title` in `theme` or `theme.``
- **title.hjust**: A number specifying horizontal justification of the title text.
- **title.vjust**: A number specifying vertical justification of the title text.
- **label**: logical. If TRUE then the labels are drawn. If FALSE then the labels are invisible.
- **label.position**: A character string indicating the position of a label. One of "top", "bottom" (default for horizontal guide), "left", or "right" (default for vertical guide).
- **label.theme**: A theme object for rendering the label text. Usually the object of `element_text` is expected. By default, the theme is specified by `legend.text` in `theme` or `theme`.
label.hjust  A numeric specifying horizontal justification of the label text.
label.vjust  A numeric specifying vertical justification of the label text.
keywidth     A numeric or a unit object specifying the width of the legend key. Default value is legend.key.width or legend.key.size in theme or theme.
keyheight    A numeric or a unit object specifying the height of the legend key. Default value is legend.key.height or legend.key.size in theme or theme.
direction    A character string indicating the direction of the guide. One of "horizontal" or "vertical."
default.unit A character string indicating unit for keywidth and keyheight.
override.aes A list specifying aesthetic parameters of legend key. See details and examples.
nrow         The desired number of rows of legends.
ncol         The desired number of column of legends.
byrow        logical. If FALSE (the default) the legend-matrix is filled by columns, otherwise the legend-matrix is filled by rows.
reverse      logical. If TRUE the order of legends is reversed.
order        positive integer less that 99 that specifies the order of this guide among multiple guides. This controls the order in which multiple guides are displayed, not the contents of the guide itself. If 0 (default), the order is determined by a secret algorithm.
...          ignored.

Details

Guides can be specified in each scale_* or in guides. guide="legend" in scale_* is syntactic sugar for guide=guide_legend() (e.g. scale_color_manual(guide = "legend")). As for how to specify the guide for each scale in more detail, see guides.

Value

A guide object

See Also

Other guides: guide_colourbar, guides

Examples

```r
df <- reshape2::melt(outer(1:4, 1:4), varnames = c("X1", "X2"))

p1 <- ggplot(df, aes(X1, X2)) + geom_tile(aes(fill = value))
p2 <- p1 + geom_point(aes(size = value))

# Basic form
p1 + scale_fill_continuous(guide = "legend")
p1 + scale_fill_continuous(guide = guide_legend())
```
# Guide title

```r
p1 + scale_fill_continuous(guide = guide_legend(title = "V"))  # title text
p1 + scale_fill_continuous(guide = guide_legend(title = NULL))  # no title
```

# Control styles

# key size

```r
p1 + guides(fill = guide_legend(keywidth = 3, keyheight = 1))
```

# title position

```r
p1 + guides(fill = guide_legend(title = "LEFT", title.position = "left"))
```

# title text styles via element_text

```r
p1 + guides(fill = guide_legend(
  title.theme = element_text(size=15, face="italic", colour = "red", angle = 45))
```

# label position

```r
p1 + guides(fill = guide_legend(label.position = "bottom"))
```

# label styles

```r
p1 + scale_fill_continuous(breaks = c(5, 10, 15),
  labels = paste("long", c(5, 10, 15)),
  guide = guide_legend(direction = "horizontal", title.position = "top",
    label.position="bottom", label.hjust = 0.5, label.vjust = 0.5,
    label.theme = element_text(angle = 90)))
```

# Set aesthetic of legend key

# very low alpha value make it difficult to see legend key

```r
p3 <- ggplot(diamonds, aes(carat, price)) +
  geom_point(aes(colour=color), alpha=1/100)
p3
```

# override.aes overwrites the alpha

```r
p3 + guides(colour = guide_legend(override.aes = list(alpha = 1)))
```

# multiple row/col legends

```r
df <- data.frame(x = 1:20, y = 1:20, color = letters[1:20])
p <- ggplot(df, aes(x, y)) +
  geom_point(aes(colour = color))
p + guides(col = guide_legend(nrow = 8))
p + guides(col = guide_legend(ncol = 8))
p + guides(col = guide_legend(nrow = 8, byrow = TRUE))
p + guides(col = guide_legend(ncol = 8, byrow = TRUE))
```

# reversed order legend

```r
p + guides(col = guide_legend(reverse = TRUE))
```
hmisc  Wrap up a selection of summary functions from Hmisc to make it easy
         to use with stat_summary.

Description
See the Hmisc documentation for details of their options.

Usage
mean_cl_boot(x, ...)
mean_cl_normal(x, ...)
mean_sdl(x, ...)
median_hilow(x, ...)

Arguments
x         a numeric vector
...       other arguments passed on to the respective Hmisc function.

See Also
smeanNclNboot, smeanNclNnormal, smeanNsdNl, smedianNhilow

is.ggproto  Is an object a ggproto object?

Description
Is an object a ggproto object?

Usage
is.ggproto(x)

Arguments
x         An object to test.
is.rel  

Reports whether x is a rel object

Description

Reports whether x is a rel object

Usage

is.rel(x)

Arguments

x  An object to test

is.theme  

Reports whether x is a theme object

Description

Reports whether x is a theme object

Usage

is.theme(x)

Arguments

x  An object to test

labeller  

Generic labeller function for facets

Description

This function makes it easy to assign different labellers to different factors. The labeller can be a function or it can be a named character vectors that will serve as a lookup table.

Usage

labeller(..., .rows = NULL, .cols = NULL, keep.as.numeric = NULL, .multi_line = TRUE, .default = label_value)
Arguments

... Named arguments of the form `variable = labeller`. Each labeller is passed to `as_labeller()` and can be a lookup table, a function taking and returning character vectors, or simply a labeller function.

`.rows, .cols` Labeller for a whole margin (either the rows or the columns). It is passed to `as_labeller()`. When a margin-wide labeller is set, make sure you don’t mention in ... any variable belonging to the margin.

`keep.as.numeric` Deprecated. All supplied labellers and on-labeller functions should be able to work with character labels.

`.multi_line` Whether to display the labels of multiple factors on separate lines. This is passed to the labeller function.

`.default` Default labeller for variables not specified. Also used with lookup tables or non-labeller functions.

Details

In case of functions, if the labeller has class `labeller`, it is directly applied on the data frame of labels. Otherwise, it is applied to the columns of the data frame of labels. The data frame is then processed with the function specified in the `.default` argument. This is intended to be used with functions taking a character vector such as `capitalize`.

Value

A labeller function to supply to `facet_grid` for the argument `labeller`.

See Also

`as_labeller()`, `labellers`

Examples

```r
p1 <- ggplot(mtcars, aes(x = mpg, y = wt)) + geom_point()

# You can assign different labellers to variables:
p1 + facet_grid(vs + am ~ gear,
  labeller = labeller(vs = label_both, am = label_value))

# Or whole margins:
p1 + facet_grid(vs + am ~ gear,
  labeller = labeller(.rows = label_both, .cols = label_value))

# You can supply functions operating on strings:
capitalize <- function(string) {
  substr(string, 1, 1) <- toupper(substr(string, 1, 1))
  string
}
p2 <- ggplot(msleep, aes(x = sleep_total, y = awake)) + geom_point()
```
labellers

```r
p2 + facet_grid(vore ~ conservation, labeller = labeller(vore = capitalize))

# Or use character vectors as lookup tables:
conservation_status <- c(
  cd = "Conservation Dependent",
  en = "Endangered",
  lc = "Least Concern",
  nt = "Near Threatened",
  vu = "Vulnerable",
  domesticated = "Domesticated"
)
p2 + facet_grid(vore ~ conservation, labeller = labeller(
  .default = capitalize,
  conservation = conservation_status
))

# In the following example, we rename the levels to the long form,
# then apply a wrap labeller to the columns to prevent cropped text
msleep$conservation2 <- plyr::revalue(msleep$conservation, conservation_status)
p2 %>% msleep + facet_grid(vore ~ conservation2)
p2 %>% msleep +
  facet_grid(vore ~ conservation2,
    labeller = labeller(conservation2 = label_wrap_gen(10))
  )

# labeller() is especially useful to act as a global labeller. You
# can set it up once and use it on a range of different plots with
# different facet specifications.

global_labeller <- labeller(
  vore = capitalize,
  conservation = conservation_status,
  conservation2 = label_wrap_gen(10),
  .default = label_both
)
p2 + facet_grid(vore ~ conservation, labeller = global_labeller)
p2 + facet_wrap(~ vore, labeller = global_labeller)
p2 %>% msleep + facet_wrap(~ conservation2, labeller = global_labeller)
```

---

labellers  

Labeller functions

---
Description

Labeller functions are in charge of formatting the strip labels of facet grids and wraps. Most of them accept a multi_line argument to control whether multiple factors (defined in formulae such as ~first + second) should be displayed on a single line separated with commas, or each on their own line.

Usage

label_value(labels, multi_line = TRUE)

label_both(labels, multi_line = TRUE, sep = "": ")

label_context(labels, multi_line = TRUE, sep = "": ")

label_parsed(labels, multi_line = TRUE)

label_wrap_gen(width = 25, multi_line = TRUE)

Arguments

labels Data frame of labels. Usually contains only one element, but facetting over multiple factors entails multiple label variables.

multi_line Whether to display the labels of multiple factors on separate lines.

sep String separating variables and values.

width Maximum number of characters before wrapping the strip.

Details

label_value() only displays the value of a factor while label_both() displays both the variable name and the factor value. label_context() is context-dependent and uses label_value() for single factor facetting and label_both() when multiple factors are involved. label_wrap_gen() uses strwrap() for line wrapping.

label_parsed() interprets the labels as plotmath expressions. label_bquote() offers a more flexible way of constructing plotmath expressions. See examples and bquote() for details on the syntax of the argument.

Writing New Labeller Functions

Note that an easy way to write a labeller function is to transform a function operating on character vectors with as_labeller().

A labeller function accepts a data frame of labels (character vectors) containing one column for each factor. Multiple factors occur with formula of the type ~first + second.

The return value must be a rectangular list where each 'row' characterises a single facet. The list elements can be either character vectors or lists of plotmath expressions. When multiple elements are returned, they get displayed on their own new lines (i.e., each facet gets a multi-line strip of labels).
To illustrate, let's say your labeller returns a list of two character vectors of length 3. This is a rectangular list because all elements have the same length. The first facet will get the first elements of each vector and display each of them on their own line. Then the second facet gets the second elements of each vector, and so on.

If it's useful to your labeller, you can retrieve the type attribute of the incoming data frame of labels. The value of this attribute reflects the kind of strips your labeller is dealing with: "cols" for columns and "rows" for rows. Note that facet_wrap() has columns by default and rows when the strips are switched with the switch option. The facet attribute also provides metadata on the labels. It takes the values "grid" or "wrap".

For compatibility with labeller(), each labeller function must have the labeller S3 class.

See Also

labeller(), as_labeller(), label_bquote()

Examples

```r
mtcars$cyl <- factor(mtcars$cyl, labels = c("alpha", "beta", "gamma"))
p <- ggplot(mtcars, aes(wt, mpg)) + geom_point()

# Displaying only the values
p + facet_grid(. ~ cyl)
p + facet_grid(. ~ cyl, labeller = label_value)

# Displaying both the values and the variables
p + facet_grid(. ~ cyl, labeller = label_both)

# Displaying only the values or both the values and variables
# depending on whether multiple factors are facetted over
p + facet_grid(am ~ vs+cyl, labeller = label_context)

# Interpreting the labels as plotmath expressions
p + facet_grid(. ~ cyl2)
p + facet_grid(. ~ cyl2, labeller = label_parsed)
p + facet_wrap(-vs + cyl2, labeller = label_parsed)
```

Description

`label_bquote()` offers a flexible way of labelling facet rows or columns with plotmath expressions. Backquoted variables will be replaced with their value in the facet.

Usage

```r
label_bquote(rows = NULL, cols = NULL, default = label_value)
```
labs

Change axis labels and legend titles

Description

Change axis labels and legend titles

Usage

labs(...)

xlab(label)

ylab(label)

ggtitle(label)

Arguments

... a list of new names in the form aesthetic = "new name"

label The text for the axis or plot title.

See Also

labellers, labeller().

Examples

# The variables mentioned in the plotmath expression must be
# backquoted and referred to by their names.
p <- ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point()
p + facet_grid(vs ~ ., labeller = label_bquote(alpha ~ .(vs)))
p + facet_grid(. ~ vs, labeller = label_bquote(cols = .(vs) ^ .(vs)))
p + facet_grid(. ~ vs + am, labeller = label_bquote(cols = .(am) ^ .(vs)))
Examples

```r
p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
p + labs(title = "New plot title")
p + labs(x = "New x label")
p + xlab("New x label")
p + ylab("New y label")
p + ggtitle("New plot title")

# This should work independently of other functions that modify the
# the scale names
p + ylab("New y label") + ylim(2, 4)
p + ylim(2, 4) + ylab("New y label")

# The labs function also modifies legend labels
p <- ggplot(mtcars, aes(mpg, wt, colour = cyl)) + geom_point()
p + labs(colour = "Cylinders")

# Can also pass in a list, if that is more convenient
p + labs(list(title = "Title", x = "X", y = "Y"))
```

---

**last_plot**

Retrieves the last plot to be modified or created.

**Description**

Retrieve the last plot to be modified or created.

**Usage**

```r
last_plot()
```

**See Also**

`ggsave`

---

**layer**

Create a new layer

**Description**

Create a new layer

**Usage**

```r
layer(geom = NULL, stat = NULL, data = NULL, mapping = NULL,
      position = NULL, params = list(), inherit.aes = TRUE, subset = NULL,
      show.legend = NA)
```
## Arguments

- **geom, stat, position**
  Geom, stat and position adjustment to use in this layer. Can either be the name of a ggproto object, or the object itself.

- **data**
  A data frame. If specified, overrides the default data frame defined at the top level of the plot.

- **mapping**
  Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.

- **params**
  Additional parameters to the `geom` and `stat`.

- **inherit.aes**
  If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

- **subset**
  DEPRECATED. An older way of subsetting the dataset used in a layer.

- **show.legend**
  logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.

## Examples

```r
# geom calls are just a short cut for layer
ggplot(mpg, aes(displ, hwy)) + geom_point()
# shortcut for
ggplot(mpg, aes(displ, hwy)) +
  layer(geom = "point", stat = "identity", position = "identity",
        params = list(na.rm = FALSE))
```

### lims

Convenience functions to set the axis limits.

## Description

Observations not in this range will be dropped completely and not passed to any other layers. If a NA value is substituted for one of the limits that limit is automatically calculated.

## Usage

- `lims(...)`
- `xlim(...)`
- `ylim(...)`

## Arguments

... If numeric, will create a continuous scale, if factor or character, will create a discrete scale. For `lims`, every argument must be named.
See Also

For changing x or y axis limits without dropping data observations, see `coord_cartesian`.

Examples

```r
# xlim
xlim(15, 20)
xlim(20, 15)
xlim(c(10, 20))
xlim("a", "b", "c")

ggplot(mtcars, aes(mpg, wt)) +
  geom_point() +
  xlim(15, 20)

# with automatic lower limit
ggplot(mtcars, aes(mpg, wt)) +
  geom_point() +
  xlim(NA, 20)

# Change both xlim and ylim
ggplot(mtcars, aes(mpg, wt)) +
  geom_point() +
  lims(x = c(10, 20), y = c(3, 5))
```

---

`luv_colours` colors() in Luv space.

Description

All built-in `colors()` translated into Luv colour space.

Usage

`luv_colours`

Format

A data frame with 657 observations and 4 variables:

- L,u,vPosition in Luv colour space
- colColour name
Create a data frame of map data.

**Description**

Create a data frame of map data.

**Usage**

```r
map_data(map, region = ".", exact = FALSE, ...)
```

**Arguments**

- `map` name of map provided by the `maps` package. These include `county`, `france`, `italy`, `nz`, `state`, `usa`, `world`, `world2`.
- `region` name of subregions to include. Defaults to . which includes all subregion. See documentation for `map` for more details.
- `exact` should the region be treated as a regular expression (FALSE) or as a fixed string (TRUE).
- `...` all other arguments passed on to `map`

**Examples**

```r
if (require("maps")) {
  states <- map_data("state")
  arrests <- USArrests
  names(arrests) <- tolower(names(arrests))
  arrests$region <- tolower(rownames(USArrests))

  choro <- merge(states, arrests, sort = FALSE, by = "region")
  choro <- choro[order(choro$order),]
  ggplot(choro, aes(long, lat)) +
    geom_polygon(aes(group = group, fill = assault)) +
    coord_map("albers", at0 = 45.5, lat1 = 29.5)

  ggplot(choro, aes(long, lat)) +
    geom_polygon(aes(group = group, fill = assault / murder)) +
    coord_map("albers", at0 = 45.5, lat1 = 29.5)
}
```
margin
  Define margins.

Description
This is a convenience function that creates a grid unit object of the correct length to use for setting margins.

Usage
```
margin(t = 0, r = 0, b = 0, l = 0, unit = "pt")
```

Arguments
- `t`, `b`, `r`, `l` Dimensions of each margin. (To remember order, think trouble).
- `unit` Default units of dimensions. Defaults to "pt" so it can be most easily scaled with the text.

Examples
```
margin(4)
margin(4, 2)
margin(4, 3, 2, 1)
```

mean_se
  Calculate mean and standard errors on either side.

Description
Calculate mean and standard errors on either side.

Usage
```
mean_se(x, mult = 1)
```

Arguments
- `x` numeric vector
- `mult` number of multiples of standard error

See Also
  for use with stat_summary
Description

Demographic information of midwest counties

Usage

midwest

Format

A data frame with 437 rows and 28 variables

- PID
- county
- state
- area
- poptotal. Total population
- popdensity. Population density
- popwhite. Number of whites.
- popblack. Number of blacks.
- popamerindan. Number of American Indians.
- popasian. Number of Asians.
- popother. Number of other races.
- perwhite. Percent white.
- perblack. Percent black.
- percamerindan. Percent American Indian.
- perasian. Percent Asian.
- perother. Percent other races.
- popadults. Number of adults.
- perchsd.
- percollege. Percent college educated.
- perprof. Percent profession.
- poppovertyknown.
- percpovertyknown
- perchbelowpoverty
- perchchildbelowpoverty
- percadultpoverty
- percelderlypoverty
- inmetro. In a metro area.
- category’
**mpg**

*Fuel economy data from 1999 and 2008 for 38 popular models of car*

**Description**

This dataset contains a subset of the fuel economy data that the EPA makes available on http://fueleconomy.gov. It contains only models which had a new release every year between 1999 and 2008 - this was used as a proxy for the popularity of the car.

**Usage**

mpg

**Format**

A data frame with 234 rows and 11 variables

- manufacturer.
- model.
- displ. engine displacement, in litres
- year.
- cyl. number of cylinders
- trans. type of transmission
- drv. f = front-wheel drive, r = rear wheel drive, 4 = 4wd
- cty. city miles per gallon
- hwy. highway miles per gallon
- fl.
- class.

---

**msleep**

*An updated and expanded version of the mammals sleep dataset.*

**Description**

This is an updated and expanded version of the mammals sleep dataset. Updated sleep times and weights were taken from V. M. Savage and G. B. West. A quantitative, theoretical framework for understanding mammalian sleep. Proceedings of the National Academy of Sciences, 104 (3):1051-1056, 2007.

**Usage**

msleep
Format

A data frame with 83 rows and 11 variables

- name. common name
- genus.
- vore. carnivore, omnivore or herbivore?
- order.
- conservation. the conservation status of the animal
- sleep\_total. total amount of sleep, in hours
- sleep\_rem. rem sleep, in hours
- sleep\_cycle. length of sleep cycle, in hours
- awake. amount of time spent awake, in hours
- brainwt. brain weight in kilograms
- bodywt. body weight in kilograms

Details

Additional variables order, conservation status and vore were added from wikipedia.

position_dodge

Adjust position by dodging overlaps to the side.

Description

Adjust position by dodging overlaps to the side.

Usage

position_dodge(width = NULL)

Arguments

width        Dodging width, when different to the width of the individual elements. This is useful when you want to align narrow geoms with wider geoms. See the examples for a use case.

See Also

Other position adjustments: position_fill, position_identity, position_jitterdodge, position_jitter, position_nudge
Examples

ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) +
g geom_bar(position = "dodge")

ggplot(diamonds, aes(price, fill = cut)) +
g geom_histogram(position="dodge")
# see ?geom_boxplot and ?geom_bar for more examples

# To dodge items with different widths, you need to be explicit
df <- data.frame(x=c("a","a","b","b"), y=2:5, g = rep(1:2, 2))
p <- ggplot(df, aes(x, y, group = g)) +
g geom_bar(
    stat = "identity", position = "dodge",
    fill = "grey50", colour = "black"
)

# A line range has no width:
p + geom_linerange(aes(ymin = y-1, ymax = y+1), position = "dodge")
# You need to explicitly specify the width for dodging
p + geom_linerange(aes(ymin = y-1, ymax = y+1),
    position = position_dodge(width = 0.9))

# Similarly with error bars:
p + geom_errorbar(aes(ymin = y-1, ymax = y+1), width = 0.2,
    position = "dodge")
p + geom_errorbar(aes(ymin = y-1, ymax = y+1, width = 0.2),
    position = position_dodge(width = 0.9)))

position_fill

Stack overlapping objects on top of one another.

Description

position_fill additionally standardises each stack to have unit height.

Usage

position_fill()

position_stack()

See Also

See geom_bar and geom_area for more examples.

Other position adjustments: position_dodge, position_identity, position_jitterdodge, position_jitter, position_nudge
Examples

# Stacking is the default behaviour for most area plots:
ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) + geom_bar()

# Fill makes it easier to compare proportions
ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) +
  geom_bar(position = "fill")

# To change stacking order, use factor() to change order of levels
mtcars$vs <- factor(mtcars$vs, levels = c(1,0))
ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) + geom_bar()

ggplot(diamonds, aes(price, fill = cut)) +
  geom_histogram(binwidth = 500)
# When used with a histogram, position_fill creates a conditional density
# estimate
ggplot(diamonds, aes(price, fill = cut)) +
  geom_histogram(binwidth = 500, position = "fill")

# Stacking is also useful for time series
data.set <- data.frame(
  Time = c(rep(1, 4),rep(2, 4), rep(3, 4), rep(4, 4)),
  Type = rep(c("a", "b", "c", "d"), 4),
  Value = rpois(16, 10)
)

ggplot(data.set, aes(Time, Value)) + geom_area(aes(fill = Type))

# If you want to stack lines, you need to say so:
ggplot(data.set, aes(Time, Value)) + geom_line(aes(colour = Type))
ggplot(data.set, aes(Time, Value)) +
  geom_line(position = "stack", aes(colour = Type))

# But realise that this makes it *much* harder to compare individual
# trends

position_identity Don’t adjust position

Description

Don’t adjust position

Usage

position_identity()

See Also

Other position adjustments: position_dodge, position_fill, position_jitterdodge, position_jitter, position_nudge
position_jitter

Jitter points to avoid overplotting.

Description

Jitter points to avoid overplotting.

Usage

position_jitter(width = NULL, height = NULL)

Arguments

width, height Amount of vertical and horizontal jitter. The jitter is added in both positive and negative directions, so the total spread is twice the value specified here. If omitted, defaults to 40% of the resolution of the data: this means the jitter values will occupy 80% of the implied bins. Categorical data is aligned on the integers, so a width or height of 0.5 will spread the data so it’s not possible to see the distinction between the categories.

See Also

Other position adjustments: position_dodge, position_fill, position_identity, position_jitterdodge, position_nudge

Examples

ggplot(mtcars, aes(am, vs)) + geom_point()

# Default amount of jittering will generally be too much for
# small datasets:
ggplot(mtcars, aes(am, vs)) + geom_jitter()

# Two ways to override
ggplot(mtcars, aes(am, vs)) +
  geom_jitter(width = 0.1, height = 0.1)
ggplot(mtcars, aes(am, vs)) +
  geom_jitter(position = position_jitter(width = 0.1, height = 0.1))

# The default works better for large datasets, where it will
# take up as much space as a boxplot or a bar
ggplot(mpg, aes(class, hwy)) +
  geom_jitter() +
  geom_boxplot()
position_jitterdodge  
Adjust position by simultaneously dodging and jittering

Description
This is primarily used for aligning points generated through `geom_point()` with dodged boxplots (e.g., a `geom_boxplot()` with a fill aesthetic supplied).

Usage
```r
position_jitterdodge(jitter.width = NULL, jitter.height = 0,
                      dodge.width = 0.75)
```

Arguments
- `jitter.width`: degree of jitter in x direction. Defaults to 40% of the resolution of the data.
- `jitter.height`: degree of jitter in y direction. Defaults to 0.
- `dodge.width`: the amount to dodge in the x direction. Defaults to 0.75, the default `position_dodge()` width.

See Also
Other position adjustments: `position_dodge`, `position_fill`, `position_identity`, `position_jitter`, `position_nudge`

Examples
```r
dsub <- diamonds[ sample(nrow(diamonds), 1000), ]
ggplot(dsub, aes(x = cut, y = carat, fill = clarity)) +
  geom_boxplot(outlier.size = 0) +
  geom_point(pch = 21, position = position_jitterdodge())
```

position_nudge  
Nudge points.

Description
This is useful if you want to nudge labels a little ways from their points.

Usage
```r
position_nudge(x = 0, y = 0)
```

Arguments
- `x, y`: Amount of vertical and horizontal distance to move.
See Also

Other position adjustments: `position_dodge`, `position_fill`, `position_identity`, `position_jitterdodge`, `position_jitter`
**Usage**

```r
## S3 method for class 'ggplot'
print(x, newpage = is.null(vp), vp = NULL, ...)

## S3 method for class 'ggplot'
plot(x, newpage = is.null(vp), vp = NULL, ...)
```

**Arguments**

- `x` plot to display
- `newpage` draw new (empty) page first?
- `vp` viewport to draw plot in
- `...` other arguments not used by this method

**Value**

Invisibly returns the result of `ggplot_build`, which is a list with components that contain the plot itself, the data, information about the scales, panels etc.

---

**Description**

If a ggproto object has a `print` method, this will call that method. Otherwise, it will print out the members of the object, and optionally, the members of the inherited objects.

**Usage**

```r
## S3 method for class 'ggproto'
print(x, ..., flat = TRUE)
```

**Arguments**

- `x` A ggproto object to print.
- `...` If the ggproto object has a `print` method, further arguments will be passed to it. Otherwise, these arguments are unused.
- `flat` If TRUE (the default), show a flattened list of all local and inherited members. If FALSE, show the inheritance hierarchy.
qplot

Description

qplot is the basic plotting function in the ggplot2 package, designed to be familiar if you’re used to base plot(). It’s a convenient wrapper for creating a number of different types of plots using a consistent calling scheme.

Usage

qplot(x, y = NULL, ..., data, facets = NULL, margins = FALSE,
      geom = "auto", xlim = c(NA, NA), ylim = c(NA, NA), log = "",
      main = NULL, xlab = deparse(substitute(x)),
      ylab = deparse(substitute(y)), asp = NA, stat = NULL, position = NULL)

qquickplot(x, y = NULL, ..., data, facets = NULL, margins = FALSE,
          geom = "auto", xlim = c(NA, NA), ylim = c(NA, NA), log = "",
          main = NULL, xlab = deparse(substitute(x)),
          ylab = deparse(substitute(y)), asp = NA, stat = NULL, position = NULL)

Arguments

x, y, ... Aesthetics passed into each layer
data Data frame to use (optional). If not specified, will create one, extracting vectors from the current environment.
facets faceting formula to use. Picks facet_wrap or facet_grid depending on whether the formula is one- or two-sided
margins See facet_grid: display marginal facets?
geom Character vector specifying geom(s) to draw. Defaults to "point" if x and y are specified, and "histogram" if only x is specified.
xlim, ylim X and y axis limits
log Which variables to log transform ("x", "y", or "xy")
main, xlab, ylab Character vector (or expression) giving plot title, x axis label, and y axis label respectively.
asp The y/x aspect ratio
stat, position DEPRECATED.

Examples

# Use data from data.frame
qplot(mpg, wt, data = mtcars)
qplot(mpg, wt, data = mtcars, colour = cyl)
qplot(mpg, wt, data = mtcars, size = cyl)
```r
qplot(mpg, wt, data = mtcars, facets = vs ~ am)

qplot(1:10, rnorm(10), colour = runif(10))
qplot(1:10, letters[1:10])
mod <- lm(mpg ~ wt, data = mtcars)
qplot(resid(mod), fitted(mod))

f <- function() {
  a <- 1:10
  b <- a ^ 2
  qplot(a, b)
}
f()

# To set aesthetics, wrap in I()
qplot(mpg, wt, data = mtcars, colour = I("red"))

# qplot will attempt to guess what geom you want depending on the input
# both x and y supplied = scatterplot
qplot(mpg, wt, data = mtcars)
# just x supplied = histogram
qplot(mpg, data = mtcars)
# just y supplied = scatterplot, with x = seq_along(y)
qplot(y = mpg, data = mtcars)

# Use different geoms
qplot(mpg, wt, data = mtcars, geom = "path")
qplot(factor(cyl), wt, data = mtcars, geom = c("boxplot", "jitter"))
qplot(mpg, data = mtcars, geom = "dotplot")
```

---

**rel**

*Relative sizing for theme elements*

**Description**

Relative sizing for theme elements

**Usage**

`rel(x)`

**Arguments**

- `x` A number representing the relative size
Examples

df <- data.frame(x = 1:3, y = 1:3)
ggplot(df, aes(x, y)) +
  geom_point() +
  theme(axis.title.x = element_text(size = rel(2.5)))

resolution

Description

The resolution 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

scale_alpha

Description

scale_alpha is an alias for scale_alpha_continuous since that is the most common use of alpha, and it saves a bit of typing.
scale_colour_brewer

Usage

scale_alpha(..., range = c(0.1, 1))
scale_alpha_continuous(..., range = c(0.1, 1))
scale_alpha_discrete(..., range = c(0.1, 1))

Arguments

... Other arguments passed on to continuous_scale or discrete_scale as appropriate, to control name, limits, breaks, labels and so forth.
range range of output alpha values. Should lie between 0 and 1.

Examples

(p <- ggplot(mtcars, aes(mpg, cyl)) +
  geom_point(aes(alpha = cyl)))
p + scale_alpha("cylinders")
p + scale_alpha("number\of\ncylinders")
p + scale_alpha(range = c(0.4, 0.8))

(p <- ggplot(mtcars, aes(mpg, cyl)) +
  geom_point(aes(alpha = factor(cyl))))
p + scale_alpha_discrete(range = c(0.4, 0.8))

---

scale_colour_brewer  Sequential, diverging and qualitative colour scales from colorbrewer.org

Description

ColorBrewer provides sequential, diverging and qualitative colour schemes which are particularly suited and tested to display discrete values (levels of a factor) on a map. ggplot2 can use those colours in discrete scales. It also allows to smoothly interpolate 6 colours from any palette to a continuous scale (6 colours per palette gives nice gradients; more results in more saturated colours which do not look as good). However, the original colour schemes (particularly the qualitative ones) were not intended for this and the perceptual result is left to the appreciation of the user. See http://colorbrewer2.org for more information.

Usage

scale_colour_brewer(..., type = "seq", palette = 1, direction = 1)
scale_fill_brewer(..., type = "seq", palette = 1, direction = 1)
scale_colour_distiller(..., type = "seq", palette = 1, direction = -1,
scale_colour_brewer

values = NULL, space = "Lab", na.value = "grey50",
guide = "colourbar")

data_fill_distiller(..., type = "seq", palette = 1, direction = -1,
values = NULL, space = "Lab", na.value = "grey50",
guide = "colourbar")

Arguments

... Other arguments passed on to discrete_scale to control name, limits, breaks, labels and so forth.
type One of seq (sequential), div (diverging) or qual (qualitative)
palette If a string, will use that named palette. If a number, will index into the list of palettes of appropriate type
direction Sets the order of colors in the scale. If 1, the default, colors are as output by brewer.pal. If -1, the order of colors is reversed.
values if colours should not be evenly positioned along the gradient this vector gives the position (between 0 and 1) for each colour in the colours vector. See rescale for a convience function to map an arbitrary range to between 0 and 1.
space colour space in which to calculate gradient. Must be "Lab" - other values are deprecated.
na.value Colour to use for missing values
guide Type of legend. Use "colourbar" for continuous colour bar, or "legend" for discrete colour legend.

Palettes

The following palettes are available for use with these scales:

Diverging BrBG, PiYG, PRGn, PuOr, RdBu, RdGy, RdYlBu, RdYlGn, Spectral
Qualitative Accent, Dark2, Paired, Pastel1, Pastel2, Set1, Set2, Set3
Sequential Blues, BuGn, BuPu, GnBu, Greens, Greens, Oranges, OrRd, PuBu, PuBuGn, PuRd, Purples, RdPu, Reds, YlGn, YlGnBu, YlOrBr, YlOrRd

See Also

Other colour scales: scale_colour_gradient, scale_colour_grey, scale_colour_hue

Examples

dsamp <- diamonds[sample(nrow(diamonds), 1000), ]
(d <- ggplot(dsamp, aes(carat, price)) +
 geom_point(aes(colour = clarity)))

# Change scale label
d + scale_colour_brewer()
d + scale_colour_brewer("Diamond\nclarity")
# Select brewer palette to use, see ?scales::brewer_pal for more details
d + scale_colour_brewer(palette = "Greens")
d + scale_colour_brewer(palette = "Set1")

# scale_fill_brewer works just the same as
# scale_colour_brewer but for fill colours
p <- ggplot(diamonds, aes(x = price, fill = cut)) +
  geom_histogram(position = "dodge", binwidth = 1000)
p + scale_fill_brewer()
# the order of colour can be reversed
p + scale_fill_brewer(direction = -1)
# the brewer scales look better on a darker background
p + scale_fill_brewer(direction = -1) + theme_dark()

# Use distiller variant with continous data
v <- ggplot(faithful) +
  geom_tile(aes(waiting, eruptions, fill = density))
v + scale_fill_distiller()
v + scale_fill_distiller(palette = "Spectral")

---

## scale_colour_gradient

**Smooth gradient between two colours**

---

### Description

scale_*_gradient creates a two colour gradient (low-high), scale_*_gradient2 creates a diverging colour gradient (low-mid-high), scale_*_gradientn creates a n-colour gradient.

### Usage

```
scale_colour_gradient(..., low = "#132B43", high = "#56B1F7",
  space = "Lab", na.value = "grey50", guide = "colourbar")

scale_fill_gradient(..., low = "#132B43", high = "#56B1F7", space = "Lab",
  na.value = "grey50", guide = "colourbar")

scale_colour_gradient2(..., low = muted("red"), mid = "white",
  high = muted("blue"), midpoint = 0, space = "Lab",
  na.value = "grey50", guide = "colourbar")

scale_fill_gradient2(..., low = muted("red"), mid = "white",
  high = muted("blue"), midpoint = 0, space = "Lab",
  na.value = "grey50", guide = "colourbar")

scale_colour_gradientn(..., colours, values = NULL, space = "Lab",
  na.value = "grey50", guide = "colourbar", colors)
```
scale_fill_gradientn(..., colours, values = NULL, space = "Lab",
na.value = "grey50", guide = "colourbar", colors)

Arguments

... Other arguments passed on to discrete_scale to control name, limits, breaks, labels and so forth.

low, high Colours for low and high ends of the gradient.
space colour space in which to calculate gradient. Must be "Lab" - other values are deprecated.

na.value Colour to use for missing values

guide Type of legend. Use "colourbar" for continuous colour bar, or "legend" for discrete colour legend.

mid colour for mid point

midpoint The midpoint (in data value) of the diverging scale. Defaults to 0.

colours, colors Vector of colours to use for n-colour gradient.

values if colours should not be evenly positioned along the gradient this vector gives the position (between 0 and 1) for each colour in the colours vector. See rescale for a convenience function to map an arbitrary range to between 0 and 1.

Details

Default colours are generated with munsell and mns1(c("2.5PB 2/4", "2.5PB 7/10"). Generally, for continuous colour scales you want to keep hue constant, but vary chroma and luminance. The munsell package makes this easy to do using the Munsell colour system.

See Also

seq_gradient_pal for details on underlying palette

Other colour scales: scale_colour_brewer, scale_colour-grey, scale_colour_hue

Examples

df <- data.frame(
  x = runif(100),
  y = runif(100),
  z1 = rnorm(100),
  z2 = abs(rnorm(100))
)

# Default colour scale colours from light blue to dark blue
ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z2))

# For diverging colour scales use gradient2
ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z2))
```r
geom_point(aes(colour = z1)) +
  scale_colour_gradient2()

# Use your own colour scale with gradientn
ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z1)) +
  scale_colour_gradientn(colours = terrain.colors(10))

# Equivalent fill scales do the same job for the fill aesthetic
ggplot(faithful, aes(waiting, eruptions)) +
  geom_raster(aes(fill = density)) +
  scale_fill_gradientn(colours = terrain.colors(10))

# Adjust colour choices with low and high
ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z2)) +
  scale_colour_gradientn(low = "white", high = "black")
# Avoid red-green colour contrasts because ~10% of men have difficulty
# seeing them
```

---

**scale_colour_grey**  
*Sequential grey colour scale.*

**Description**

Based on `gray.colors`

**Usage**

```r
scale_colour_grey(..., start = 0.2, end = 0.8, na.value = "red")

scale_fill_grey(..., start = 0.2, end = 0.8, na.value = "red")
```

**Arguments**

- `...`: Other arguments passed on to `discrete_scale` to control name, limits, breaks, labels and so forth.
- `start`: Gray value at low end of palette
- `end`: Gray value at high end of palette
- `na.value`: Colour to use for missing values

**See Also**

Other colour scales: `scale_colour_brewer, scale_colour_gradient, scale_colour_hue`
Examples

```r
p <- ggplot(mtcars, aes(mpg, wt)) + geom_point(aes(colour = factor(cyl)))
p + scale_colour_grey()
p + scale_colour_grey(end = 0)

# You may want to turn off the pale grey background with this scale
p + scale_colour_grey() + theme_bw()

# Colour of missing values is controlled with na.value:
miss <- factor(sample(c(NA, 1:5), nrow(mtcars), replace = TRUE))
ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(colour = miss)) +
  scale_colour_grey()
ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(colour = miss)) +
  scale_colour_grey(na.value = "green")
```

---

`scale_colour_hue`  
*Qualitative colour scale with evenly spaced hues.*

Description

Qualitative colour scale with evenly spaced hues.

Usage

```r
scale_colour_hue(..., h = c(0, 360) + 15, c = 100, l = 65, h.start = 0,
  direction = 1, na.value = "grey50")
```

```r
scale_fill_hue(..., h = c(0, 360) + 15, c = 100, l = 65, h.start = 0,
  direction = 1, na.value = "grey50")
```

Arguments

- `...`: Other arguments passed on to `discrete_scale` to control name, limits, breaks, labels and so forth.
- `h`: range of hues to use, in [0, 360]
- `c`: chroma (intensity of colour), maximum value varies depending on combination of hue and luminance.
- `l`: luminance (lightness), in [0, 100]
- `h.start`: hue to start at
- `direction`: direction to travel around the colour wheel, 1 = clockwise, -1 = counter-clockwise
- `na.value`: Colour to use for missing values

See Also

Other colour scales: `scale_colour_brewer`, `scale_colour_gradient`, `scale_colour_grey`
Examples

dsamp <- diamonds[sample(nrow(diamonds), 1000), ]
(d <- ggplot(dsamp, aes(carat, price)) + geom_point(aes(colour = clarity)))

# Change scale label
d + scale_colour_hue()
# Adjust luminosity and chroma
# Change range of hues used
# Vary opacity
# (only works with pdf, quartz and cairo devices)
# Colour of missing values is controlled with na.value:

# Colour of missing values is controlled with na.value:

scale_continuous

Description

`scale_x_continuous` and `scale_y_continuous` are the key functions. The others, `scale_x_log10`, `scale_y_sqrt` etc, are aliases that set the `trans` argument to commonly used transformations.

Usage

```r
scale_x_continuous(name = waiver(), breaks = waiver(),
                   minor_breaks = waiver(), labels = waiver(), limits = NULL,
```
```r
expand = waiver(), oob = censor, na.value = NA_real_,
        trans = "identity")

scale_y_continuous(name = waiver(), breaks = waiver(),
         minor_breaks = waiver(), labels = waiver(), limits = NULL,
        expand = waiver(), oob = censor, na.value = NA_real_,
        trans = "identity")

scale_x_log10(...)  
scale_y_log10(...)  
scale_x_reverse(...)  
scale_y_reverse(...)  
scale_x_sqrt(...)  
scale_y_sqrt(...)  

Arguments

name The name of the scale. Used as axis or legend title. If NULL, the default, the
        name of the scale is taken from the first mapping used for that aesthetic.

breaks One of:
        • NULL for no breaks
        • waiver() for the default breaks computed by the transformation object
        • A numeric vector of positions
        • A function that takes the limits as input and returns breaks as output

minor_breaks One of:
        • NULL for no minor breaks
        • waiver() for the default breaks (one minor break between each major
         break)
        • A numeric vector of positions
        • A function that given the limits returns a vector of minor breaks.

labels One of:
        • NULL for no labels
        • waiver() for the default labels computed by the transformation object
        • A character vector giving labels (must be same length as breaks)
        • A function that takes the breaks as input and returns labels as output

limits A numeric vector of length two providing limits of the scale. Use NA to refer to
        the existing minimum or maximum.

expand A numeric vector of length two giving multiplicative and additive expansion
        constants. These constants ensure that the data is placed some distance away
        from the axes. The defaults are c(0.05, 0) for continuous variables, and
        c(0, 0.6) for discrete variables.
```
Function that handles limits outside of the scale limits (out of bounds). The default replaces out of bounds values with NA.

Missing values will be replaced with this value.

Either the name of a transformation object, or the object itself. Built-in transformations include "asn", "atanh", "boxcox", "exp", "identity", "log", "log10", "log1p", "log2", "logit", "probability", "probit", "reciprocal", "reverse" and "sqrt". A transformation object bundles together a transform, its inverse, and methods for generating breaks and labels. Transformation objects are defined in the scales package, and are called name_trans, e.g. boxcox_trans. You can create your own transformation with trans_new.

Other arguments passed on to scale_{x|y}_continuous

See Also

scale_date for date/time position scales.

Examples

if (require(ggplot2movies)) {
  m <- ggplot(subset(movies, votes > 1000), aes(rating, votes)) +
    geom_point(na.rm = TRUE)
  m

  # Manipulating the default position scales lets you:
  # * change the axis labels
  m + scale_y_continuous("number of votes")
  m + scale_y_continuous(quote(votes ^ alpha))

  # * modify the axis limits
  m + scale_y_continuous(limits = c(0, 5000))
  m + scale_y_continuous(limits = c(1000, 10000))
  m + scale_x_continuous(limits = c(7, 8))

  # you can also use the short hand functions xlim and ylim
  m + ylim(0, 5000)
  m + ylim(1000, 10000)
  m + xlim(7, 8)

  # * choose where the ticks appear
  m + scale_x_continuous(breaks = 1:10)
  m + scale_x_continuous(breaks = c(1,3,7,9))

  # * manually label the ticks
  m + scale_x_continuous(breaks = c(2,5,8), labels = c("two", "five", "eight"))
  m + scale_x_continuous(breaks = c(2,5,8), labels = c("horrible", "ok", "awesome"))
  m + scale_x_continuous(breaks = c(2,5,8), labels = expression(Alpha, Beta, Omega))

  # There are a few built in transformation that you can use:
  m + scale_y_log10()
m + scale_y_sqrt()

m + scale_y_reverse()

# You can also create your own and supply them to the trans argument.
# See ?scales::trans_new

# You can control the formatting of the labels with the formatter argument. Some common formats are built into the scales package:
df <- data.frame(
  x = rnorm(10) * 100000,
  y = seq(0, 1, length.out = 10)
)
p <- ggplot(df, aes(x, y)) + geom_point()
p + scale_y_continuous(labels = scales::percent)
p + scale_y_continuous(labels = scales::dollar)
p + scale_x_continuous(labels = scales::comma)

# Other shortcut functions
ggplot(movies, aes(rating, votes)) +
  geom_point() +
  ylim(1e4, 5e4)
# * axis labels
ggplot(movies, aes(rating, votes)) +
  geom_point() +
  labs(x = "My x axis", y = "My y axis")
# * log scaling
ggplot(movies, aes(rating, votes)) +
  geom_point() +
  scale_x_log10() +
  scale_y_log10()
}

scale_date

Position scale, date & date times

Description

Use scale_*_date with Date variables, and scale_*_datetime with POSIXct variables.

Usage

scale_x_date(name = waiver(), breaks = waiver(), date_breaks = waiver(),
  labels = waiver(), date_labels = waiver(), minor_breaks = waiver(),
  date_minor_breaks = waiver(), limits = NULL, expand = waiver())

scale_y_date(name = waiver(), breaks = waiver(), date_breaks = waiver(),
  labels = waiver(), date_labels = waiver(), minor_breaks = waiver(),
  date_minor_breaks = waiver(), limits = NULL, expand = waiver())

scale_x_datetime(name = waiver(), breaks = waiver(),
  date_breaks = waiver(),
  labels = waiver(), date_labels = waiver(), minor_breaks = waiver(),
  date_minor_breaks = waiver(), limits = NULL, expand = waiver())
scale_date

```r
date_breaks = waiver(), labels = waiver(), date_labels = waiver(),
minor_breaks = waiver(), date_minor_breaks = waiver(), limits = NULL,
expand = waiver())

scale_y_datetime(name = waiver(), breaks = waiver(),
date_breaks = waiver(), labels = waiver(), date_labels = waiver(),
minor_breaks = waiver(), date_minor_breaks = waiver(), limits = NULL,
expand = waiver())
```

**Arguments**

- `name` The name of the scale. Used as axis or legend title. If `NULL`, the default, the name of the scale is taken from the first mapping used for that aesthetic.

- `breaks` One of:
  - `NULL` for no breaks
  - `waiver()` for the default breaks computed by the transformation object
  - A numeric vector of positions
  - A function that takes the limits as input and returns breaks as output

- `date_breaks` A string giving the distance between breaks like "2 weeks", or "10 years". If both breaks and date_breaks are specified, date_breaks wins.

- `labels` One of:
  - `NULL` for no labels
  - `waiver()` for the default labels computed by the transformation object
  - A character vector giving labels (must be same length as breaks)
  - A function that takes the breaks as input and returns labels as output

- `date_labels` A string giving the formatting specification for the labels. Codes are defined in `strftime`. If both labels and date_labels are specified, date_labels wins.

- `minor_breaks` One of:
  - `NULL` for no minor breaks
  - `waiver()` for the default breaks (one minor break between each major break)
  - A numeric vector of positions
  - A function that given the limits returns a vector of minor breaks.

- `date_minor_breaks` A string giving the distance between minor breaks like "2 weeks", or "10 years". If both minor_breaks and date_minor_breaks are specified, date_minor_breaks wins.

- `limits` A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum.

- `expand` A numeric vector of length two giving multiplicative and additive expansion constants. These constants ensure that the data is placed some distance away from the axes. The defaults are `c(0.05, 0)` for continuous variables, and `c(0, 0.6)` for discrete variables.
scale_identity

Use values without scaling.

See Also

scale_continuous for continuous position scales.

Examples

last_month <- Sys.Date() - 0:29
df <- data.frame(
  date = last_month,
  price = runif(30)
)
base <- ggplot(df, aes(date, price)) +
  geom_line()

# The date scale will attempt to pick sensible defaults for
# major and minor tick marks. Override with date_breaks, date_labels
# date_minor_breaks arguments.
base + scale_x_date(date_labels = "%b %d")
base + scale_x_date(date_breaks = "1 week", date_labels = "%W")
base + scale_x_date(date_minor_breaks = "1 day")

# Set limits
base + scale_x_date(limits = c(Sys.Date() - 7, NA))

scale_identity

Description

Use values without scaling.

Usage

scale_colour_identity(..., guide = "none")

scale_fill_identity(..., guide = "none")

scale_shape_identity(..., guide = "none")

scale_linetype_identity(..., guide = "none")

scale_alpha_identity(..., guide = "none")

scale_size_identity(..., guide = "none")

Arguments

... Other arguments passed on to discrete_scale or continuous_scale
guide Guide to use for this scale - defaults to "none".
Examples

```r
ggplot(luv_colours, aes(u, v)) +
  geom_point(aes(colour = col), size = 3) +
  scale_color_identity() +
  coord_equal()

df <- data.frame(
  x = 1:4,
  y = 1:4,
  colour = c("red", "green", "blue", "yellow")
)
ggplot(df, aes(x, y)) + geom_tile(aes(fill = colour))
ggplot(df, aes(x, y)) +
  geom_tile(aes(fill = colour)) +
  scale_fill_identity()

# To get a legend guide, specify guide = "legend"
ggplot(df, aes(x, y)) +
  geom_tile(aes(fill = colour)) +
  scale_fill_identity(guide = "legend")
# But you'll typically also need to supply breaks and labels:
ggplot(df, aes(x, y)) +
  geom_tile(aes(fill = colour)) +
  scale_fill_identity("trt", labels = letters[1:4], breaks = df$colour,
                  guide = "legend")

# cyl scaled to appropriate size
ggplot(mtcars, aes(mpg, wt)) + geom_point(aes(size = cyl))

# cyl used as point size
ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(size = cyl)) +
  scale_size_identity()
```

---

**scale_linetype**

*Scale for line patterns.*

**Description**

Default line types based on a set supplied by Richard Pearson, University of Manchester. Line types can not be mapped to continuous values.

**Usage**

```r
scale_linetype(..., na.value = "blank")

scale_linetype_continuous(...)  

scale_linetype_discrete(..., na.value = "blank")
```
**Arguments**

... common discrete scale parameters: name, breaks, labels, na.value, limits and guide. See `discrete_scale` for more details

na.value The linetype to use for NA values.

**Examples**

```r
base <- ggplot(economics_long, aes(date, value))
base + geom_line(aes(group = variable))
base + geom_line(aes(linetype = variable))

# See scale_manual for more flexibility
```

**Description**

Create your own discrete scale.

**Usage**

- `scale_colour_manual(..., values)`
- `scale_fill_manual(..., values)`
- `scale_size_manual(..., values)`
- `scale_shape_manual(..., values)`
- `scale_linetype_manual(..., values)`
- `scale_alpha_manual(..., values)`

**Arguments**

... common discrete scale parameters: name, breaks, labels, na.value, limits and guide. See `discrete_scale` for more details

values a set of aesthetic values to map data values to. If this is a named vector, then the values will be matched based on the names. If unnamed, values will be matched in order (usually alphabetical) with the limits of the scale. Any data values that don’t match will be given na.value.
Examples

```r
p <- ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(colour = factor(cyl)))

p + scale_colour_manual(values = c("red","blue", "green"))
p + scale_colour_manual(
  values = c("8" = "red","4" = "blue","6" = "green"))
# With rgb hex values
p + scale_colour_manual(values = c("#FF0000", "#0000FF", "#00FF00"))

# As with other scales you can use breaks to control the appearance
# of the legend
cols <- c("8" = "red","4" = "blue","6" = "darkgreen","10" = "orange")
p + scale_colour_manual(values = cols)
p + scale_colour_manual(values = cols, breaks = c("4", "6", "8"))
p + scale_colour_manual(values = cols, breaks = c("8", "6", "4"))
p + scale_colour_manual(values = cols, breaks = c("4", "6", "8"),
  labels = c("four", "six", "eight"))

# And limits to control the possible values of the scale
p + scale_colour_manual(values = cols, limits = c("4", "8"))
p + scale_colour_manual(values = cols, limits = c("4", "6", "8", "10"))

# Notice that the values are matched with limits, and not breaks
p + scale_colour_manual(limits = c(6, 8, 4), breaks = c(8, 4, 6),
  values = c("grey50", "grey80", "black"))
```

---

**scale_shape**  
*Scale for shapes, aka glyphs.*

**Description**
A continuous variable can not be mapped to shape.

**Usage**

```r
scale_shape(..., solid = TRUE)
```

```r
scale_shape_discrete(..., solid = TRUE)
```

```r
scale_shape_continuous(...)
```

**Arguments**

```
... common discrete scale parameters: name, breaks, labels, na.value, limits
and guide. See discrete_scale for more details

solid Are the shapes solid, TRUE, or hollow FALSE?
```
Examples

dsmall &lt;- diamonds[sample(nrow(diamonds), 100), ]

d &lt;= ggplot(dsmall, aes(carat, price)) + geom_point(aes(shape = cut))
d + scale_shape(solid = TRUE) # the default
d + scale_shape(solid = FALSE)
d + scale_shape(name = "Cut of diamond")
d + scale_shape(name = "Cut of 'diamond"")

# To change order of levels, change order of
# underlying factor
levels(dsmall$cut) &lt;- c("Fair", "Good", "Very Good", "Premium", "Ideal")

# Need to recreate plot to pick up new data
ggplot(dsmall, aes(price, carat)) + geom_point(aes(shape = cut))

# Or for short:
d %+% dsmall

scale_size

Scale size (area or radius).

Description

scale_size scales area, scale_radius scales radius. The size aesthetic is most commonly used for points and text, and humans perceive the area of points (not their radius), so this provides for optimal perception. scale_size_area ensures that a value of 0 is mapped to a size of 0.

Usage

scale_radius(name = waiver(), breaks = waiver(), labels = waiver(),
limits = NULL, range = c(1, 6), trans = "identity", guide = "legend")

scale_size(name = waiver(), breaks = waiver(), labels = waiver(),
limits = NULL, range = c(1, 6), trans = "identity", guide = "legend")

scale_size_area(..., max_size = 6)

Arguments

name The name of the scale. Used as axis or legend title. If NULL, the default, the name of the scale is taken from the first mapping used for that aesthetic.

breaks One of:

• NULL for no breaks
• waiver() for the default breaks computed by the transformation object
• A numeric vector of positions
• A function that takes the limits as input and returns breaks as output
labels  One of:

• NULL for no labels
• waiver() for the default labels computed by the transformation object
• A character vector giving labels (must be same length as breaks)
• A function that takes the breaks as input and returns labels as output

limits  A numeric vector of length two providing limits of the scale. Use NA to refer to
the existing minimum or maximum.

range  a numeric vector of length 2 that specifies the minimum and maximum size of
the plotting symbol after transformation.

trans  Either the name of a transformation object, or the object itself. Built-in trans-
formations include "asn", "atanh", "boxcox", "exp", "identity", "log", "log10",
"log1p", "log2", "logit", "probability", "probit", "reciprocal", "reverse" and "sqrt".
A transformation object bundles together a transform, it’s inverse, and methods
for generating breaks and labels. Transformation objects are defined in the scales
package, and are called name_trans, e.g., boxcox_trans. You can create your
own transformation with trans_new.

guide  Name of guide object, or object itself.

...  Other arguments passed on to continuous_scale to control name, limits, breaks,
labels and so forth.

max_size  Size of largest points.

See Also

scale_size_area if you want 0 values to be mapped to points with size 0.

Examples

```r
p <- ggplot(mpg, aes(displ, hwy, size = hwy)) +
  geom_point()
p
p + scale_size("Highway mpg")
p + scale_size(range = c(0, 10))

# If you want zero value to have zero size, use scale_size_area:
p + scale_size_area()

# This is most useful when size is a count
ggplot(mpg, aes(class, cyl)) +
  geom_count() +
  scale_size_area()

# If you want to map size to radius (usually bad idea), use scale_radius
p + scale_radius()
```
scale_x_discrete  Discrete position.

Description

You can use continuous positions even with a discrete position scale - this allows you (e.g.) to place labels between bars in a bar chart. Continuous positions are numeric values starting at one for the first level, and increasing by one for each level (i.e. the labels are placed at integer positions). This is what allows jittering to work.

Usage

scale_x_discrete(..., expand = waiver())
scale_y_discrete(..., expand = waiver())

Arguments

... common discrete scale parameters: name, breaks, labels, na.value, limits and guide. See discrete_scale for more details
expand a numeric vector of length two giving multiplicative and additive expansion constants. These constants ensure that the data is placed some distance away from the axes.

Examples

ggplot(diamonds, aes(cut)) + stat_bin()
ggplot(diamonds, aes(cut)) + geom_bar()

# The discrete position scale is added automatically whenever you
# have a discrete position.
(d <- ggplot(subset(diamonds, carat > 1), aes(cut, clarity)) +
  geom_jitter())

d + scale_x_discrete("Cut")
d + scale_x_discrete("Cut", labels = c("Fair" = "F","Good" = "G",
  "Very Good" = "VG","Perfect" = "P","Ideal" = "I"))

d + scale_y_discrete("Clarity")
d + scale_x_discrete("Cut") + scale_y_discrete("Clarity")

# Use limits to adjust the which levels (and in what order)
# are displayed
(d + scale_x_discrete(limits=c("Fair","Ideal")))

# you can also use the short hand functions xlim and ylim
d + xlim("Fair","Ideal", "Good")
seals

```r
d + ylim("I1", "IF")

# See ?reorder to reorder based on the values of another variable
ggplot(mpg, aes(manufacturer, cty)) + geom_point()

# Use abbreviate as a formatter to reduce long names
ggplot(mpg, aes(reorder(manufacturer, displ), cty)) + geom_point()

# Use abbreviate as a formatter to reduce long names
ggplot(mpg, aes(reorder(manufacturer, displ), cty)) + geom_point()

scale_x_discrete(labels = abbreviate)
```

---

**seals**

*Vector field of seal movements.*

**Description**


**Usage**

seals

**Format**

A data frame with 1155 rows and 4 variables

**References**

[http://www.stat.berkeley.edu/~brill/Papers/jspifinal.pdf](http://www.stat.berkeley.edu/~brill/Papers/jspifinal.pdf)

---

**stat_ecdf**

*Empirical Cumulative Density Function*

**Description**

Empirical Cumulative Density Function

**Usage**

```r
stat_ecdf(mapping = NULL, data = NULL, geom = "step",
  position = "identity", n = NULL, na.rm = FALSE, show.legend = NA,
  inherit.aes = TRUE, ...)
```
Arguments

- **mapping**
  The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- **data**
  A layer specific dataset - only needed if you want to override the plot defaults.
- **geom**
  The geometric object to use display the data
- **position**
  The position adjustment to use for overlapping points on this layer
- **n**
  if NULL, do not interpolate. If not NULL, this is the number of points to interpolate with.
- **na.rm**
  If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
- **show.legend**
  logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
- **inherit.aes**
  If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.
- **...**
  other arguments passed on to `layer`. This can include aesthetics whose values you want to set, not map. See `layer` for more details.

Computed variables

- **x** x in data
- **y** cumulative density corresponding x

Examples

```r
df <- data.frame(x = rnorm(1000))
ggplot(df, aes(x)) + stat_ecdf(geom = "step")

df <- data.frame(x = c(rnorm(100, 0, 3), rnorm(100, 0, 10)),
                 g = gl(2, 100))
ggplot(df, aes(x, colour = g)) + stat_ecdf()
```

---

**stat_ellipse**

*Plot data ellipses.*

Description

The method for calculating the ellipses has been modified from `car::ellipse` (Fox and Weisberg, 2011)
Usage

stat_ellipse(mapping = NULL, data = NULL, geom = "path",
position = "identity", type = "t", level = 0.95, segments = 51,
na.rm = FALSE, show.legend = NA, inherit.aes = TRUE, ...)

Arguments

mapping The aesthetic mapping, usually constructed with \texttt{aes} or \texttt{aes_string}. Only needs to be set at the layer level if you are overriding the plot defaults.
data A layer specific dataset - only needed if you want to override the plot defaults.
geom The geometric object to use display the data
position The position adjustment to use for overlapping points on this layer
type The type of ellipse. The default "t" assumes a multivariate t-distribution, and "norm" assumes a multivariate normal distribution. "euclid" draws a circle with the radius equal to level, representing the euclidean distance from the center. This ellipse probably won’t appear circular unless \texttt{coord_fixed()} is applied.
level The confidence level at which to draw an ellipse (default is 0.95), or, if \texttt{type=euclid}, the radius of the circle to be drawn.
segments The number of segments to be used in drawing the ellipse.
na.rm If \texttt{FALSE} (the default), removes missing values with a warning. If \texttt{TRUE} silently removes missing values.
show.legend logical. Should this layer be included in the legends? \texttt{NA}, the default, includes if any aesthetics are mapped. \texttt{FALSE} never includes, and \texttt{TRUE} always includes.
inherit.aes If \texttt{FALSE}, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. \texttt{borders}.
...
other arguments passed on to \texttt{layer}. This can include aesthetics whose values you want to set, not map. See \texttt{layer} for more details.

References


Examples

\begin{verbatim}
ggplot(faithful, aes(waiting, eruptions)) +
  geom_point() +
  stat_ellipse()

ggplot(faithful, aes(waiting, eruptions, color = eruptions > 3)) +
  geom_point() +
  stat_ellipse()
\end{verbatim}
stat_function

Superimpose a function.

Description
Superimpose a function.

Usage

```r
stat_function(mapping = NULL, data = NULL, geom = "path", 
position = "identity", fun, n = 101, args = list(), na.rm = FALSE, 
show.legend = NA, inherit.aes = TRUE, ...)
```

Arguments

- **mapping** The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- **data** A layer specific dataset - only needed if you want to override the plot defaults.
- **geom** The geometric object to use display the data
- **position** The position adjustment to use for overlapping points on this layer
- **fun** function to use
- **n** number of points to interpolate along
- **args** list of additional arguments to pass to `fun`
- **na.rm** If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
- **show.legend** logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
- **inherit.aes** If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.
- **...** other arguments passed on to `layer`. This can include aesthetics whose values you want to set, not map. See `layer` for more details.
**Aesthetics**

`stat_function` understands the following aesthetics (required aesthetics are in bold):

- `y`

**Computed variables**

- `x` x's along a grid
- `y` value of function evaluated at corresponding `x`

**Examples**

```r
set.seed(1492)
df <- data.frame(  
x = rnorm(100)
)
x <- df$x
base <- ggplot(df, aes(x)) + geom_density()
base + stat_function(fun = dnorm, colour = "red")
base + stat_function(fun = dnorm, colour = "red", arg = list(mean = 3))

# Plot functions without data
# Examples adapted from Kohske Takahashi

# Specify range of x-axis
ggplot(data.frame(x = c(0, 2)), aes(x)) +
  stat_function(fun = exp, geom = "line")

# Plot a normal curve
ggplot(data.frame(x = c(-5, 5)), aes(x)) + stat_function(fun = dnorm)

# To specify a different mean or sd, use the args parameter to supply new values
  gggplot(data.frame(x = c(-5, 5)), aes(x)) +
  stat_function(fun = dnorm, args = list(mean = 2, sd = .5))

# Two functions on the same plot
f <- gggplot(data.frame(x = c(0, 10)), aes(x))
f + stat_function(fun = sin, colour = "red") +
  stat_function(fun = cos, colour = "blue")

# Using a custom function
test <- function(x) (x^2 + x + 20)
f + stat_function(fun = test)
```

---

**stat_identity**

Identity statistic.

**Description**

The identity statistic leaves the data unchanged.
Usage

\[
\text{stat_identity}(\text{mapping} = \text{NULL}, \text{data} = \text{NULL}, \text{geom} = \text{"point"}, \text{position} = \text{"identity"}, \text{show.legend} = \text{NA}, \text{inherit.aes} = \text{TRUE}, \ldots)
\]

Arguments

- **mapping** The aesthetic mapping, usually constructed with \text{aes} or \text{aes_string}. Only needs to be set at the layer level if you are overriding the plot defaults.
- **data** A layer specific dataset - only needed if you want to override the plot defaults.
- **geom** The geometric object to use display the data
- **position** The position adjustment to use for overlapping points on this layer
- **show.legend** logical. Should this layer be included in the legends? \text{NA}, the default, includes if any aesthetics are mapped. \text{FALSE} never includes, and \text{TRUE} always includes.
- **inherit.aes** If \text{FALSE}, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. \text{borders}.
- **\ldots** other arguments passed on to \text{layer}. This can include aesthetics whose values you want to set, not map. See \text{layer} for more details.

Examples

\[
p <- \text{ggplot(mtcars, aes(wt, mpg))}
p + \text{stat_identity()}
\]

\[
\text{stat.qq} \quad \text{Calculation for quantile-quantile plot.}
\]

Description

Calculation for quantile-quantile plot.

Usage

\[
\text{stat.qq}(\text{mapping} = \text{NULL}, \text{data} = \text{NULL}, \text{geom} = \text{"point"}, \text{position} = \text{"identity"}, \text{distribution} = \text{stats::qnorm}, \text{dparams} = \text{list()}, \text{na.rm} = \text{FALSE}, \text{show.legend} = \text{NA}, \text{inherit.aes} = \text{TRUE}, \ldots)
\]

\[
\text{geom.qq}(\text{mapping} = \text{NULL}, \text{data} = \text{NULL}, \text{geom} = \text{"point"}, \text{position} = \text{"identity"}, \text{distribution} = \text{stats::qnorm}, \text{dparams} = \text{list()}, \text{na.rm} = \text{FALSE}, \text{show.legend} = \text{NA}, \text{inherit.aes} = \text{TRUE}, \ldots)
\]
Arguments

- **mapping**
  The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.

- **data**
  A layer specific dataset - only needed if you want to override the plot defaults.

- **geom**
  The geometric object to use display the data

- **position**
  The position adjustment to use for overlapping points on this layer

- **distribution**
  Distribution function to use, if `x` not specified

- **dparams**
  Additional parameters passed on to `distribution` function.

- **na.rm**
  If `FALSE` (the default), removes missing values with a warning. If `TRUE` silently removes missing values.

- **show.legend**
  Logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.

- **inherit.aes**
  If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

- **...**
  Other arguments passed on to `layer`. This can include aesthetics whose values you want to set, not map. See `layer` for more details.

Aesthetics

`stat_qq` understands the following aesthetics (required aesthetics are in bold):

- **sample**
- **x**
- **y**

Computed variables

- **sample** sample quantiles
- **theoretical** theoretical quantiles

Examples

```r
df <- data.frame(y = rt(200, df = 5))
p <- ggplot(df, aes(sample = y))
p + stat_qq()
p + geom_point(stat = "qq")

# Use fitdistr from MASS to estimate distribution params
params <- as.list(MASS::fitdistr(df$y, "t")$estimate)
ggplot(df, aes(sample = y)) +
  stat_qq(distribution = qt, dparams = params["df"])

# Using to explore the distribution of a variable
ggplot(mtcars) +
```
stat_summary_2d

bin and summarise in 2d (rectangle & hexagons)

Description

stat_summary_2d is a 2d variation of `stat_summary`. `stat_summary_hex` is a hexagonal variation of `stat_summary_2d`. The data are divided into bins defined by x and y, and then the values of z in each cell are summarised with `fun`.

Usage

```r
stat_summary_2d(mapping = NULL, data = NULL, geom = "tile",
position = "identity", bins = 30, binwidth = NULL, drop = TRUE,
fun = "mean", fun.args = list(), na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)
```

```r
stat_summary_hex(mapping = NULL, data = NULL, geom = "hex",
position = "identity", bins = 30, binwidth = NULL, drop = TRUE,
fun = "mean", fun.args = list(), na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)
```

Arguments

- `mapping` The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- `data` A layer specific dataset - only needed if you want to override the plot defaults.
- `geom` The geometric object to use display the data
- `position` The position adjustment to use for overlapping points on this layer
- `bins` numeric vector giving number of bins in both vertical and horizontal directions. Set to 30 by default.
- `binwidth` Numeric vector giving bin width in both vertical and horizontal directions. Overrides bins if both set.
- `drop` drop if the output of `fun` is `NA`.
- `fun` function for summary.
- `fun.args` A list of extra arguments to pass to `fun`.
- `na.rm` If `FALSE` (the default), removes missing values with a warning. If `TRUE` silently removes missing values.
- `show.legend` logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.
stat_summary_bin

inherit.aes

If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. borders.

... other arguments passed on to layer. This can include aesthetics whose values you want to set, not map. See layer for more details.

Aesthetics

• x: horizontal position
• y: vertical position
• z: value passed to the summary function

See Also

stat_summary_hex for hexagonal summarization. stat_bin2d for the binning options.

Examples

d <- ggplot(diamonds, aes(carat, depth, z = price))
d + stat_summary_2d()

# Specifying function
d + stat_summary_2d(fun = function(x) sum(x^2))
d + stat_summary_2d(fun = var)
d + stat_summary_2d(fun = "quantile", fun.args = list(probs = 0.1))

if (requireNamespace("hexbin")) {
  d + stat_summary_hex()
}

stat_summary_bin

Summarise y values at unique/binned x x.

Description

stat_summary operates on unique x; stat_summary_bin operators on binned x. They are more flexible versions of stat_bin: instead of just counting, the can compute any aggregate.

Usage

stat_summary_bin(mapping = NULL, data = NULL, geom = "pointrange",
  fun.data = NULL, fun.y = NULL, fun.ymax = NULL, fun.ymin = NULL,
  fun.args = list(), na.rm = FALSE, position = "identity",
  show.legend = NA, inherit.aes = TRUE, ...)

stat_summary(mapping = NULL, data = NULL, geom = "pointrange",
  fun.data = NULL, fun.y = NULL, fun.ymax = NULL, fun.ymin = NULL,
  fun.args = list(), na.rm = FALSE, position = "identity",
  show.legend = NA, inherit.aes = TRUE, ...)

Arguments

mapping: Set of aesthetic mappings created by `aes` or `aes_`. If specified and `inherit.aes = TRUE` (the default), is combined with the default mapping at the top level of the plot. You only need to supply `mapping` if there isn’t a mapping defined for the plot.

data: A data frame. If specified, overrides the default data frame defined at the top level of the plot.

geom: Use to override the default connection between `geom_histogram/geom_freqpoly` and `stat_bin`.

fun.data: A function that is given the complete data and should return a data frame with variables `ymin`, `y`, and `ymax`.

`fun.ymin`, `fun.y`, `fun.ymax`: Alternatively, supply three individual functions that are each passed a vector of x’s and should return a single number.

`fun.args`: Optional additional arguments passed on to the functions.

`na.rm`: If `FALSE` (the default), removes missing values with a warning. If `TRUE` silently removes missing values.

`position`: Position adjustment, either as a string, or the result of a call to a position adjustment function.

`show.legend`: Logical. Should this layer be included in the legends? `NA`, the default, includes if any aesthetics are mapped. `FALSE` never includes, and `TRUE` always includes.

`inherit.aes`: If `FALSE`, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.

...: other arguments passed on to `layer`. There are three types of arguments you can use here:

- Aesthetics: to set an aesthetic to a fixed value, like `color = "red"` or `size = 3`.
- Other arguments to the layer, for example you override the default `stat` associated with the layer.
- Other arguments passed on to the stat.

Aesthetics

`stat_summary` understands the following aesthetics (required aesthetics are in bold):

- `x`
- `y`

Summary functions

You can either supply summary functions individually (`fun.y`, `fun.ymax`, `fun.ymin`), or as a single function (`fun.data`):

`fun.data`: Complete summary function. Should take numeric vector as input and return data frame as output.
fun.ymin  ymin summary function (should take numeric vector and return single number)

fun.y  y summary function (should take numeric vector and return single number)

fun.ymax  ymax summary function (should take numeric vector and return single number)

A simple vector function is easiest to work with as you can return a single number, but is somewhat less flexible. If your summary function computes multiple values at once (e.g. ymin and ymax), use fun.data.

If no aggregation functions are supplied, will default to mean_se.

See Also

geom_errorbar, geom_pointrange, geom_linerange, geom_crossbar for geoms to display summarised data

Examples

d <- ggplot(mtcars, aes(cyl, mpg)) + geom_point()
d + stat_summary(fun.data = "mean_cl_boot", colour = "red", size = 2)

# You can supply individual functions to summarise the value at
# each x:
d + stat_summary(fun.y = "median", colour = "red", size = 2)
d + stat_summary(fun.y = "mean", colour = "red", size = 2)
d + aes(colour = factor(vs)) + stat_summary(fun.y = mean, geom="line")

d + stat_summary(fun.y = mean, fun.ymin = min, fun.ymax = max,
    colour = "red")

# d <- ggplot(diamonds, aes(carat, price))
d + geom_smooth()
d + geom_line(stat = "summary_bin", binwidth = 0.1, fun.y = "mean")

d <- ggplot(diamonds, aes(cut))
d + geom_bar()
d + stat_summary_bin(aes(y = price), fun.y = "mean", geom = "bar")

# A set of useful summary functions is provided from the Hmisc package:
stat_sum_df <- function(fun, geom="crossbar", ...)
  stat_summary(fun.data=fun, colour="red", geom=geom, width=0.2, ...)
}

# Don't use ylim to zoom into a summary plot - this throws the
# data away
p <- ggplot(mtcars, aes(cyl, mpg)) +
  stat_summary(fun.y = "mean", geom = "point")
p + ylim(15, 30)
# Instead use coord_cartesian
p + coord_cartesian(ylim = c(15, 30))

# The crossbar geom needs grouping to be specified when used with
# a continuous x axis.
  d + stat_sum_df("mean_cl_boot", mapping = aes(group = cyl))
  d + stat_sum_df("mean_sdl", mapping = aes(group = cyl))
  d + stat_sum_df("mean_sdl", mult = 1, mapping = aes(group = cyl))
  d + stat_sum_df("median_hilow", mapping = aes(group = cyl))

# There are lots of different geoms you can use to display the summaries
  d + stat_sum_df("mean_cl_normal", mapping = aes(group = cyl))
  d + stat_sum_df("mean_cl_normal", geom = "errorbar")
  d + stat_sum_df("mean_cl_normal", geom = "pointrange")
  d + stat_sum_df("mean_cl_normal", geom = "smooth")

# Summaries are more useful with a bigger data set:
  mpg2 <- subset(mpg, cyl != SL)
  m <- ggplot(mpg2, aes(x=cyl, y=hwy)) +
      geom_point() +
      stat_summary(fun.data = "mean_sdl", geom = "linerange",
                   colour = "red", size = 2, mult = 1) +
      xlab("cyl")

# An example with highly skewed distributions:
if (require("ggplot2movies")) {
  set.seed(596)
  mov <- movies[sample(nrow(movies), 1000), ]
  m2 <- ggplot(mov, aes(x=factor(round(rating)), y=votes)) + geom_point()
  m2 <- m2 + stat_summary(fun.data = "mean_cl_boot", geom = "crossbar",
                          colour = "red", width = 0.3) + xlab("rating")
  m2

# Notice how the overplotting skews off visual perception of the mean
# supplementing the raw data with summary statistics is _very_ important

# Next, we'll look at votes on a log scale.
# Transforming the scale means the data are transformed
# first, after which statistics are computed:
  m2 + scale_y_log10()
# Transforming the coordinate system occurs after the
# statistic has been computed. This means we're calculating the summary on the raw data
# and stretching the geoms onto the log scale. Compare the widths of the
# standard errors.
  m2 + coord_trans(y="log10")
}

---

**stat_unique**

Remove duplicates.

**Description**

Remove duplicates.
Usage

```r
stat_unique(mapping = NULL, data = NULL, geom = "point",
position = "identity", na.rm = FALSE, show.legend = NA,
inherit.aes = TRUE, ...)
```

Arguments

- **mapping**: The aesthetic mapping, usually constructed with `aes` or `aes_string`. Only needs to be set at the layer level if you are overriding the plot defaults.
- **data**: A layer specific dataset - only needed if you want to override the plot defaults.
- **geom**: The geometric object to use display the data
- **position**: The position adjustment to use for overlapping points on this layer
- **na.rm**: If FALSE (the default), removes missing values with a warning. If TRUE silently removes missing values.
- **show.legend**: logical. Should this layer be included in the legends? NA, the default, includes if any aesthetics are mapped. FALSE never includes, and TRUE always includes.
- **inherit.aes**: If FALSE, overrides the default aesthetics, rather than combining with them. This is most useful for helper functions that define both data and aesthetics and shouldn’t inherit behaviour from the default plot specification, e.g. `borders`.
- **...**: other arguments passed on to `layer`. This can include aesthetics whose values you want to set, not map. See `layer` for more details.

Aesthetics

`stat_unique` understands the following aesthetics (required aesthetics are in bold):

-  

Examples

```r
ggplot(mtcars, aes(vs, am)) + geom_point(alpha = 0.1)
ggplot(mtcars, aes(vs, am)) + geom_point(alpha = 0.1, stat="unique")
```

---

**theme**

*Set theme elements*

Description

Use this function to modify theme settings.

Usage

```r
theme(..., complete = FALSE, validate = TRUE)
```
Arguments

- Arguments: a list of element name, element pairings that modify the existing theme.
- complete: set this to TRUE if this is a complete theme, such as the one returned by `theme_grey()`. Complete themes behave differently when added to a ggplot object.
- validate: TRUE to run validate_element, FALSE to bypass checks.

Details

Theme elements can inherit properties from other theme elements. For example, `axis.title.x` inherits from `axis.title`, which in turn inherits from `text`. All text elements inherit directly or indirectly from `text`, all lines inherit from `line`, and all rectangular objects inherit from `rect`.

For more examples of modifying properties using inheritance, see `+.gg` and `%+replace%`.

To see a graphical representation of the inheritance tree, see the last example below.

Theme elements

The individual theme elements are:

- line: all line elements (`element_line`)
- rect: all rectangular elements (`element_rect`)
- text: all text elements (`element_text`)
- title: all title elements: plot, axes, legends (`element_text`; inherits from `text`)
- aspect.ratio: aspect ratio of the panel
- axis.title: label of axes (`element_text`; inherits from `text`)
- axis.title.x: x axis label (`element_text`; inherits from `axis.title`)
- axis.title.y: y axis label (`element_text`; inherits from `axis.title`)
- axis.text: tick labels along axes (`element_text`; inherits from `text`)
- axis.text.x: x axis tick labels (`element_text`; inherits from `axis.text`)
- axis.text.y: y axis tick labels (`element_text`; inherits from `axis.text`)
- axis.ticks: tick marks along axes (`element_line`; inherits from `line`)
- axis.ticks.x: x axis tick marks (`element_line`; inherits from `axis.ticks`)
- axis.ticks.y: y axis tick marks (`element_line`; inherits from `axis.ticks`)
- axis.ticks.length: length of tick marks (unit)
- axis.line: lines along axes (`element_line`; inherits from `line`)
- axis.line.x: line along x axis (`element_line`; inherits from `axis.line`)
- axis.line.y: line along y axis (`element_line`; inherits from `axis.line`)
- legend.background: background of legend (`element_rect`; inherits from `rect`)
- legend.margin: extra space added around legend (unit)
- legend.key: background underneath legend keys (`element_rect`; inherits from `rect`)
- legend.key.size: size of legend keys (unit; inherits from `legend.key.size`)
- legend.key.height: key background height (unit; inherits from `legend.key.size`)
- legend.key.width: key background width (unit; inherits from `legend.key.size`)
- legend.text: legend item labels (`element_text`; inherits from `text`)
- legend.text.align: alignment of legend labels (number from 0 (left) to 1 (right))
- legend.title: title of legend (`element_text`; inherits from `title`)
- legend.title.align: alignment of legend title (number from 0 (left) to 1 (right))
- legend.position: the position of legends ("none", "left", "right", "bottom", "top", or two-element numeric vector)
theme

- **legend.direction**: layout of items in legends ("horizontal" or "vertical")
- **legend.justification**: anchor point for positioning legend inside plot ("center" or two-element numeric vector)
- **legend.box**: arrangement of multiple legends ("horizontal" or "vertical")
- **legend.box.just**: justification of each legend within the overall bounding box, when there are multiple legends ("top", "bottom", "left", "right")
- **panel.background**: background of plotting area, drawn underneath plot (element_rect; inherits from rect)
- **panel.border**: border around plotting area, drawn on top of plot so that it covers tick marks and grid lines. This should probably always be used with fill (element_rect; inherits from rect)
- **panel.margin**: margin around facet panels (unit)
- **panel.margin.x**: horizontal margin around facet panels (unit; inherits from panel.margin)
- **panel.margin.y**: vertical margin around facet panels (unit; inherits from panel.margin)
- **panel.grid**: grid lines (element_line; inherits from line)
- **panel.grid.major**: major grid lines (element_line; inherits from panel.grid)
- **panel.grid.minor**: minor grid lines (element_line; inherits from panel.grid)
- **panel.grid.major.x**: vertical major grid lines (element_line; inherits from panel.grid.major)
- **panel.grid.major.y**: horizontal major grid lines (element_line; inherits from panel.grid.major)
- **panel.grid.minor.x**: vertical minor grid lines (element_line; inherits from panel.grid.minor)
- **panel.grid.minor.y**: horizontal minor grid lines (element_line; inherits from panel.grid.minor)
- **panel.ontop**: option to place the panel (background, gridlines) over the data layers. Usually used with a transparent panel.background (logical)
- **plot.background**: background of the entire plot (element_rect; inherits from rect)
- **plot.title**: plot title (text appearance) (element_text; inherits from title)
- **plot.margin**: margin around entire plot (unit with the sizes of the top, right, bottom, and left margins)
- **strip.background**: background of facet labels (element_rect; inherits from rect)
- **strip.text**: facet labels (element_text; inherits from text)
- **strip.text.x**: facet labels along horizontal direction (element_text; inherits from strip.text)
- **strip.text.y**: facet labels along vertical direction (element_text; inherits from strip.text)
- **strip.switch.pad.grid**: space between strips and axes when strips are switched (unit)
- **strip.switch.pad.wrap**: space between strips and axes when strips are switched (unit)

**See Also**

* gg
* %+replace%
* rel
  * element_blank
  * element_line
  * element_rect
  * element_text

**Examples**

```r
p <- ggplot(mtcars, aes(mpg, wt)) +
  geom_point()
p
p + theme(panel.background = element_rect(colour = "pink"))
p + theme_bw()
```
# Scatter plot of gas mileage by vehicle weight
p <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point()

# Calculate slope and intercept of line of best fit
coef(lm(mpg ~ wt, data = mtcars))

# Calculate correlation coefficient
with(mtcars, cor(wt, mpg, use = "everything", method = "pearson"))

# Add text annotation
p + geom_abline(intercept = 37, slope = -5) + geom_text(data = data.frame(),
  aes(4.5, 30, label = "Pearson-R = -.87"))

# Change the axis labels
# Original plot
p + labs(x = "Vehicle Weight", y = "Miles per Gallon")
# Or
p + labs(x = "Vehicle Weight", y = "Miles per Gallon")

# Change title appearance
# Set title to twice the base font size
p + theme(plot.title = element_text(size = rel(2)))

# Changing plot look with themes
DF <- data.frame(x = rnorm(400))
m <- ggplot(DF, aes(x = x)) +
  geom_histogram()

# Default is theme_grey()
m

# # Compare with
m + theme_bw()

# Manipulate Axis Attributes
m + theme(axis.line = element_line(size = 3, colour = "red", linetype = "dotted"))
m + theme(axis.text = element_text(colour = "blue"))
m + theme(axis.text.y = element_blank())
m + theme(axis.ticks = element_line(size = 2))
m + theme(axis.title.y = element_text(size = rel(1.5), angle = 90))
m + theme(axis.title.x = element_blank())
m + theme(axis.ticks.length = unit(0.85, "cm"))

# Legend Attributes
z <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point(aes(colour = factor(cyl))))

# Legend Attributes
z + theme(legend.position = "none")
z + theme(legend.position = "bottom")
# Or use relative coordinates between 0 and 1
z + theme(legend.position = c(.5, .5))
# Add a border to the whole legend
z + theme(legend.background = element_rect(colour = "black"))
# Legend margin controls extra space around outside of legend:
z + theme(legend.background = element_rect(),
    legend.margin = unit(1, "cm"))
z + theme(legend.background = element_rect(),
    legend.margin = unit(0, "cm"))
# Or to just the keys
z + theme(legend.key = element_rect(colour = "black"))
z + theme(legend.key = element_rect(fill = "yellow"))
z + theme(legend.key.size = unit(2.5, "cm"))
z + theme(legend.text = element_text(size = 20, colour = "red", angle = 45))
z + theme(legend.title = element_text(face = "italic"))

# To change the title of the legend use the name argument
# in one of the scale options
z + scale_colour_brewer(name = "My Legend")
z + scale_colour_grey(name = "Number of \nCylinders")

# Panel and Plot Attributes
z + theme(panel.background = element_rect(fill = "black"))
z + theme(panel.border = element_rect(linetype = "dashed", colour = "black"))
z + theme(panel.grid.major = element_line(colour = "blue"))
z + theme(panel.grid.minor = element_line(colour = "red", linetype = "dotted"))
z + theme(panel.grid.major = element_line(size = 2))
z + theme(panel.grid.major.y = element_blank(),
    panel.grid.minor.y = element_blank())
z + theme(plot.background = element_rect())
z + theme(plot.background = element_rect(fill = "green"))

# Faceting Attributes
set.seed(4940)
dsmall <- diamonds[sample(nrow(diamonds), 1000), ]
k <- ggplot(dsmall, aes(carat, ..density..)) +
    geom_histogram(binwidth = 0.2) +
    facet_grid(. ~ cut)
k + theme(strip.background = element_rect(colour = "purple", fill = "pink",
    size = 3, linetype = "dashed"))
k + theme(strip.text.x = element_text(colour = "red", angle = 45, size = 10,
    hjust = 0.5, vjust = 0.5))
k + theme(panel.margin = unit(5, "lines"))
k + theme(panel.margin.y = unit(0, "lines"))

# Put gridlines on top
meanprice <- tapply(diamonds$price, diamonds$cut, mean)
cut <- factor(levels(diamonds$cut), levels = levels(diamonds$cut))
df <- data.frame(meanprice, cut)
g <- ggplot(df, aes(cut, meanprice)) + geom_bar(stat = "identity")
g + geom_bar(stat = "identity") +
    theme(panel.background = element_blank(),
        panel.grid.major.x = element_blank(),
        panel.grid.minor.x = element_blank(),
        panel.grid.minor.y = element_blank(),
        panel.ontop = TRUE)
# Modify a theme and save it
mytheme <- theme_grey() + theme(plot.title = element_text(colour = "red"))
p + mytheme

## Not run:
## Run this to generate a graph of the element inheritance tree
build_element_graph <- function(tree) {
  require(igraph)
  require(plyr)

  inheritdf <- function(name, item) {
    if (length(item$inherit) == 0)
      data.frame()
    else
      data.frame(child = name, parent = item$inherit)
  }

  edges <- plyr::bind.fill(mapply(inheritdf, names(tree), tree))

  # Explicitly add vertices (since not all are in edge list)
  vertices <- data.frame(name = names(tree))
  graph.data.frame(edges, vertices)
}

g <- build_element_graph(ggplot2::..element_tree)
V(g)$label <- V(g)$name

set.seed(324)
par(mar=c(0,0,0,0)) # Remove unnecessary margins
plot(g, layout=layout.fruchterman.reingold, vertex.size=4, vertex.label.dist=.25)

## End(Not run)

theme_update  Get, set and update themes.

Description

Use theme_update to modify a small number of elements of the current theme or use theme_set to completely override it.

Usage

theme_update(...)
theme_get()
theme_set(new)
Arguments

... named list of theme settings
new new theme (a list of theme elements)

See Also

%+replace% and +.gg

Examples

p <- ggplot(mtcars, aes(mpg, wt)) +
  geom_point()
p
old <- theme_set(theme_bw())
p
theme_set(old)
p

old <- theme_update(panel.background = element_rect(colour = "pink"))
p
theme_set(old)
theme_get()

ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(color = mpg)) +
  theme(legend.position = c(0.95, 0.95),
        legend.justification = c(1, 1))
last_plot() +
  theme(legend.background = element_rect(fill = "white", colour = "white", size = 3))

Description

There are two types of graphics functions in base graphics, those that draw complete graphics and those that add to existing graphics.

Details

qplot() has been designed to mimic plot(), and can do the job of all other high-level plotting commands. There are only two graph types from base graphics that cannot be replicated with ggplot2: filled.contour() and persp()
Examples

```r
# High-level plotting commands

x <- runif(10)
y <- 1:10
plot(x, y); dotchart(x, y)
qplot(x, y)

plot(x, y, type = "l")
qplot(x, y, geom = "line")

plot(x, y, type = "s")
qplot(x, y, geom = "step")

plot(x, y, type = "b")
qplot(x, y, geom = c("point", "line"))

boxplot(x, y)
qplot(x, y, geom = "boxplot")

hist(x)
qplot(x, geom = "histogram")

# cdplot(factor(x), y)
# qplot(x, fill = y, geom = "density", position = "fill")

# coplot(y ~ x | a ~ b)
# qplot(x, y, facets = a ~ b)

# Many of the geoms are parameterised differently than base graphics. For
# example, hist() is parameterised in terms of the number of bins, while
# geom_histogram() is parameterised in terms of the width of each bin.
hist(x, bins = 10)
qplot(x, geom = "histogram", binwidth = .1)

# qplot() often requires data in a slightly different format to the base
# graphics functions. For example, the bar geom works with untabulated data,
# not tabulated data like barplot(); the tile and contour geoms expect data
# in a data frame, not a matrix like image() and contour().
barplot(table(x))
qplot(x, geom = "bar")

barplot(x)
qplot(seq_along(x), x, geom = "bar", stat = "identity")

# image(x)
# qplot(X1, X2, data = melt(x), geom = "tile", fill = value)

# contour(x)
# qplot(X1, X2, data = melt(x), geom = "contour", fill = value)
```
C generally the base graphics functions work with individual vectors, not
data frames like ggplot2. qplot() will try to construct a data frame if one
is not specified, but it is not always possible. If you get strange errors,
you may need to create the data frame yourself.
df <- data.frame(x = x, y = y)
with(df, plot(x, y))
qplot(x, y, data = df)

# By default, qplot() maps values to aesthetics with a scale. To override
# this behaviour and set aesthetics, overriding the defaults, you need to use I().
plot(x, y, col = "red", cex = 1)
qplot(x, y, colour = I("red"), size = I(1))

# Low-level drawing

# The low-level drawing functions which add to an existing plot are equivalent
# to adding a new layer in ggplot2.

# Base function ggplot2 layer
# curve() geom_curve()
# hline() geom_hline()
# lines() geom_line()
# points() geom_point()
# polygon() geom_polygon()
# rect() geom_rect()
# rug() geom_rug()
# segments() geom_segment()
# text() geom_text()
# vline() geom_vline()
# abline(lm(y ~ x)) geom_smooth(method = "lm")
# lines(density(x)) geom_density()
# lines(loess(x, y)) geom_smooth()

plot(x, y)
lines(x, y)
qplot(x, y) + geom_line()

# Or, building up piece-meal
qplot(x, y)
last_plot() + geom_line()

# Legends, axes and grid lines

# In ggplot2, the appearance of legends and axes is controlled by the scales.
# Axes are produced by the x and y scales, while all other scales produce legends.
# See ?theme for help changing the appearance of axes and legends.
# The appearance of grid lines is controlled by the grid.major and grid.minor
# theme options, and their position by the breaks of the x and y scales.

# Colour palettes
translate_qplot_ggplot

Translating between qplot and ggplot

Description

Within ggplot2, there are two basic methods to create plots, with qplot() and ggplot(). qplot() is designed primarily for interactive use: it makes a number of assumptions that speed most cases, but when designing multilayered plots with different data sources it can get in the way. This section describes what those defaults are, and how they map to the fuller ggplot() syntax.

Examples

# By default, qplot() assumes that you want a scatterplot,
# i.e., you want to use geom_point()
# qplot(x, y, data = data)
# ggplot(data, aes(x, y)) + geom_point()

# Using Aesthetics

# Instead of global colour palettes, ggplot2 has scales for individual plots. Much
# of the time you can rely on the default colour scale (which has somewhat better
# perceptual properties), but if you want to reuse an existing colour palette, you
# can use scale_colour_manual(). You will need to make sure that the colour
# is a factor for this to work.

palette(rainbow(5))
plot(1:5, 1:5, col = 1:5, pch = 19, cex = 4)
qplot(1:5, 1:5, col = factor(1:5), size = I(4))
last_plot() + scale_colour_manual(values = rainbow(5))

# In ggplot2, you can also use palettes with continuous values,
# with intermediate values being linearly interpolated.
qplot(0:100, 0:100, col = 0:100, size = I(4)) +
  scale_colour_gradientn(colours = rainbow(7))
last_plot() + scale_colour_gradientn(colours = terrain.colors(7))

# Graphical parameters

# The majority of par settings have some analogue within the theme system, or
# in the defaults of the geoms and scales. The appearance plot border drawn
# by box() can be controlled in a similar way by the panel.background and
# plot.background theme elements. Instead of using title(), the plot title is
# set with the title option. See ?theme for more theme elements.
last_plot() + labs(title = "My Plot Title")
# If you map additional aesthetics, these will be added to the defaults. With
# qplot() there is no way to use different aesthetic mappings (or data) in
# different layers
# qplot(x, y, data = data, shape = shape, colour = colour)
# ggplot(data, aes(x, y, shape = shape, colour = colour)) + geom_point()
#
# Aesthetic parameters in qplot() always try to map the aesthetic to a
# variable. If the argument is not a variable but a value, effectively a new column
# is added to the original dataset with that value. To set an aesthetic to a
# value and override the default appearance, you surround the value with \( I() \) in
# qplot(), or pass it as a parameter to the layer.
# qplot(x, y, data = data, colour = I("red"))
# ggplot(data, aes(x, y)) + geom_point(colour = "red")

# Changing the geom parameter changes the geom added to the plot
# qplot(x, y, data = data, geom = "line")
# ggplot(data, aes(x, y)) + geom_line()

# Not all geoms require both x and y, e.g., geom_bar() and geom_histogram().
# For these two geoms, if the y aesthetic is not supplied, both qplot and
# ggplot commands default to "count" on the y-axis
# ggplot(data, aes(x)) + geom_bar()
# qplot(x, data = data, geom = "bar")

# If a vector of multiple geom names is supplied to the geom argument, each
# geom will be added in turn
# qplot(x, y, data = data, geom = c("point", "smooth"))
# ggplot(data, aes(x, y)) + geom_point() + geom_smooth()

# Unlike the rest of ggplot2, stats and geoms are independent
# qplot(x, y, data = data, stat = "bin")
# ggplot(data, aes(x, y)) + geom_point(stat = "bin")
#
# Any layer parameters will be passed on to all layers. Most layers will ignore
# parameters that they don't need
# qplot(x, y, data = data, geom = c("point", "smooth"), method = "lm")
# ggplot(data, aes(x, y)) + geom_point(method = "lm") + geom_smooth(method = "lm")

# Scales and axes

# You can control basic properties of the x and y scales with the xlim, ylim,
# xlab and ylab arguments
# qplot(x, y, data = data, xlim = c(1, 5), xlab = "my label")
# ggplot(data, aes(x, y)) + geom_point() +
# scale_x_continuous("my label", limits = c(1, 5))
# qplot(x, y, data = data, xlim = c(1, 5), ylim = c(10, 20))
# ggplot(data, aes(x, y)) + geom_point() +
# scale_x_continuous(limits = c(1, 5)) + scale_y_continuous(limits = c(10, 20))

# Like plot(), qplot() has a convenient way of log transforming the axes.
# qplot(x, y, data = data, log = "xy")
# ggplot(data, aes(x, y)) + geom_point() + scale_x_log10() + scale_y_log10()
There are many other possible transformations, but not all are accessible from within qplot(), see ?scale_continuous for more.

Plot options

qplot() recognises the same options as plot does, and converts them to their ggplot2 equivalents. See ?theme for more on ggplot options.

```
qplot(x, y, data = data, main = "title", asp = 1)
```

```
ggplot(data, aes(x, y)) + geom_point() + labs(title = "title") + theme(aspect.ratio = 1)
```

---

**Translating between qplot and lattice**

**Description**

The major difference between lattice and ggplot2 is that lattice uses a formula based interface. ggplot2 does not because the formula does not generalise well to more complicated situations.

**Examples**

```r
## Not run:
library(lattice)
if (require("ggplot2movies")) {
  xyplot(rating ~ year, data=movies)
  qplot(year, rating, data=movies)

  xyplot(rating ~ year | Comedy + Action, data = movies)
  qplot(year, rating, data = movies, facets = ~ Comedy + Action)
  # Or maybe
  qplot(year, rating, data = movies, facets = Comedy ~ Action)

  # While lattice has many different functions to produce different types of graphics (which are all basically equivalent to setting the panel argument),
  # ggplot2 has qplot().

  stripplot(~ rating, data = movies, jitter.data = TRUE)
  qplot(rating, l, data = movies, geom = "jitter")

  histogram(~ rating, data = movies)
  qplot(rating, data = movies, geom = "histogram")

  bwplot(Comedy ~ rating ,data = movies)
  qplot(factor(Comedy), rating, data = movies, type = "boxplot")

  xyplot(wt ~ mpg, mtcars, type = c("p","smooth"))
  qplot(mpg, wt, data = mtcars, geom = c("point","smooth"))

  xyplot(wt ~ mpg, mtcars, type = c("p","r"))
```
txhousing

Housing sales in TX.

Description

Information about the housing market in Texas provided by the TAMU real estate center, http://recenter.tamu.edu/.
Usage

txhousing

Format

A data frame with 8602 observations and 9 variables:

- cityName of MLS area
- year, month, date
- sales
- volume
- median
- listings
- inventory: "Months inventory": amount of time it would take to sell all current listings at current pace of sales.

Description

Modify geom/stat aesthetic defaults for future plots

Usage

update_geom_defaults(geom, new)

update_stat_defaults(stat, new)

Arguments

new

Named list of aesthetics.

stat, geom

Name of geom/stat to modify (like "point" or "bin"), or a Geom/Stat object (like GeomPoint or StatBin).

Examples

update_geom_defaults("point", list(colour = "darkblue"))
ggplot(mtcars, aes(mpg, wt)) + geom_point()
update_geom_defaults("point", list(colour = "black"))
update_labels

Update axis/legend labels

Description
Update axis/legend labels

Usage
update_labels(p, labels)

Arguments
- \texttt{p} plot to modify
- \texttt{labels} named list of new labels

Examples
\begin{verbatim}
p <- ggplot(mtcars, aes(mpg, wt)) + geom_point()
update_labels(p, list(x = "New x"))
update_labels(p, list(x = expression(x / y ^ 2)))
update_labels(p, list(x = "New x", y = "New Y"))
update_labels(p, list(colour = "Fail silently"))
\end{verbatim}
Index

*Topic **datasets**
  - diamonds, 33
  - economics, 34
  - faithful, 43
  - graphical-units, 117
  - luv_colours, 134
  - midwest, 137
  - mpg, 138
  - msleep, 138
  - presidential, 144
  - seals, 167
  - txhousing, 191
*Topic **hplot**
  - print.ggplot, 144
  - .gg, 180, 181, 185
  - .pt (graphical-units), 117
  - .stroke (graphical-units), 117
  - %+% (+.gg), 6
  - %+replace% (+.gg), 6
  - %+replace%, 180, 181, 185

  - add_theme, 7
  - aes, 8, 9, 30, 52, 54, 56, 57, 60, 62, 64, 66, 69, 71, 74, 76, 79, 80, 82, 86, 88, 91, 93, 95, 97, 100, 101, 103, 105, 108, 110, 133, 168–170, 172–174, 176, 179
  - aes_, 9, 50, 52, 54, 56, 57, 60, 62, 64, 66, 69, 71, 74, 76, 79, 80, 82, 86, 88, 91, 93, 95, 97, 100, 101, 103, 105, 108, 110, 133, 176
  - aes_colour_fill_alpha, 8, 10
  - aes_group_order, 8, 11
  - aes_linetype_size_shape, 8, 12
  - aes_position, 8, 14
  - aes_q, 8
  - aes_q(aes_), 9
  - aes_string, 8, 168–170, 172–174, 179
  - aes_string (aes_), 9
  - annotate, 15
  - annotation_custom, 16
  - annotation_logticks, 17
  - annotation_map, 19
  - annotation_raster, 19
  - arrow, 88
  - as.list.ggproto, 20
  - as_labeller, 21, 127, 129, 130
  - autoplot, 22
  - bandwidth.nrd, 69
  - boxcox_trans, 157, 165
  - boxplot.stats, 57
  - bquote, 129
  - brewer.pal, 150
  - calc_element, 23
  - capitalize, 127
  - cld, 45
  - color (aes_colour_fill_alpha), 10
  - colors, 134
  - colour (aes_colour_fill_alpha), 10
  - continuous_scale, 149, 160, 165
  - cooks.distance, 46
  - coord_cartesian, 24, 134
  - coord_equal (coord_fixed), 25
  - coord_fixed, 25
  - coord_flip, 26
  - coord_map, 27, 27
  - coord_polar, 29
  - coord_quickmap, 27
  - coord_quickmap (coord_map), 27
  - coord_trans, 18, 30
  - county, 135
  - cut, 32
  - cut_interval, 32
  - cut_number, 33
INDEX

195

cut_number (cut_interval), 32
cut_width (cut_interval), 32
density, 67, 110
diamonds, 33
discrete_scale, 149, 150, 152–154, 160, 162, 163, 166
economics, 34
economics_long (economics), 34
element_blank, 34, 181
element_line, 35, 181
element_rect, 35, 181
element_text, 36, 120, 122, 181
eqscplot, 25
expand_limits, 36
facet_grid, 37, 127, 146
facet_null, 41
facet_wrap, 41, 130, 146
faithful, 43
faithfuld, 43
fill (aes_colour_fill_alpha), 10
format.ggproto, 44
fortify, 19, 22, 44, 86, 112
fortify-multcomp, 45
fortify.cld (fortify-multcomp), 45
fortify.confint.glht (fortify-multcomp), 45
fortify.glht (fortify-multcomp), 45
fortify.Line (fortify.sp), 48
fortify.Lines (fortify.sp), 48
fortify.lm, 44, 46
fortify.map, 47
fortify.Polygon (fortify.sp), 48
fortify.Polygons (fortify.sp), 48
fortify.sp, 48
fortify.SpatialLinesDataFrame (fortify.sp), 48
fortify.SpatialPolygons (fortify.sp), 48
fortify.SpatialPolygonsDataFrame (fortify.sp), 48
fortify.summary.glht (fortify-multcomp), 45
france, 135
gam, 105
gem_abline, 49
gem_area, 140
gem_area (geom_ribbon), 99
gem_bar, 51, 100, 101, 140
gem_bin2d, 54
gem_blank, 36, 56
gem_boxplot, 57, 81, 91
gem_contour, 60, 70
gem_count, 62
gem_crossbar, 64, 177
gem_curve (geom_segment), 102
gem_density, 66
gem_density2d (geom_density_2d), 68
gem_density_2d, 61, 68, 91
gem_dotplot, 71
gem_errorbar, 75, 177
gem_errorbar (geom_crossbar), 64
gem_errorbarh, 74
gem_freqpoly, 68, 75
gem_hex, 78
gem_histogram, 52, 53, 68
gem_histogram (geom_freqpoly), 75
gem_hline (geom_abline), 49
gem_jitter, 59, 80, 91, 92
gem_label, 82
gem_line, 50, 104
gem_line (geom_path), 87
gem_linerange, 101, 177
gem_linerange (geom_crossbar), 64
gem_map, 85
gem_path, 87, 94, 104
gem_point, 62, 81, 90
gem_pointrange, 177
gem_pointrange (geom_crossbar), 64
gem_polygon, 22, 89, 93, 101
gem_qq (stat_qq), 172
gem_quantile, 91, 95
gem_raster, 19, 97
gem_rect (geom_raster), 97
gem_ribbon, 94, 99
gem_rug, 101
gem_segment, 50, 89, 102
gem_smooth, 65, 91, 105
gem_spoke, 104, 108
gem_step (geom_path), 87
gem_text (geom_label), 82
gem_tile, 27
gem_tile (geom_raster), 97
gem_violin, 68, 109, 111
gem_vline (geom_abline), 49
scale_color_gradient
(scale_colour_gradient), 151
scale_color_gradient2
(scale_colour_gradient), 151
scale_color_gradientn
(scale_colour_gradient), 151
scale_color_hue (scale_colour_hue), 154
scale_color_identity (scale_identity), 160
scale_color_manual (scale_manual), 162
scale_colour_brewer, 149, 152–154
scale_colour_continuous
(scale_colour_continuous), 151
scale_colour_discrete
(scale_colour_discrete), 154
scale_colour_distiller
(scale_colour_brewer), 149
scale_colour_gradient, 150, 151, 153, 154
scale_colour_gradient2
(scale_colour_gradient), 151
scale_colour_gradientn
(scale_colour_gradient), 151
scale_colour_grey, 150, 152, 153, 154
scale_colour_identity (scale_identity), 160
scale_colour_manual (scale_manual), 162
scale_continuous, 155, 160
scale_date, 157, 158
scale_fill_brewer
(scale_colour_brewer), 149
scale_fill_continuous
(scale_colour_gradient), 151
scale_fill_discrete (scale_colour_hue), 154
scale_fill_distiller
(scale_colour_brewer), 149
scale_fill_gradient
(scale_colour_gradient), 151
scale_fill_gradient2
(scale_colour_gradient), 151
scale_fill_gradientn
(scale_colour_gradient), 151
scale_fill_grey (scale_colour_grey), 153
scale_fill_hue (scale_colour_hue), 154
scale_fill_identity (scale_identity), 160
scale_fill_manual (scale_manual), 162
scale_identity, 160
scale_linetype, 161
scale_linetype_continuous
(scale_linetype), 161
scale_linetype_discrete
(scale_linetype), 161
scale_linetype_identity
(scale_identity), 161
scale_linetype_manual (scale_manual), 162
scale_manual, 162
scale_radius (scale_size), 164
scale_shape, 163
scale_shape_continuous (scale_shape), 163
scale_shape_discrete (scale_shape), 163
scale_shape_identity (scale_identity), 160
scale_shape_manual (scale_manual), 162
scale_size, 92, 164
scale_size_area, 165
scale_size_area (scale_size), 164
scale_size_continuous (scale_size), 164
scale_size_discrete (scale_size), 164
scale_size_identity (scale_identity), 160
scale_size_manual (scale_manual), 162
scale_x_continuous (scale_continuous), 155
scale_x_date (scale_date), 158
scale_x_datetime (scale_date), 158
scale_x_discrete, 166
scale_x_log10 (scale_continuous), 155
scale_x_reverse (scale_continuous), 155
scale_x_sqrt (scale_continuous), 155
scale_y_continuous (scale_continuous), 155
scale_y_date (scale_date), 158
scale_y_datetime (scale_date), 158
scale_y_discrete (scale_x_discrete), 166
scale_y_log10, 18
scale_y_reverse (scale_continuous), 155
scale_y_sqrt (scale_continuous), 155
seals, 167
seq_gradient_pal, 152
shape(aes_linetype_size_shape), 12
size(aes_linetype_size_shape), 12
smean.cl.boot, 125
smean.cl.normal, 125
smean.sdl, 125
smedian.hilow, 125
stat_bin, 51, 53, 77, 175
stat_bin(geom_freqpoly), 75
stat_bin2d, 79, 175
stat_bin2d(geom_bin2d), 54
stat_bin2d(geom_bin2d), 54
stat_bin_hex(geom_hex), 78
stat_binhex, 55
stat_binhex(geom_hex), 78
stat_boxplot(geom_boxplot), 57
stat_contour(geom_contour), 70
stat_contour(geom_contourplot), 60
stat_count(geom_bar), 51
stat_density, 111
stat_density(geom_density), 66
stat_density2d(geom_density_2d), 68
stat_density2d(geom_density_2d), 68
stat_ecdf, 167
stat_ellipse, 168
stat_function, 170
stat_identity, 171
stat_qq, 172
stat_quantile, 59
stat_quantile(geom_quantile), 95
stat_smooth(geom_smooth), 105
stat_spoke(geom_spoke), 108
stat_sum, 70, 91
stat_sum(geom_count), 62
stat_summary, 65, 125, 136, 174
stat_summary(stat_summary_bin), 175
stat_summary2d(stat_summary_2d), 174
stat_summary2d(stat_summary_2d), 174
stat_summary_bin, 175
stat_summary_hex, 175
stat_summary_hex(stat_summary_2d), 174
stat_unique, 178
stat_ydensity(geom_violin), 109
state, 135
strftime, 159
strwrap, 129
substitute, 9
theme, 7, 120, 122, 123, 179
theme_bw(ggtheme), 116
theme_classic(ggtheme), 116
theme_dark(ggtheme), 116
theme_get(theme_update), 184
theme_gray(ggtheme), 116
theme_light(ggtheme), 116
theme_minimum(ggtheme), 116
theme_set(theme_update), 184
theme_update, 184
theme_void(ggtheme), 116
trs_new, 30, 157, 165
translate_qplot_base, 185
translate_qplot_ggplot, 188
txhousing, 191
unit, 17, 120, 123
update_geom_defaults, 192
update_labels, 193
update_stat_defaults
(update_geom_defaults), 192
usa, 135
waiver, 120, 122
world, 135
world2, 135
x(aes_position), 14
xend(aes_position), 14
xlab(labs), 131
xlim(lims), 133
xmax(aes_position), 14
xmin(aes_position), 14
y(aes_position), 14
yend(aes_position), 14
ylab(labs), 131
ylim(lims), 133
ymax(aes_position), 14
ymin(aes_position), 14