Package ‘ggplot2’

September 11, 2025

Title Create Elegant Data Visualisations Using the Grammar of Graphics

Version 4.0.0

Description A system for 'declaratively' creating graphics, based on ``The Grammar of Graphics''. You provide the data, tell 'ggplot2' how to map variables to aesthetics, what graphical primitives to use, and it takes care of the details.

License MIT + file LICENSE

URL `https://ggplot2.tidyverse.org`, `https://github.com/tidyverse/ggplot2`

BugReports `https://github.com/tidyverse/ggplot2/issues`

Depends R (>= 4.1)

Imports cli, grDevices, grid, gtable (>= 0.3.6), isoband, lifecycle (> 1.0.1), rlang (>= 1.1.0), S7, scales (>= 1.4.0), stats, vctrs (>= 0.6.0), withr (>= 2.5.0)

Suggests broom, covr, dplyr, ggplot2movies, hexbin, Hmisc, knitr, mapproj, maps, MASS, mgcv, multcomp, munsell, nlme, profvis, quantreg, ragg (>= 1.2.6), RColorBrewer, rmarkdown, roxygen2, rpart, sf (>= 0.7-3), svglite (>= 2.1.2), testthat (>= 3.1.5), tibble, vdiffr (>= 1.0.6), xml2

Enhances sp

VignetteBuilder knitr

Config/Needs/website ggtext, tidyr, forcats, tidyverse/tidytemplate

Config/testthat/edition 3

Config/usethis/last-upkeep 2025-04-23

Encoding UTF-8

LazyData true

RoxygenNote 7.3.2

Collate 'ggproto.R' 'ggplot-global.R' 'aaa-.R' 'aes-colour-fill-alpha.R' 'aes-evaluation.R' 'aes-group-order.R' 'aes-linetype-size-shape.R'

'aes-position.R' 'all-classes.R' 'compat-plyr.R' 'utilities.R'
'aes.R' 'annotation-borders.R' 'utilities-checks.R'
'legend-draw.R' 'geom-.R' 'annotation-custom.R'
'annotation-logticks.R' 'scale-type.R' 'layer.R'
'make-constructor.R' 'geom-polygon.R' 'geom-map.R'
'annotation-map.R' 'geom-raster.R' 'annotation-raster.R'
'annotation.R' 'autolayer.R' 'autoplot.R' 'axis-secondary.R'
'backports.R' 'bench.R' 'bin.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-radial.R'
'coord-sf.R' 'coord-transform.R' 'data.R' 'docs_layer.R'
'facet-.R' 'facet-grid-.R' 'facet-null.R' 'facet-wrap.R'
'fortify-map.R' 'fortify-models.R' 'fortify-spatial.R'
'fortify.R' 'stat-.R' 'geom-abline.R' 'geom-rect.R'
'geom-bar.R' 'geom-tile.R' 'geom-bin2d.R' 'geom-blank.R'
'geom-boxplot.R' 'geom-col.R' 'geom-path.R' 'geom-contour.R'
'geom-point.R' 'geom-count.R' 'geom-crossbar.R'
'geom-segment.R' 'geom-curve.R' 'geom-defaults.R'
'geom-ribbon.R' 'geom-density.R' 'geom-density2d.R'
'geom-dotplot.R' 'geom-errorbar.R' 'geom-freqpoly.R'
'geom-function.R' 'geom-hex.R' 'geom-histogram.R'
'geom-hline.R' 'geom-jitter.R' 'geom-label.R'
'geom-linerange.R' 'geom-pointrange.R' 'geom-quantile.R'
'geom-rug.R' 'geom-sf.R' 'geom-smooth.R' 'geom-spoke.R'
'geom-text.R' 'geom-violin.R' 'geom-vline.R'
'ggplot2-package.R' 'grob-absolute.R' 'grob-dotstack.R'
'grob-null.R' 'grouping.R' 'properties.R' 'margins.R'
'theme-elements.R' 'guide-.R' 'guide-axis.R'
'guide-axis-logticks.R' 'guide-axis-stack.R'
'guide-axis-theta.R' 'guide-legend.R' 'guide-bins.R'
'guide-colorbar.R' 'guide-colorsteps.R' 'guide-custom.R'
'guide-none.R' 'guide-old.R' 'guides-.R' 'guides-grid.R'
'hexbin.R' 'import-standalone-obj-type.R'
'import-standalone-types-check.R' 'labeller.R' 'labels.R'
'layer-sf.R' 'layout.R' 'limits.R' 'performance.R'
'plot-build.R' 'plot-construction.R' 'plot-last.R' 'plot.R'
'position-.R' 'position-collide.R' 'position-dodge.R'
'position-dodge2.R' 'position-identity.R' 'position-jitter.R'
'position-jitterdodge.R' 'position-nudge.R' 'position-stack.R'
'quick-plot.R' 'reshape-add-margins.R' 'save.R' 'scale-.R'
'scale-alpha.R' 'scale-binned.R' 'scale-brewer.R'
'scale-colour.R' 'scale-continuous.R' 'scale-date.R'
'scale-discrete-.R' 'scale-expansion.R' 'scale-gradient.R'
'scale-grey.R' 'scale-hue.R' 'scale-identity.R'
'scale-linetype.R' 'scale-linewidth.R' 'scale-manual.R'
'scale-shape.R' 'scale-size.R' 'scale-steps.R' 'scale-view.R'
'scale-viridis.R' 'scales-.R' 'stat-align.R' 'stat-bin.R'
'stat-summary-2d.R' 'stat-bin2d.R' 'stat-bindot.R'

'stat-binhex.R' 'stat-boxplot.R' 'stat-connect.R'
'stat-contour.R' 'stat-count.R' 'stat-density-2d.R'
'stat-density.R' 'stat-ecdf.R' 'stat-ellipse.R'
'stat-function.R' 'stat-identity.R' 'stat-manual.R'
'stat-qq-line.R' 'stat-qq.R' 'stat-quantilemethods.R'
'stat-sf-coordinates.R' 'stat-sf.R' 'stat-smooth-methods.R'
'stat-smooth.R' 'stat-sum.R' 'stat-summary-bin.R'
'stat-summary-hex.R' 'stat-summary.R' 'stat-unique.R'
'stat-ydensity.R' 'summarise-plot.R' 'summary.R' 'theme.R'
'theme-defaults.R' 'theme-current.R' 'theme-sub.R'
'utilities-break.R' 'utilities-grid.R' 'utilities-help.R'
'utilities-patterns.R' 'utilities-resolution.R'
'utilities-tidy-eval.R' 'zxx.R' 'zzz.R'

NeedsCompilation no

Author Hadley Wickham [aut] (ORCID: <`https://orcid.org/0000-0003-4757-117X`>),
Winston Chang [aut] (ORCID: <`https://orcid.org/0000-0002-1576-2126`>),
Lionel Henry [aut],
Thomas Lin Pedersen [aut, cre] (ORCID:
<`https://orcid.org/0000-0002-5147-4711`>),
Kohske Takahashi [aut],
Claus Wilke [aut] (ORCID: <`https://orcid.org/0000-0002-7470-9261`>),
Kara Woo [aut] (ORCID: <`https://orcid.org/0000-0002-5125-4188`>),
Hiroaki Yutani [aut] (ORCID: <`https://orcid.org/0000-0002-3385-7233`>),
Dewey Dunnington [aut] (ORCID: <`https://orcid.org/0000-0002-9415-4582`>),
Teun van den Brand [aut] (ORCID:
<`https://orcid.org/0000-0002-9335-7468`>),
Posit, PBC [cph, fnd] (ROR: <`https://ror.org/03wc8by49`>)

Maintainer Thomas Lin Pedersen <`thomas.pedersen@posit.co`>

Repository CRAN

Date/Publication 2025-09-11 07:10:02 UTC

Contents

`add_gg` *Add components to a plot*

Description

+ is the key to constructing sophisticated ggplot2 graphics. It allows you to start simple, then get more and more complex, checking your work at each step.

Usage

```
add_gg(e1, e2)

e1 %+% e2
```

Arguments

`e1`	An object of class `ggplot()` or a `theme()`.
`e2`	A plot component, as described below.

What can you add?

You can add any of the following types of objects:

- An `aes()` object replaces the default aesthetics.
- A layer created by a `geom_` or `stat_` function adds a new layer.
- A `scale` overrides the existing scale.
- A `theme()` modifies the current theme.
- A `coord` overrides the current coordinate system.
- A `facet` specification overrides the current faceting.

To replace the current default data frame, you must use `%+%`, due to S3 method precedence issues.

You can also supply a list, in which case each element of the list will be added in turn.

See Also

`theme()`

Examples

```
base <-
 ggplot(mpg, aes(displ, hwy)) +
 geom_point()
base + geom_smooth()

# To override the data, you must use %+%
base %+% subset(mpg, fl == "p")

# Alternatively, you can add multiple components with a list.
# This can be useful to return from a function.
base + list(subset(mpg, fl == "p"), geom_smooth())
```

`aes` *Construct aesthetic mappings*

Description

Aesthetic mappings describe how variables in the data are mapped to visual properties (aesthetics) of geoms. Aesthetic mappings can be set in `ggplot()` and in individual layers.

Usage

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

Arguments

x, y, ...	<data-masking> List of name-value pairs in the form aesthetic = variable describing which variables in the layer data should be mapped to which aesthetics used by the paired geom/stat. The expression variable is evaluated within the layer data, so there is no need to refer to the original dataset (i.e., use ggplot(df, aes(variable)) instead of ggplot(df, aes(df$variable))). The names for x and y aesthetics are typically omitted because they are so common; all other aesthetics must be named.

Details

This function also standardises aesthetic names by converting color to colour (also in substrings, e.g., point_color to point_colour) and translating old style R names to ggplot names (e.g., pch to shape and cex to size).

Value

An S7 object representing a list with class mapping. Components of the list are either quosures or constants.

Quasiquotation

aes() is a quoting function. This means that its inputs are quoted to be evaluated in the context of the data. This makes it easy to work with variables from the data frame because you can name those directly. The flip side is that you have to use quasiquotation to program with aes(). See a tidy evaluation tutorial such as the dplyr programming vignette to learn more about these techniques.

Note

Using I() to create objects of class 'AsIs' causes scales to ignore the variable and assumes the wrapped variable is direct input for the grid package. Please be aware that variables are sometimes combined, like in some stats or position adjustments, that may yield unexpected results with 'AsIs' variables.

See Also

vars() for another quoting function designed for faceting specifications.

Run vignette("ggplot2-specs") to see an overview of other aesthetics that can be modified.

Delayed evaluation for working with computed variables.

Other aesthetics documentation: aes_colour_fill_alpha, aes_group_order, aes_linetype_size_shape, aes_position

Examples

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

# You can also map aesthetics to functions of variables
aes(x = mpg ^ 2, y = wt / cyl)
```

```
# Or to constants
aes(x = 1, colour = "smooth")

# Aesthetic names are automatically standardised
aes(col = x)
aes(fg = x)
aes(color = x)
aes(colour = x)

# aes() is passed to either ggplot() or specific layer. Aesthetics supplied
# to ggplot() are used as defaults for every layer.
ggplot(mpg, aes(displ, hwy)) + geom_point()
ggplot(mpg) + geom_point(aes(displ, hwy))

# Tidy evaluation ----------------------------------------------------
# aes() automatically quotes all its arguments, so you need to use tidy
# evaluation to create wrappers around ggplot2 pipelines. The
# simplest case occurs when your wrapper takes dots:
scatter_by <- function(data, ...) {
  ggplot(data) + geom_point(aes(...))
}
scatter_by(mtcars, disp, drat)

# If your wrapper has a more specific interface with named arguments,
# you need the "embrace operator":
scatter_by <- function(data, x, y) {
  ggplot(data) + geom_point(aes({{ x }}, {{ y }}))
}
scatter_by(mtcars, disp, drat)

# Note that users of your wrapper can use their own functions in the
# quoted expressions and all will resolve as it should!
cut3 <- function(x) cut_number(x, 3)
scatter_by(mtcars, cut3(disp), drat)
```

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

Description

These aesthetics parameters change the colour (colour and fill) and the opacity (alpha) of geom elements on a plot. Almost every geom has either colour or fill (or both), as well as can have their alpha modified. Modifying colour on a plot is a useful way to enhance the presentation of data, often especially when a plot graphs more than two variables.

Colour and fill

The colour aesthetic is used to draw lines and strokes, such as in geom_point() and geom_line(), but also the line contours of geom_rect() and geom_polygon(). The fill aesthetic is used to colour the inside areas of geoms, such as `geom_rect()` and `geom_polygon()`, but also the insides of shapes 21-25 of `geom_point()`.

Colours and fills can be specified in the following ways:

- A name, e.g., `"red"`. R has 657 built-in named colours, which can be listed with `grDevices::colors()`.
- An rgb specification, with a string of the form `"#RRGGBB"` where each of the pairs `RR`, `GG`, `BB` consists of two hexadecimal digits giving a value in the range `00` to `FF`. You can optionally make the colour transparent by using the form `"#RRGGBBAA"`.
- An `NA`, for a completely transparent colour.

Alpha

Alpha refers to the opacity of a geom. Values of `alpha` range from 0 to 1, with lower values corresponding to more transparent colors.

Alpha can additionally be modified through the `colour` or `fill` aesthetic if either aesthetic provides color values using an rgb specification (`"#RRGGBBAA"`), where `AA` refers to transparency values.

See Also

- Other options for modifying colour: `scale_colour_brewer()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_hue()`, `scale_colour_identity()`, `scale_colour_manual()`, `scale_colour_viridis_d()`
- Other options for modifying fill: `scale_fill_brewer()`, `scale_fill_gradient()`, `scale_fill_grey()`, `scale_fill_hue()`, `scale_fill_identity()`, `scale_fill_manual()`, `scale_fill_viridis_d()`
- Other options for modifying alpha: `scale_alpha()`, `scale_alpha_manual()`, `scale_alpha_identity()`
- Run `vignette("ggplot2-specs")` to see an overview of other aesthetics that can be modified.

Other aesthetics documentation: `aes()`, `aes_group_order`, `aes_linetype_size_shape`, `aes_position`

Examples

```
# Bar chart example
p <- ggplot(mtcars, aes(factor(cyl)))
# Default plotting
p + geom_bar()
# To change the interior colouring use fill aesthetic
p + geom_bar(fill = "red")
# Compare with the colour aesthetic which changes just the bar outline
p + geom_bar(colour = "red")
# Combining both, you can see the changes more clearly
p + geom_bar(fill = "white", colour = "red")
# Both colour and fill can take an rgb specification.
p + geom_bar(fill = "#00abff")
# Use NA for a completely transparent colour.
p + geom_bar(fill = NA, colour = "#00abff")

# Colouring scales differ depending on whether a discrete or
# continuous variable is being mapped. For example, when mapping
# fill to a factor variable, a discrete colour scale is used.
ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) + geom_bar()

# When mapping fill to continuous variable a continuous colour
# scale is used.
ggplot(faithfuld, aes(waiting, eruptions)) +
  geom_raster(aes(fill = density))

# Some geoms only use the colour aesthetic but not the fill
# aesthetic (e.g. geom_point() or geom_line()).
p <- ggplot(economics, aes(x = date, y = unemploy))
p + geom_line()
p + geom_line(colour = "green")
p + geom_point()
p + geom_point(colour = "red")

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

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

aes_eval *Control aesthetic evaluation*

Description

Most aesthetics are mapped from variables found in the data. Sometimes, however, you want to delay the mapping until later in the rendering process. ggplot2 has three stages of the data that you can map aesthetics from, and three functions to control at which stage aesthetics should be evaluated.

`after_stat()` replaces the old approaches of using either `stat()`, e.g. `stat(density)`, or surrounding the variable names with `..`, e.g. `..density..`.

Usage

```
# These functions can be used inside the `aes()` function
# used as the `mapping` argument in layers, for example:
# geom_density(mapping = aes(y = after_stat(scaled)))

after_stat(x)

after_scale(x)

from_theme(x)

stage(start = NULL, after_stat = NULL, after_scale = NULL)
```

Arguments

`x`	<data-masking> An aesthetic expression using variables calculated by the stat (`after_stat()`) or layer aesthetics (`after_scale()`).
`start`	<data-masking> An aesthetic expression using variables from the layer data.
`after_stat`	<data-masking> An aesthetic expression using variables calculated by the stat.
`after_scale`	<data-masking> An aesthetic expression using layer aesthetics.

Staging

Below follows an overview of the three stages of evaluation and how aesthetic evaluation can be controlled.

Stage 1: direct input at the start:

The default is to map at the beginning, using the layer data provided by the user. If you want to map directly from the layer data you should not do anything special. This is the only stage where the original layer data can be accessed.

```
# 'x' and 'y' are mapped directly
ggplot(mtcars) + geom_point(aes(x = mpg, y = disp))
```

Stage 2: after stat transformation:

The second stage is after the data has been transformed by the layer stat. The most common example of mapping from stat transformed data is the height of bars in `geom_histogram()`: the height does not come from a variable in the underlying data, but is instead mapped to the `count` computed by `stat_bin()`. In order to map from stat transformed data you should use the `after_stat()` function to flag that evaluation of the aesthetic mapping should be postponed until after stat transformation. Evaluation after stat transformation will have access to the variables calculated by the stat, not the original mapped values. The 'computed variables' section in each stat lists which variables are available to access.

```
# The 'y' values for the histogram are computed by the stat
ggplot(faithful, aes(x = waiting)) +
  geom_histogram()
```

```
# Choosing a different computed variable to display, matching up the
# histogram with the density plot
ggplot(faithful, aes(x = waiting)) +
  geom_histogram(aes(y = after_stat(density))) +
  geom_density()
```

Stage 3: after scale transformation:

The third and last stage is after the data has been transformed and mapped by the plot scales. An example of mapping from scaled data could be to use a desaturated version of the stroke colour for fill. You should use `after_scale()` to flag evaluation of mapping for after data has been scaled. Evaluation after scaling will only have access to the final aesthetics of the layer (including non-mapped, default aesthetics).

```
# The exact colour is known after scale transformation
ggplot(mpg, aes(cty, colour = factor(cyl))) +
  geom_density()

# We re-use colour properties for the fill without a separate fill scale
ggplot(mpg, aes(cty, colour = factor(cyl))) +
  geom_density(aes(fill = after_scale(alpha(colour, 0.3))))
```

Complex staging:

Sometimes, you may want to map the same aesthetic multiple times, e.g. map `x` to a data column at the start for the layer stat, but remap it later to a variable from the stat transformation for the layer geom. The `stage()` function allows you to control multiple mappings for the same aesthetic across all three stages of evaluation.

```
# Use stage to modify the scaled fill
ggplot(mpg, aes(class, hwy)) +
  geom_boxplot(aes(fill = stage(class, after_scale = alpha(fill, 0.4))))

# Using data for computing summary, but placing label elsewhere.
# Also, we're making our own computed variables to use for the label.
ggplot(mpg, aes(class, displ)) +
  geom_violin() +
  stat_summary(
    aes(
      y = stage(displ, after_stat = 8),
      label = after_stat(paste(mean, "±", sd))
    ),
    geom = "text",
    fun.data = ~ round(data.frame(mean = mean(.x), sd = sd(.x)), 2)
  )
```

Conceptually, `aes(x)` is equivalent to `aes(stage(start = x))`, and `aes(after_stat(count))` is equivalent to `aes(stage(after_stat = count))`, and so on. `stage()` is most useful when at least two of its arguments are specified.

Theme access:

The `from_theme()` function can be used to acces the `element_geom()` fields of the `theme(geom)` argument. Using `aes(colour = from_theme(ink))` and `aes(colour = from_theme(accent))` allows swapping between foreground and accent colours.

Examples

```
# Default histogram display
ggplot(mpg, aes(displ)) +
  geom_histogram(aes(y = after_stat(count)))

# Scale tallest bin to 1
ggplot(mpg, aes(displ)) +
  geom_histogram(aes(y = after_stat(count / max(count))))

# Use a transparent version of colour for fill
ggplot(mpg, aes(class, hwy)) +
  geom_boxplot(aes(colour = class, fill = after_scale(alpha(colour, 0.4))))

# Use stage to modify the scaled fill
ggplot(mpg, aes(class, hwy)) +
  geom_boxplot(aes(fill = stage(class, after_scale = alpha(fill, 0.4))))

# Making a proportional stacked density plot
ggplot(mpg, aes(cty)) +
  geom_density(
    aes(
      colour = factor(cyl),
      fill = after_scale(alpha(colour, 0.3)),
      y = after_stat(count / sum(n[!duplicated(group)]))
    ),
    position = "stack", bw = 1
  ) +
  geom_density(bw = 1)

# Imitating a ridgeline plot
ggplot(mpg, aes(cty, colour = factor(cyl))) +
  geom_ribbon(
    stat = "density", outline.type = "upper",
    aes(
      fill = after_scale(alpha(colour, 0.3)),
      ymin = after_stat(group),
      ymax = after_stat(group + ndensity)
    )
  )

# Labelling a bar plot
ggplot(mpg, aes(class)) +
  geom_bar() +
  geom_text(
    aes(
      y = after_stat(count + 2),
      label = after_stat(count)
    ),
    stat = "count"
  )

# Labelling the upper hinge of a boxplot,
# inspired by June Choe
ggplot(mpg, aes(displ, class)) +
  geom_boxplot(outlier.shape = NA) +
  geom_text(
    aes(
      label = after_stat(xmax),
      x = stage(displ, after_stat = xmax)
    ),
    stat = "boxplot", hjust = -0.5
  )
```

`aes_group_order` *Aesthetics: grouping*

Description

The `group` aesthetic is by default set to the interaction of all discrete variables in the plot. This choice 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.

Details

For most applications the grouping is set implicitly by mapping one or more discrete variables to `x`, `y`, `colour`, `fill`, `alpha`, `shape`, `size`, and/or `linetype`. This is demonstrated in the examples below.

There are three common cases where the default does not display the data correctly.

1. `geom_line()` where there are multiple individuals and the plot tries to connect every observation, even across individuals, with a line.
2. `geom_line()` where a discrete x-position implies groups, whereas observations span the discrete x-positions.
3. When the grouping needs to be different over different layers, for example when computing a statistic on all observations when another layer shows individuals.

The examples below use a longitudinal dataset, `Oxboys`, from the nlme package to demonstrate these cases. `Oxboys` records the heights (height) and centered ages (age) of 26 boys (Subject), measured on nine occasions (Occasion).

See Also

- Geoms commonly used with groups: `geom_bar()`, `geom_histogram()`, `geom_line()`
- Run `vignette("ggplot2-specs")` to see an overview of other aesthetics that can be modified.

Other aesthetics documentation: `aes()`, `aes_colour_fill_alpha`, `aes_linetype_size_shape`, `aes_position`

Examples

```
p <- ggplot(mtcars, aes(wt, mpg))
# A basic scatter plot
p + geom_point(size = 4)
# Using the colour aesthetic
p + geom_point(aes(colour = factor(cyl)), size = 4)
# Using the shape aesthetic
p + geom_point(aes(shape = factor(cyl)), size = 4)

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

# Using linetypes
ggplot(economics_long, aes(date, value01)) +
  geom_line(aes(linetype = variable))

# Multiple groups with one aesthetic
p <- ggplot(nlme::Oxboys, aes(age, height))
# The default is not sufficient here. A single line tries to connect all
# the observations.
p + geom_line()
# To fix this, use the group aesthetic to map a different line for each
# subject.
p + geom_line(aes(group = Subject))

# Different groups on different layers
p <- p + 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
p + 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
p + geom_smooth(aes(group = 1), size = 2, method = "lm", se = FALSE)

# Overriding the default grouping
# Sometimes 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.
p <- ggplot(nlme::Oxboys, aes(Occasion, height)) + geom_boxplot()
p
# 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)
p + geom_line(aes(group = Subject), colour = "blue")
```

aes_linetype_size_shape *Differentiation related aesthetics: linetype, size, shape*

Description

The linetype, linewidth, size, and shape aesthetics modify the appearance of lines and/or points. They also apply to the outlines of polygons (linetype and linewidth) or to text (size).

Linetype

The linetype aesthetic can be specified with either an integer (0-6), a name (0 = blank, 1 = solid, 2 = dashed, 3 = dotted, 4 = dotdash, 5 = longdash, 6 = twodash), a mapping to a discrete variable, or a string of an even number (up to eight) of hexadecimal digits which give the lengths in consecutive positions in the string. See examples for a hex string demonstration.

Linewidth and stroke

The linewidth aesthetic sets the widths of lines, and can be specified with a numeric value (for historical reasons, these units are about 0.75 millimetres). Alternatively, they can also be set via mapping to a continuous variable. The stroke aesthetic serves the same role for points, but is distinct for discriminating points from lines in geoms such as geom_pointrange().

Size

The size aesthetic control the size of points and text, and can be specified with a numerical value (in millimetres) or via a mapping to a continuous variable.

Shape

The shape aesthetic controls the symbols of points, and can be specified with an integer (between 0 and 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, or a mapping to a discrete variable. Symbols and filled shapes are described in the examples below.

See Also

- geom_line() and geom_point() for geoms commonly used with these aesthetics.
- aes_group_order() for using linetype, size, or shape for grouping.
- Scales that can be used to modify these aesthetics: scale_linetype(), scale_linewidth(), scale_size(), and scale_shape().
- Run vignette("ggplot2-specs") to see an overview of other aesthetics that can be modified.

Other aesthetics documentation: aes(), aes_colour_fill_alpha, aes_group_order, aes_position

Examples

```
df <- data.frame(x = 1:10 , y = 1:10)
p <- ggplot(df, aes(x, y))
p + geom_line(linetype = 2)
p + 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.
p + geom_line(linetype = "3313")

# Mapping line type from a grouping variable
ggplot(economics_long, aes(date, value01)) +
  geom_line(aes(linetype = variable))

# Linewidth examples
ggplot(economics, aes(date, unemploy)) +
  geom_line(linewidth = 2, lineend = "round")
ggplot(economics, aes(date, unemploy)) +
  geom_line(aes(linewidth = uempmed), lineend = "round")

# Size examples
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
p + geom_point()
p + geom_point(shape = 5)
p + geom_point(shape = "k", size = 3)
p + geom_point(shape = ".")
p + geom_point(shape = NA)
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)
p <- ggplot(df2, aes(x, y))
p + geom_point(aes(shape = z), size = 4) +
  scale_shape_identity()
# While all symbols have a foreground colour, symbols 19-25 also take a
# background colour (fill)
p + geom_point(aes(shape = z), size = 4, colour = "Red") +
  scale_shape_identity()
p + geom_point(aes(shape = z), size = 4, colour = "Red", fill = "Black") +
  scale_shape_identity()
```

`aes_position` *Position related aesthetics: x, y, xmin, xmax, ymin, ymax, xend, yend*

Description

The following aesthetics can be used to specify the position of elements: `x`, `y`, `xmin`, `xmax`, `ymin`, `ymax`, `xend`, `yend`.

Details

`x` and `y` define the locations of points or of positions along a line or path.

`x`, `y` and `xend`, `yend` define the starting and ending points of segment and curve geometries.

`xmin`, `xmax`, `ymin` and `ymax` can be used to specify the position of annotations and to represent rectangular areas.

In addition, there are position aesthetics that are contextual to the geometry that they're used in. These are `xintercept`, `yintercept`, `xmin_final`, `ymin_final`, `xmax_final`, `ymax_final`, `xlower`, `lower`, `xmiddle`, `middle`, `xupper`, `upper`, `x0` and `y0`. Many of these are used and automatically computed in `geom_boxplot()`.

Relation to `width` and `height`:

The position aesthetics mentioned above like `x` and `y` are all location based. The `width` and `height` aesthetics are closely related length based aesthetics, but are not position aesthetics. Consequently, `x` and `y` aesthetics respond to scale transformations, whereas the length based `width` and `height` aesthetics are not transformed by scales. For example, if we have the pair `x = 10, width = 2`, that gets translated to the locations `xmin = 9, xmax = 11` when using the default identity scales. However, the same pair becomes `xmin = 1, xmax = 100` when using log10 scales, as `width = 2` in log10-space spans a 100-fold change.

See Also

- Geoms that commonly use these aesthetics: `geom_crossbar()`, `geom_curve()`, `geom_errorbar()`, `geom_line()`, `geom_linerange()`, `geom_path()`, `geom_point()`, `geom_pointrange()`, `geom_rect()`, `geom_segment()`
- Scales that can be used to modify positions: `scale_continuous()`, `scale_discrete()`, `scale_binned()`, `scale_date()`.
- See also `annotate()` for placing annotations.

Other aesthetics documentation: `aes()`, `aes_colour_fill_alpha`, `aes_group_order`, `aes_linetype_size_shape`

Examples

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

# Using annotate
p <- ggplot(mtcars, aes(x = wt, y = mpg)) + geom_point()
p
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")
# from base R:
set.seed(1)
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 = x, y = Freq)) +
  geom_segment(aes(yend = 0, xend = x), size = 10)
```

`annotate` *Create an annotation layer*

Description

This function adds geoms to a plot, but unlike a typical 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 arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

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.

Unsupported geoms

Due to their special nature, reference line geoms `geom_abline()`, `geom_hline()`, and `geom_vline()` can't be used with `annotate()`. You can use these geoms directly for annotations.

See Also

The custom annotations section of the online ggplot2 book.

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 = 2.5, linewidth = 1.5)

p + annotate("text", x = 2:3, y = 20:21, label = c("my label", "label 2"))

p + annotate("text", x = 4, y = 25, label = "italic(R) ^ 2 == 0.75",
  parse = TRUE)
p + annotate("text", x = 4, y = 25,
  label = "paste(italic(R) ^ 2, \" = .75\")", parse = TRUE)
```

`annotation_borders` *Create a layer of map borders*

Description

This is a quick and dirty way to get map data (from the **maps** package) onto your plot. This is a good place to start if you need some crude reference lines, but you'll typically want something more sophisticated for communication graphics.

Usage

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

borders(...) # Deprecated
```

Arguments

`database`	map data, see `maps::map()` for details
`regions`	map region
`fill`	fill colour
`colour`	border colour
`xlim, ylim`	latitudinal and longitudinal ranges for extracting map polygons, see `maps::map()` for details.
`...`	Arguments passed on to `geom_polygon`

`lineend` Line end style (round, butt, square).

`linejoin` Line join style (round, mitre, bevel).

`linemitre` Line mitre limit (number greater than 1).

`rule` Either `"evenodd"` or `"winding"`. If polygons with holes are being drawn (using the `subgroup` aesthetic) this argument defines how the hole coordinates are interpreted. See the examples in `grid::pathGrob()` for an explanation.

`mapping` Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.

`data` The data to be displayed in this layer. There are three options:
If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`.
A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created.
A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`stat` The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following:

- A `Stat` ggproto subclass, for example `StatCount`.
- A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.
- For more information and other ways to specify the stat, see the layer stat documentation.

`position` A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`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. `annotation_borders()`.

`na.rm` If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

Examples

```
if (require("maps")) {
data(us.cities)
capitals <- subset(us.cities, capital == 2)
ggplot(capitals, aes(long, lat)) +
  annotation_borders("state") +
  geom_point(aes(size = pop)) +
  scale_size_area() +
  coord_quickmap()
}

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

`annotation_custom` *Annotation: 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 (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

```
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

```
# Dummy plot
df <- data.frame(x = 1:10, y = 1:10)
base <- ggplot(df, aes(x, y)) +
  geom_blank() +
  theme_bw()

# Full panel annotation
base + annotation_custom(
  grob = grid::roundrectGrob(),
  xmin = -Inf, xmax = Inf, ymin = -Inf, ymax = Inf
)

# Inset plot
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)
```

`annotation_logticks` *Annotation: log tick marks*

Description

[Superseded]

This function is superseded by using `guide_axis_logticks()`.

This annotation adds log tick marks with diminishing spacing. These tick marks probably make sense only for base 10.

Usage

```
annotation_logticks(
  base = 10,
  sides = "bl",
  outside = FALSE,
  scaled = TRUE,
  short = unit(0.1, "cm"),
  mid = unit(0.2, "cm"),
  long = unit(0.3, "cm"),
  colour = "black",
  linewidth = 0.5,
  linetype = 1,
  alpha = 1,
  color = NULL,
  ...,
  size = deprecated()
)
```

Arguments

`base`	the base of the log (default 10)
`sides`	a string that controls which sides of the plot the log ticks appear on. It can be set to a string containing any of `"trbl"`, for top, right, bottom, and left.
`outside`	logical that controls whether to move the log ticks outside of the plot area. Default is off (`FALSE`). You will also need to use `coord_cartesian(clip = "off")`. See examples.
`scaled`	is the data already log-scaled? This should be `TRUE` (default) when the data is already transformed with `log10()` or when using `scale_y_log10()`. It should be `FALSE` when using `coord_transform(y = "log10")`.
`short`	a `grid::unit()` object specifying the length of the short tick marks
`mid`	a `grid::unit()` object specifying the length of the middle tick marks. In base 10, these are the "5" ticks.
`long`	a `grid::unit()` object specifying the length of the long tick marks. In base 10, these are the "1" (or "10") ticks.

`colour`	Colour of the tick marks.
`linewidth`	Thickness of tick marks, in mm.
`linetype`	Linetype of tick marks (`solid`, `dashed`, etc.)
`alpha`	The transparency of the tick marks.
`color`	An alias for `colour`.
`...`	Other parameters passed on to the layer
`size`	**[Deprecated]**

See Also

`scale_y_continuous()`, `scale_y_log10()` for log scale transformations.

`coord_transform()` for log coordinate transformations.

Examples

```
# 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", \(x) 10^x),
   labels = scales::trans_format("log10", scales::math_format(10^.x))
 ) +
 scale_y_log10(
   breaks = scales::trans_breaks("log10", \(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

a + annotation_logticks(sides = "lr", outside = TRUE) +
 coord_cartesian(clip = "off")  # Ticks outside plot

# 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::label_math(10^.x)) +
 scale_y_continuous(name = "brain", labels = scales::label_math(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_transform(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: a map*

Description

Display a fixed map on a plot. This function predates the `geom_sf()` framework and does not work with sf geometry columns as input. However, it can be used in conjunction with `geom_sf()` layers and/or `coord_sf()` (see examples).

Usage

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

Arguments

`map`	Data frame representing a map. See `geom_map()` for details.
`...`	Other arguments used to modify visual parameters, such as `colour` or `fill`.

Examples

```
## Not run:
if (requireNamespace("maps", quietly = TRUE)) {
# location of cities in North Carolina
df <- data.frame(
  name = c("Charlotte", "Raleigh", "Greensboro"),
  lat = c(35.227, 35.772, 36.073),
  long = c(-80.843, -78.639, -79.792)
)

p <- ggplot(df, aes(x = long, y = lat)) +
  annotation_map(
    map_data("state"),
    fill = "antiquewhite", colour = "darkgrey"
  ) +
  geom_point(color = "blue") +
  geom_text(
    aes(label = name),
    hjust = 1.105, vjust = 1.05, color = "blue"
  )

# use without coord_sf() is possible but not recommended
p + xlim(-84, -76) + ylim(34, 37.2)

if (requireNamespace("sf", quietly = TRUE)) {
# use with coord_sf() for appropriate projection
p +
  coord_sf(
    crs = sf::st_crs(3347),
    default_crs = sf::st_crs(4326),  # data is provided as long-lat
    xlim = c(-84, -76),
    ylim = c(34, 37.2)
  )

# you can mix annotation_map() and geom_sf()
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
p +
  geom_sf(
    data = nc, inherit.aes = FALSE,
    fill = NA, color = "black", linewidth = 0.1
  ) +
  coord_sf(crs = sf::st_crs(3347), default_crs = sf::st_crs(4326))
}}
## End(Not run)
```

`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). This is useful for adding bitmap images.

Usage

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

Arguments

`raster`	raster object to display, may be an `array` or a `nativeRaster`
`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.

Examples

```
# 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()

rainbow2 <- matrix(hcl(seq(0, 360, length.out = 10), 80, 70), nrow = 1)
ggplot(mtcars, aes(mpg, wt)) +
  annotation_raster(rainbow2, -Inf, Inf, -Inf, Inf) +
  geom_point()
rainbow2 <- matrix(hcl(seq(0, 360, length.out = 10), 80, 70), nrow = 1)
ggplot(mtcars, aes(mpg, wt)) +
  annotation_raster(rainbow2, -Inf, Inf, -Inf, Inf, interpolate = TRUE) +
  geom_point()
```

`autolayer` *Create a ggplot layer appropriate to a particular data type*

Description

`autolayer()` uses ggplot2 to draw a particular layer for an object of a particular class in a single command. This defines the S3 generic that other classes and packages can extend.

Usage

```
autolayer(object, ...)
```

Arguments

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

Value

a ggplot layer

See Also

Other plotting automation topics: `automatic_plotting`, `autoplot()`, `fortify()`

`automatic_plotting` *Tailoring plots to particular data types*

Description

There are three functions to make plotting particular data types easier: `autoplot()`, `autolayer()` and `fortify()`. These are S3 generics for which other packages can write methods to display classes of data. The three functions are complementary and allow different levels of customisation. Below we'll explore implementing this series of methods to automate plotting of some class.

Let's suppose we are writing a packages that has a class called 'my_heatmap', that wraps a matrix and we'd like users to easily plot this heatmap.

```
my_heatmap <- function(...) {
  m <- matrix(...)
  class(m) <- c("my_heatmap", class(m))
  m
}

my_data <- my_heatmap(volcano)
```

Automatic data shaping

One of the things we have to do is ensure that the data is shaped in the long format so that it is compatible with ggplot2. This is the job of the `fortify()` function. Because 'my_heatmap' wraps a matrix, we can let the fortify method 'melt' the matrix to a long format. If your data is already based on a long-format `<data.frame>`, you can skip implementing a `fortify()` method.

```
fortify.my_heatmap <- function(model, ...) {
  data.frame(
    row = as.vector(row(model)),
    col = as.vector(col(model)),
    value = as.vector(model)
  )
}

fortify(my_data)
```

When you have implemented the `fortify()` method, it should be easier to construct a plot with the data than with the matrix.

```
ggplot(my_data, aes(x = col, y = row, fill = value)) +
  geom_raster()
```

Automatic layers

A next step in automating plotting of your data type is to write an `autolayer()` method. These are typically wrappers around geoms or stats that automatically set aesthetics or other parameters. If you haven't implemented a `fortify()` method for your data type, you might have to reshape the data in `autolayer()`.

If you require multiple layers to display your data type, you can use an `autolayer()` method that constructs a list of layers, which can be added to a plot.

```
autolayer.my_heatmap <- function(object, ...) {
  geom_raster(
    mapping = aes(x = col, y = row, fill = value),
    data = object,
    ...,
    inherit.aes = FALSE
  )
}

ggplot() + autolayer(my_data)
```

As a quick tip: if you define a mapping in `autolayer()`, you might want to set `inherit.aes = FALSE` to not have aesthetics set in other layers interfere with your layer.

Automatic plots

The last step in automating plotting is to write an `autoplot()` method for your data type. The expectation is that these return a complete plot. In the example below, we're exploiting the `autolayer()` method that we have already written to make a complete plot.

```
autoplot.my_heatmap <- function(object, ..., option = "magma") {
  ggplot() +
    autolayer(my_data) +
    scale_fill_viridis_c(option = option) +
    theme_void()
}

autoplot(my_data)
```

If you don't have a wish to implement a base R plotting method, you can set the plot method for your class to the autoplot method.

```
plot.my_heatmap <- autoplot.my_heatmap
plot(my_data)
```

See Also

Other plotting automation topics: `autolayer()`, `autoplot()`, `fortify()`

`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

Other plotting automation topics: `autolayer()`, `automatic_plotting`, `fortify()`

`CoordSf` *Visualise sf objects*

Description

This set of geom, stat, and coord are used to visualise simple feature (sf) objects. For simple plots, you will only need `geom_sf()` as it uses `stat_sf()` and adds `coord_sf()` for you. `geom_sf()` is an unusual geom because it will draw different geometric objects depending on what simple features are present in the data: you can get points, lines, or polygons. For text and labels, you can use `geom_sf_text()` and `geom_sf_label()`.

Usage

```
coord_sf(
  xlim = NULL,
  ylim = NULL,
  expand = TRUE,
  crs = NULL,
  default_crs = NULL,
  datum = sf::st_crs(4326),
  label_graticule = waiver(),
  label_axes = waiver(),
  lims_method = "cross",
  ndiscr = 100,
  default = FALSE,
  clip = "on",
  reverse = "none"
)

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

geom_sf_label(
  mapping = aes(),
  data = NULL,
  stat = "sf_coordinates",
  position = "nudge",
  ...,
  parse = FALSE,
  label.padding = unit(0.25, "lines"),
  label.r = unit(0.15, "lines"),
  label.size = deprecated(),
  border.colour = NULL,
  border.color = NULL,
  text.colour = NULL,
  text.color = NULL,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE,
  fun.geometry = NULL
)

geom_sf_text(
  mapping = aes(),
  data = NULL,
  stat = "sf_coordinates",
  position = "nudge",
  ...,
  parse = FALSE,
  check_overlap = FALSE,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE,
  fun.geometry = NULL
)

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

Arguments

`xlim, ylim`	Limits for the x and y axes. These limits are specified in the units of the default CRS. By default, this means projected coordinates (`default_crs = NULL`). How limit specifications translate into the exact region shown on the plot can be confusing when non-linear or rotated coordinate systems are used as the default crs. First, different methods can be preferable under different conditions. See parameter `lims_method` for details. Second, specifying limits along only one direction can affect the automatically generated limits along the other direction. Therefore, it is best to always specify limits for both x and y. Third, specifying limits via position scales or `xlim()`/`ylim()` is strongly discouraged, as it can result in data points being dropped from the plot even though they would be visible in the final plot region.
`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`. Giving a logical vector will separately control the expansion for the four directions (top, left, bottom and right). The `expand` argument will be recycled to length 4 if necessary. Alternatively, can be a named logical vector to control a single direction, e.g. `expand = c(bottom = FALSE)`.
`crs`	The coordinate reference system (CRS) into which all data should be projected before plotting. If not specified, will use the CRS defined in the first sf layer of the plot.
`default_crs`	The default CRS to be used for non-sf layers (which don't carry any CRS information) and scale limits. The default value of `NULL` means that the setting for `crs` is used. This implies that all non-sf layers and scale limits are assumed to be specified in projected coordinates. A useful alternative setting is `default_crs = sf::st_crs(4326)`, which means x and y positions are interpreted as longitude and latitude, respectively, in the World Geodetic System 1984 (WGS84).
`datum`	CRS that provides datum to use when generating graticules.

`label_graticule`	Character vector indicating which graticule lines should be labeled where. Meridians run north-south, and the letters `"N"` and `"S"` indicate that they should be labeled on their north or south end points, respectively. Parallels run east-west, and the letters `"E"` and `"W"` indicate that they should be labeled on their east or west end points, respectively. Thus, `label_graticule = "SW"` would label meridians at their south end and parallels at their west end, whereas `label_graticule = "EW"` would label parallels at both ends and meridians not at all. Because meridians and parallels can in general intersect with any side of the plot panel, for any choice of `label_graticule` labels are not guaranteed to reside on only one particular side of the plot panel. Also, `label_graticule` can cause labeling artifacts, in particular if a graticule line coincides with the edge of the plot panel. In such circumstances, `label_axes` will generally yield better results and should be used instead. This parameter can be used alone or in combination with `label_axes`.
`label_axes`	Character vector or named list of character values specifying which graticule lines (meridians or parallels) should be labeled on which side of the plot. Meridians are indicated by `"E"` (for East) and parallels by `"N"` (for North). Default is `"--EN"`, which specifies (clockwise from the top) no labels on the top, none on the right, meridians on the bottom, and parallels on the left. Alternatively, this setting could have been specified with `list(bottom = "E", left = "N")`. This parameter can be used alone or in combination with `label_graticule`.
`lims_method`	Method specifying how scale limits are converted into limits on the plot region. Has no effect when `default_crs = NULL`. For a very non-linear CRS (e.g., a perspective centered around the North pole), the available methods yield widely differing results, and you may want to try various options. Methods currently implemented include `"cross"` (the default), `"box"`, `"orthogonal"`, and `"geometry_bbox"`. For method `"cross"`, limits along one direction (e.g., longitude) are applied at the midpoint of the other direction (e.g., latitude). This method avoids excessively large limits for rotated coordinate systems but means that sometimes limits need to be expanded a little further if extreme data points are to be included in the final plot region. By contrast, for method `"box"`, a box is generated out of the limits along both directions, and then limits in projected coordinates are chosen such that the entire box is visible. This method can yield plot regions that are too large. Finally, method `"orthogonal"` applies limits separately along each axis, and method `"geometry_bbox"` ignores all limit information except the bounding boxes of any objects in the `geometry` aesthetic.
`ndiscr`	Number of segments to use for discretising graticule lines; try increasing this number when graticules look incorrect.
`default`	Is this the default coordinate system? If `FALSE` (the default), then replacing this coordinate system with another one creates a message alerting the user that the coordinate system is being replaced. If `TRUE`, that warning is suppressed.
`clip`	Should drawing be clipped to the extent of the plot panel? A setting of `"on"` (the default) means yes, and a setting of `"off"` means no. In most cases, the default of `"on"` should not be changed, as setting `clip = "off"` can cause unexpected results. It allows drawing of data points anywhere on the plot, including in the plot margins. If limits are set via `xlim` and `ylim` and some data points fall

	outside those limits, then those data points may show up in places such as the axes, the legend, the plot title, or the plot margins.
`reverse`	A string giving which directions to reverse. `"none"` (default) keeps directions as is. `"x"` and `"y"` can be used to reverse their respective directions. `"xy"` can be used to reverse both directions.
`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. You can also set this to one of "polygon", "line", and "point" to override the default legend.

`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. `annotation_borders()`.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`parse`	If `TRUE`, the labels will be parsed into expressions and displayed as described in `?plotmath`.
`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`	**[Deprecated]** Replaced by the `linewidth` aesthetic. Size of label border, in mm.
`border.colour, border.color`	Colour of label border. When `NULL` (default), the `colour` aesthetic determines the colour of the label border. `border.color` is an alias for `border.colour`.
`text.colour, text.color`	Colour of the text. When `NULL` (default), the `colour` aesthetic determines the colour of the text. `text.color` is an alias for `text.colour`.
`fun.geometry`	A function that takes a `sfc` object and returns a `sfc_POINT` with the same length as the input. If `NULL`, `function(x) sf::st_point_on_surface(sf::st_zm(x))` will be used. Note that the function may warn about the incorrectness of the result if the data is not projected, but you can ignore this except when you really care about the exact locations.
`check_overlap`	If `TRUE`, text that overlaps previous text in the same layer will not be plotted. `check_overlap` happens at draw time and in the order of the data. Therefore data should be arranged by the label column before calling `geom_text()`. Note that this argument is not supported by `geom_label()`.

`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`. • A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`. • For more information and other ways to specify the geom, see the layer geom documentation.

Geometry aesthetic

`geom_sf()` uses a unique aesthetic: `geometry`, giving an column of class `sfc` containing simple features data. There are three ways to supply the `geometry` aesthetic:

- Do nothing: by default `geom_sf()` assumes it is stored in the `geometry` column.
- Explicitly pass an `sf` object to the `data` argument. This will use the primary geometry column, no matter what it's called.
- Supply your own using `aes(geometry = my_column)`

Unlike other aesthetics, `geometry` will never be inherited from the plot.

CRS

`coord_sf()` ensures that all layers use a common CRS. You can either specify it using the `crs` param, or `coord_sf()` will take it from the first layer that defines a CRS.

Combining sf layers and regular geoms

Most regular geoms, such as `geom_point()`, `geom_path()`, `geom_text()`, `geom_polygon()` etc. will work fine with `coord_sf()`. However when using these geoms, two problems arise. First, what CRS should be used for the x and y coordinates used by these non-sf geoms? The CRS applied to non-sf geoms is set by the `default_crs` parameter, and it defaults to `NULL`, which means positions for non-sf geoms are interpreted as projected coordinates in the coordinate system set by the `crs` parameter. This setting allows you complete control over where exactly items are placed on the plot canvas, but it may require some understanding of how projections work and how to generate data in projected coordinates. As an alternative, you can set `default_crs = sf::st_crs(4326)`, the World Geodetic System 1984 (WGS84). This means that x and y positions are interpreted as longitude and latitude, respectively. You can also specify any other valid CRS as the default CRS for non-sf geoms.

The second problem that arises for non-sf geoms is how straight lines should be interpreted in projected space when `default_crs` is not set to `NULL`. The approach `coord_sf()` takes is to break straight lines into small pieces (i.e., segmentize them) and then transform the pieces into projected coordinates. For the default setting where x and y are interpreted as longitude and latitude, this approach means that horizontal lines follow the parallels and vertical lines follow the meridians. If you need a different approach to handling straight lines, then you should manually segmentize and project coordinates and generate the plot in projected coordinates.

See Also

The simple feature maps section of the online ggplot2 book.

`stat_sf_coordinates()`

Examples

```
if (requireNamespace("sf", quietly = TRUE)) {
nc <- sf::st_read(system.file("shape/nc.shp", package = "sf"), quiet = TRUE)
ggplot(nc) +
  geom_sf(aes(fill = AREA))

# If not supplied, coord_sf() will take the CRS from the first layer
# and automatically transform all other layers to use that CRS. This
# ensures that all data will correctly line up
nc_3857 <- sf::st_transform(nc, 3857)
ggplot() +
  geom_sf(data = nc) +
  geom_sf(data = nc_3857, colour = "red", fill = NA)

# Unfortunately if you plot other types of feature you'll need to use
# show.legend to tell ggplot2 what type of legend to use
nc_3857$mid <- sf::st_centroid(nc_3857$geometry)
ggplot(nc_3857) +
  geom_sf(colour = "white") +
  geom_sf(aes(geometry = mid, size = AREA), show.legend = "point")

# You can also use layers with x and y aesthetics. To have these interpreted
# as longitude/latitude you need to set the default CRS in coord_sf()
ggplot(nc_3857) +
  geom_sf() +
  annotate("point", x = -80, y = 35, colour = "red", size = 4) +
  coord_sf(default_crs = sf::st_crs(4326))

# To add labels, use geom_sf_label().
ggplot(nc_3857[1:3, ]) +
   geom_sf(aes(fill = AREA)) +
   geom_sf_label(aes(label = NAME))
}

# Thanks to the power of sf, a geom_sf nicely handles varying projections
# setting the aspect ratio correctly.
if (requireNamespace('maps', quietly = TRUE)) {
library(maps)
world1 <- sf::st_as_sf(map('world', plot = FALSE, fill = TRUE))
ggplot() + geom_sf(data = world1)

world2 <- sf::st_transform(
  world1,
  "+proj=laea +y_0=0 +lon_0=155 +lat_0=-90 +ellps=WGS84 +no_defs"
)
ggplot() + geom_sf(data = world2)
}
```

`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

```
coord_cartesian(
  xlim = NULL,
  ylim = NULL,
  expand = TRUE,
  default = FALSE,
  clip = "on",
  reverse = "none",
  ratio = NULL
)
```

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`. Giving a logical vector will separately control the expansion for the four directions (top, left, bottom and right). The `expand` argument will be recycled to length 4 if necessary. Alternatively, can be a named logical vector to control a single direction, e.g. `expand = c(bottom = FALSE)`.
`default`	Is this the default coordinate system? If `FALSE` (the default), then replacing this coordinate system with another one creates a message alerting the user that the coordinate system is being replaced. If `TRUE`, that warning is suppressed.
`clip`	Should drawing be clipped to the extent of the plot panel? A setting of `"on"` (the default) means yes, and a setting of `"off"` means no. In most cases, the default of `"on"` should not be changed, as setting `clip = "off"` can cause unexpected results. It allows drawing of data points anywhere on the plot, including in the plot margins. If limits are set via `xlim` and `ylim` and some data points fall outside those limits, then those data points may show up in places such as the axes, the legend, the plot title, or the plot margins.
`reverse`	A string giving which directions to reverse. `"none"` (default) keeps directions as is. `"x"` and `"y"` can be used to reverse their respective directions. `"xy"` can be used to reverse both directions.
`ratio`	aspect ratio, expressed as `y / x`. Can be `NULL` (default) to not use an aspect ratio. Using `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.

Examples

```
# 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)

# Using a fixed ratio: 1 y-axis unit is 100 x-axis units
# Plot window can be resized and aspect ratio will be maintained
p + coord_cartesian(ratio = 100)

# You can see the same thing with this 2d histogram
d <- ggplot(diamonds, aes(carat, price)) +
  stat_bin_2d(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 "aspect ratio"*

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 `MASS::eqscplot()`, but it works for all types of graphics.

Usage

```
coord_fixed(ratio = 1, ...)
```

Arguments

`ratio`	aspect ratio, expressed as `y / x`. Can be `NULL` (default) to not use an aspect ratio. Using `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.
`...`	Arguments passed on to `coord_cartesian`

`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`. Giving a logical vector will separately control the expansion for the four directions (top, left, bottom and right). The `expand` argument will be recycled to length 4 if necessary. Alternatively, can be a named logical vector to control a single direction, e.g. `expand = c(bottom = FALSE)`.

`default` Is this the default coordinate system? If `FALSE` (the default), then replacing this coordinate system with another one creates a message alerting the user that the coordinate system is being replaced. If `TRUE`, that warning is suppressed.

`clip` Should drawing be clipped to the extent of the plot panel? A setting of `"on"` (the default) means yes, and a setting of `"off"` means no. In most cases, the default of `"on"` should not be changed, as setting `clip = "off"` can cause unexpected results. It allows drawing of data points anywhere on the plot, including in the plot margins. If limits are set via `xlim` and `ylim` and some data points fall outside those limits, then those data points may show up in places such as the axes, the legend, the plot title, or the plot margins.

`reverse` A string giving which directions to reverse. `"none"` (default) keeps directions as is. `"x"` and `"y"` can be used to reverse their respective directions. `"xy"` can be used to reverse both directions.

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)
p + coord_fixed(xlim = c(15, 30))
```

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

`coord_flip` *Cartesian coordinates with x and y flipped*

Description

[Superseded]

This function is superseded because in many cases, `coord_flip()` can easily be replaced by swapping the x and y aesthetics, or optionally setting the `orientation` argument in geom and stat layers.

`coord_flip()` is useful for geoms and statistics that do not support the `orientation` setting, and converting the display of y conditional on x, to x conditional on y.

Usage

```
coord_flip(xlim = NULL, ylim = NULL, expand = TRUE, clip = "on")
```

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`. Giving a logical vector will separately control the expansion for the four directions (top, left, bottom and right). The `expand` argument will be recycled to length 4 if necessary. Alternatively, can be a named logical vector to control a single direction, e.g. `expand = c(bottom = FALSE)`.
`clip`	Should drawing be clipped to the extent of the plot panel? A setting of `"on"` (the default) means yes, and a setting of `"off"` means no. In most cases, the default of `"on"` should not be changed, as setting `clip = "off"` can cause unexpected results. It allows drawing of data points anywhere on the plot, including in the plot margins. If limits are set via `xlim` and `ylim` and some data points fall outside those limits, then those data points may show up in places such as the axes, the legend, the plot title, or the plot margins.

Details

Coordinate systems interact with many parts of the plotting system. You can expect the following for `coord_flip()`:

- It does *not* change the facet order in `facet_grid()` or `facet_wrap()`.
- The `scale_x_*()` functions apply to the vertical direction, whereas `scale_y_*()` functions apply to the horizontal direction. The same holds for the `xlim` and `ylim` arguments of `coord_flip()` and the `xlim()` and `ylim()` functions.
- The x-axis theme settings, such as `axis.line.x` apply to the horizontal direction. The y-axis theme settings, such as `axis.text.y` apply to the vertical direction.

Examples

```
# The preferred method of creating horizontal instead of vertical boxplots
ggplot(diamonds, aes(price, cut)) +
  geom_boxplot()

# Using `coord_flip()` to make the same plot
ggplot(diamonds, aes(cut, price)) +
  geom_boxplot() +
  coord_flip()

# With swapped aesthetics, the y-scale controls the left axis
ggplot(diamonds, aes(y = carat)) +
  geom_histogram() +
  scale_y_reverse()

# In `coord_flip()`, the x-scale controls the left axis
ggplot(diamonds, aes(carat)) +
  geom_histogram() +
  coord_flip() +
  scale_x_reverse()

# In line and area plots, swapped aesthetics require an explicit orientation
df <- data.frame(a = 1:5, b = (1:5) ^ 2)
ggplot(df, aes(b, a)) +
  geom_area(orientation = "y")

# The same plot with `coord_flip()`
ggplot(df, aes(a, b)) +
  geom_area() +
  coord_flip()
```

`coord_map` *Map projections*

Description

[Superseded]

`coord_map()` projects a portion of the earth, which is approximately spherical, onto a flat 2D plane using any projection defined by the `mapproj` package. Map projections do not, in general, preserve straight lines, so this requires considerable computation. `coord_quickmap()` is a quick approximation that does preserve straight lines. It works best for smaller areas closer to the equator.

Both `coord_map()` and `coord_quickmap()` are superseded by `coord_sf()`, and should no longer be used in new code. All regular (non-sf) geoms can be used with `coord_sf()` by setting the default coordinate system via the `default_crs` argument. See also the examples for `annotation_map()` and `geom_map()`.

Usage

```
coord_map(
  projection = "mercator",
  ...,
  parameters = NULL,
  orientation = NULL,
  xlim = NULL,
  ylim = NULL,
  clip = "on"
)

coord_quickmap(xlim = NULL, ylim = NULL, expand = TRUE, clip = "on")
```

Arguments

`projection`	projection to use, see `mapproj::mapproject()` for list
`..., parameters`	Other arguments passed on to `mapproj::mapproject()`. Use `...` for named parameters to the projection, and `parameters` for unnamed parameters. `...` is ignored if the `parameters` argument is present.
`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 `mapproj::mapproject()` for more information.
`xlim, ylim`	Manually specific x/y limits (in degrees of longitude/latitude)
`clip`	Should drawing be clipped to the extent of the plot panel? A setting of `"on"` (the default) means yes, and a setting of `"off"` means no. For details, please see `coord_cartesian()`.
`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`. Giving a logical vector will separately control the expansion for the four directions (top, left, bottom and right). The `expand` argument will be recycled to length 4 if necessary. Alternatively, can be a named logical vector to control a single direction, e.g. `expand = c(bottom = FALSE)`.

Details

Map projections must 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 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, and is much faster (particularly for complex plots like `geom_tile()`) at the expense of correctness.

See Also

The polygon maps section of the online ggplot2 book.

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
}

if (require("maps")) {
# With correct mercator projection
nzmap + coord_map()
}

if (require("maps")) {
# With the aspect ratio approximation
nzmap + coord_quickmap()
}

if (require("maps")) {
# Other projections
nzmap + coord_map("azequalarea", orientation = c(-36.92, 174.6, 0))
}

if (require("maps")) {
states <- map_data("state")
usamap <- ggplot(states, aes(long, lat, group = group)) +
  geom_polygon(fill = "white", colour = "black")

# Use cartesian coordinates
usamap
}

if (require("maps")) {
# With mercator projection
usamap + coord_map()
}

if (require("maps")) {
# See ?mapproject for coordinate systems and their parameters
usamap + coord_map("gilbert")
}

if (require("maps")) {
# 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")
}
```

```
if (require("maps")) {
usamap + coord_map("conic", lat0 = 30)
}

if (require("maps")) {
usamap + coord_map("bonne", lat0 = 50)
}

## Not run:
if (require("maps")) {
# 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")
}

if (require("maps")) {
# Looking up up at South Pole
worldmap + coord_map("ortho", orientation = c(-90, 0, 0))
}

if (require("maps")) {
# Centered on New York (currently has issues with closing polygons)
worldmap + coord_map("ortho", orientation = c(41, -74, 0))
}

## End(Not run)
```

coord_polar *Polar coordinates*

Description

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

[Superseded]: `coord_polar()` has been in favour of `coord_radial()`.

Usage

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

coord_radial(
  theta = "x",
  start = 0,
  end = NULL,
  thetalim = NULL,
  rlim = NULL,
  expand = TRUE,
  direction = deprecated(),
  clip = "off",
  r.axis.inside = NULL,
  rotate.angle = FALSE,
  inner.radius = 0,
  reverse = "none",
  r_axis_inside = deprecated(),
  rotate_angle = deprecated()
)
```

Arguments

`theta`	variable to map angle to (`x` or `y`)
`start`	Offset of starting point from 12 o'clock in radians. Offset is applied clockwise or anticlockwise depending on value of `direction`.
`direction`	1, clockwise; -1, anticlockwise
`clip`	Should drawing be clipped to the extent of the plot panel? A setting of `"on"` (the default) means yes, and a setting of `"off"` means no. For details, please see `coord_cartesian()`.
`end`	Position from 12 o'clock in radians where plot ends, to allow for partial polar coordinates. The default, `NULL`, is set to `start + 2 * pi`.
`thetalim`, `rlim`	Limits for the theta and r axes.
`expand`	If `TRUE`, the default, adds a small expansion factor to the limits to prevent overlap between data and axes. If `FALSE`, limits are taken directly from the scale.
`r.axis.inside`	One of the following: • `NULL` (default) places the axis next to the panel if `start` and `end` arguments form a full circle and inside the panel otherwise. • `TRUE` to place the radius axis inside the panel. • `FALSE` to place the radius axis next to the panel. • A numeric value, setting a theta axis value at which the axis should be placed inside the panel. Can be given as a length 2 vector to control primary and secondary axis placement separately.
`rotate.angle`	If `TRUE`, transforms the `angle` aesthetic in data in accordance with the computed `theta` position. If `FALSE` (default), no such transformation is performed. Can be useful to rotate text geoms in alignment with the coordinates.
`inner.radius`	A `numeric` between 0 and 1 setting the size of a inner radius hole.
`reverse`	A string giving which directions to reverse. `"none"` (default) keep directions as is. `"theta"` reverses the angle and `"r"` reverses the radius. `"thetar"` reverses both the angle and the radius.
`r_axis_inside`, `rotate_angle`	**[Deprecated]**

Note

In `coord_radial()`, position guides can be defined by using `guides(r = ..., theta = ..., r.sec = ..., theta.sec = ...)`. Note that these guides require `r` and `theta` as available aesthetics. The classic `guide_axis()` can be used for the `r` positions and `guide_axis_theta()` can be used for the `theta` positions. Using the `theta.sec` position is only sensible when `inner.radius > 0`.

See Also

The polar coordinates section of the online ggplot2 book.

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_radial(theta = "y", expand = FALSE)



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

# The bullseye chart
pie + coord_radial(expand = FALSE)

# Hadley's favourite pie chart
df <- data.frame(
  variable = c("does not resemble", "resembles"),
  value = c(20, 80)
)
ggplot(df, aes(x = "", y = value, fill = variable)) +
  geom_col(width = 1) +
  scale_fill_manual(values = c("red", "yellow")) +
  coord_radial("y", start = pi / 3, expand =  FALSE) +
  labs(title = "Pac man")

# Windrose + doughnut plot
if (require("ggplot2movies")) {
movies$rrating <- cut_interval(movies$rating, length = 1)
movies$budgetq <- cut_number(movies$budget, 4)

doh <- ggplot(movies, aes(x = rrating, fill = budgetq))
# Wind rose
doh + geom_bar(width = 1) + coord_radial(expand = FALSE)
# Race track plot
doh + geom_bar(width = 0.9, position = "fill") +
  coord_radial(theta = "y", expand = FALSE)
}

# A partial polar plot
ggplot(mtcars, aes(disp, mpg)) +
  geom_point() +
  coord_radial(start = -0.4 * pi, end = 0.4 * pi, inner.radius = 0.3)

# Similar with coord_cartesian(), you can set limits.
ggplot(mtcars, aes(disp, mpg)) +
  geom_point() +
  coord_radial(
    start = -0.4 * pi,
    end = 0.4 * pi, inner.radius = 0.3,
    thetalim = c(200, 300),
    rlim = c(15, 30),
  )
```

`coord_transform` *Transformed Cartesian coordinate system*

Description

`coord_transform()` 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_transform(
  x = "identity",
  y = "identity",
  xlim = NULL,
  ylim = NULL,
  limx = deprecated(),
  limy = deprecated(),
  clip = "on",
  expand = TRUE,
  reverse = "none"
)

coord_trans(...)
```

Arguments

`x, y`	Transformers for x and y axes or their names.
`xlim, ylim`	Limits for the x and y axes.
`limx, limy`	**[Deprecated]** use `xlim` and `ylim` instead.
`clip`	Should drawing be clipped to the extent of the plot panel? A setting of `"on"` (the default) means yes, and a setting of `"off"` means no. In most cases, the default of `"on"` should not be changed, as setting `clip = "off"` can cause unexpected results. It allows drawing of data points anywhere on the plot, including in the plot margins. If limits are set via `xlim` and `ylim` and some data points fall outside those limits, then those data points may show up in places such as the axes, the legend, the plot title, or the plot margins.
`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`. Giving a logical vector will separately control the expansion for the four directions (top, left, bottom and right). The `expand` argument will be recycled to length 4 if necessary. Alternatively, can be a named logical vector to control a single direction, e.g. `expand = c(bottom = FALSE)`.
`reverse`	A string giving which directions to reverse. `"none"` (default) keeps directions as is. `"x"` and `"y"` can be used to reverse their respective directions. `"xy"` can be used to reverse both directions.
`...`	Arguments forwarded to `coord_transform()`.

Details

Transformations only work with continuous values: see `scales::new_transform()` for list of transformations, and instructions on how to create your own.

The `coord_trans()` function is deprecated in favour of `coord_transform()`.

See Also

The coord transformations section of the online ggplot2 book.

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_transform(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_transform(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_transform(x = scales::transform_exp(10), y = scales::transform_exp(10))

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

# Also works with discrete scales
set.seed(1)
df <- data.frame(a = abs(rnorm(26)),letters)
plot <- ggplot(df,aes(a,letters)) + geom_point()

plot + coord_transform(x = "log10")
plot + coord_transform(x = "sqrt")
```

`cut_interval` *Discretise numeric data into categorical*

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 = "right", ...)
```

Arguments

`x`	numeric vector
`n`	number of intervals to create, OR
`length`	length of each interval
`...`	Arguments passed on to `base::cut.default` `breaks` either a numeric vector of two or more unique cut points or a single number (greater than or equal to 2) giving the number of intervals into which `x` is to be cut. `labels` labels for the levels of the resulting category. By default, labels are constructed using `"(a,b]"` interval notation. If `labels = FALSE`, simple integer codes are returned instead of a factor. `right` logical, indicating if the intervals should be closed on the right (and open on the left) or vice versa. `dig.lab` integer which is used when labels are not given. It determines the number of digits used in formatting the break numbers. `ordered_result` logical: should the result be an ordered factor?
`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()`.

Examples

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

set.seed(1)

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))
table(cut_width(runif(1000), 0.1, labels = FALSE))
```

`diamonds` *Prices of over 50,000 round cut diamonds*

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 D (best) to J (worst)

clarity a measurement of how clear the diamond is (I1 (worst), SI2, SI1, VS2, VS1, VVS2, VVS1, 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 / mean(x, y) = 2 * z / (x + y) (43–79)

table width of top of diamond relative to widest point (43–95) draw_key *Key glyphs for legends*

Description

Each geom has an associated function that draws the key when the geom needs to be displayed in a legend. These functions are called `draw_key_*()`, where `*` stands for the name of the respective key glyph. The key glyphs can be customized for individual geoms by providing a geom with the `key_glyph` argument (see `layer()` or examples below.)

Usage

```
draw_key_point(data, params, size)

draw_key_abline(data, params, size)

draw_key_rect(data, params, size)

draw_key_polygon(data, params, size)

draw_key_blank(data, params, size)

draw_key_boxplot(data, params, size)

draw_key_crossbar(data, params, size)

draw_key_path(data, params, size)

draw_key_vpath(data, params, size)

draw_key_dotplot(data, params, size)

draw_key_linerange(data, params, size)

draw_key_pointrange(data, params, size)

draw_key_smooth(data, params, size)

draw_key_text(data, params, size)

draw_key_label(data, params, size)

draw_key_vline(data, params, size)

draw_key_timeseries(data, params, size)
```

Arguments

`data`	A single row data frame containing the scaled aesthetics to display in this key
`params`	A list of additional parameters supplied to the geom.
`size`	Width and height of key in mm.

Value

A grid grob.

Examples

```
p <- ggplot(economics, aes(date, psavert, color = "savings rate"))
# key glyphs can be specified by their name
p + geom_line(key_glyph = "timeseries")

# key glyphs can be specified via their drawing function
p + geom_line(key_glyph = draw_key_rect)
```

`economics` *US economic time series*

Description

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

Usage

```
economics

economics_long
```

Format

A data frame with 574 rows and 6 variables:

date Month of data collection

pce personal consumption expenditures, in billions of dollars, https://fred.stlouisfed.org/series/PCE

pop total population, in thousands, https://fred.stlouisfed.org/series/POP

psavert personal savings rate, https://fred.stlouisfed.org/series/PSAVERT/

uempmed median duration of unemployment, in weeks, https://fred.stlouisfed.org/series/UEMPMED

unemploy number of unemployed in thousands, https://fred.stlouisfed.org/series/UNEMPLOY

An object of class `tbl_df` (inherits from `tbl`, `data.frame`) with 2870 rows and 4 columns.

expand_limits *Expand the plot limits, using data*

Description

[Superseded]: It is recommended to pass a function to the `limits` argument in scales instead. For example: `scale_x_continuous(limits = ~range(.x, 0))` to include zero.

Sometimes you may want to ensure limits include a single value, for all panels or all plots. This function is a thin wrapper around `geom_blank()` that makes it easy to add such values.

Usage

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

Arguments

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

Examples

```
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)))
```

expansion *Generate expansion vector for scales*

Description

This is a convenience function for generating scale expansion vectors for the expand argument of scale_(x|y)_continuous and scale_(x|y)_discrete. The expansion vectors are used to add some space between the data and the axes.

Usage

```
expansion(mult = 0, add = 0)

expand_scale(mult = 0, add = 0)
```

Arguments

`mult`	vector of multiplicative range expansion factors. If length 1, both the lower and upper limits of the scale are expanded outwards by `mult`. If length 2, the lower limit is expanded by `mult[1]` and the upper limit by `mult[2]`.
`add`	vector of additive range expansion constants. If length 1, both the lower and upper limits of the scale are expanded outwards by `add` units. If length 2, the lower limit is expanded by `add[1]` and the upper limit by `add[2]`.

Examples

```
# No space below the bars but 10% above them
ggplot(mtcars) +
  geom_bar(aes(x = factor(cyl))) +
  scale_y_continuous(expand = expansion(mult = c(0, .1)))

# Add 2 units of space on the left and right of the data
ggplot(subset(diamonds, carat > 2), aes(cut, clarity)) +
  geom_jitter() +
  scale_x_discrete(expand = expansion(add = 2))

# Reproduce the default range expansion used
# when the 'expand' argument is not specified
ggplot(subset(diamonds, carat > 2), aes(cut, price)) +
  geom_jitter() +
  scale_x_discrete(expand = expansion(add = .6)) +
  scale_y_continuous(expand = expansion(mult = .05))
```

`facet_grid` *Lay out panels in a grid*

Description

`facet_grid()` forms a matrix of panels defined by row and column faceting variables. It is most useful when you have two discrete variables, and all combinations of the variables exist in the data. If you have only one variable with many levels, try `facet_wrap()`.

Usage

```
facet_grid(
  rows = NULL,
  cols = NULL,
  scales = "fixed",
  space = "fixed",
  shrink = TRUE,
  labeller = "label_value",
  as.table = TRUE,
  switch = NULL,
  drop = TRUE,
  margins = FALSE,
  axes = "margins",
  axis.labels = "all",
  facets = deprecated()
)
```

Arguments

`rows, cols`	A set of variables or expressions quoted by `vars()` and defining faceting groups on the rows or columns dimension. The variables can be named (the names are passed to `labeller`). For compatibility with the classic interface, `rows` can also be 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).
`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.
`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 `vars(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()`. You can use different labeling functions for different kind of labels, for example use `label_parsed()` for formatting facet labels. `label_value()` is used by default, check it 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.
`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.

`axes`	Determines which axes will be drawn. When `"margins"` (default), axes will be drawn at the exterior margins. `"all_x"` and `"all_y"` will draw the respective axes at the interior panels too, whereas `"all"` will draw all axes at all panels.
`axis.labels`	Determines whether to draw labels for interior axes when the `axes` argument is not `"margins"`. When `"all"` (default), all interior axes get labels. When `"margins"`, only the exterior axes get labels and the interior axes get none. When `"all_x"` or `"all_y"`, only draws the labels at the interior axes in the x- or y-direction respectively.
`facets`	**[Deprecated]** Please use `rows` and `cols` instead.

Layer layout

The `layer(layout)` argument in context of `facet_grid()` can take the following values:

- `NULL` (default) to use the faceting variables to assign panels.
- An integer vector to include selected panels. Panel numbers not included in the integer vector are excluded.
- `"fixed"` to repeat data across every panel.
- `"fixed_rows"` to repeat data across rows.
- `"fixed_cols"` to repeat data across columns.

See Also

The facet grid section of the online ggplot2 book.

Examples

```
p <- ggplot(mpg, aes(displ, cty)) + geom_point()

# Use vars() to supply variables from the dataset:
p + facet_grid(rows = vars(drv))
p + facet_grid(cols = vars(cyl))
p + facet_grid(vars(drv), vars(cyl))

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

# If you combine a facetted dataset with a dataset that lacks those
# faceting variables, the data will be repeated across the missing
# combinations:
df <- data.frame(displ = mean(mpg$displ), cty = mean(mpg$cty))
p +
  facet_grid(cols = vars(cyl)) +
  geom_point(data = df, colour = "red", size = 2)

# When scales are constant, duplicated axes can be shown with
# or without labels
ggplot(mpg, aes(cty, hwy)) +
  geom_point() +
  facet_grid(year ~ drv, axes = "all", axis.labels = "all_x")
```

```
# Free scales -------------------------------------------------------
# You can also choose whether the scales should be constant
# across all panels (the default), or whether they should be allowed
# to vary
mt <- ggplot(mtcars, aes(mpg, wt, colour = factor(cyl))) +
  geom_point()

mt + facet_grid(vars(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. This
# is particularly useful for categorical axes
ggplot(mpg, aes(drv, model)) +
  geom_point() +
  facet_grid(manufacturer ~ ., scales = "free", space = "free") +
  theme(strip.text.y = element_text(angle = 0))

# Margins ----------------------------------------------------------

# Margins can be specified logically (all yes or all no) or for specific
# variables as (character) variable names
mg <- ggplot(mtcars, aes(x = mpg, y = wt)) + geom_point()
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")
```

facet_wrap — *Wrap a 1d ribbon of panels into 2d*

Description

`facet_wrap()` wraps a 1d sequence of panels into 2d. This is generally a better use of screen space than `facet_grid()` because most displays are roughly rectangular.

Usage

```
facet_wrap(
  facets,
  nrow = NULL,
  ncol = NULL,
  scales = "fixed",
  space = "fixed",
  shrink = TRUE,
  labeller = "label_value",
  as.table = TRUE,
  switch = deprecated(),
  drop = TRUE,
  dir = "h",
  strip.position = "top",
  axes = "margins",
  axis.labels = "all"
)
```

Arguments

`facets`	A set of variables or expressions quoted by `vars()` and defining faceting groups on the rows or columns dimension. The variables can be named (the names are passed to `labeller`). For compatibility with the classic interface, can also be 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"`)?
`space`	If `"fixed"` (default), all panels have the same size and the number of rows and columns in the layout can be arbitrary. If `"free_x"`, panels have widths proportional to the length of the x-scale, but the layout is constrained to one row. If `"free_y"`, panels have heights proportional to the length of the y-scale, but the layout is constrained to one column.
`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 `vars(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()`. You can use different labeling functions for different kind of labels, for example use `label_parsed()` for formatting facet labels. `label_value()` is used by default, check it for more details and pointers to other options.
`as.table`	**[Superseded]** The `as.table` argument is now absorbed into the `dir` argument via the two letter options. If `TRUE`, 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.

`dir`	Direction: either `"h"` for horizontal, the default, or `"v"`, for vertical. When `"h"` or `"v"` will be combined with `as.table` to set final layout. Alternatively, a combination of `"t"` (top) or `"b"` (bottom) with `"l"` (left) or `"r"` (right) to set a layout directly. These two letters give the starting position and the first letter gives the growing direction. For example `"rt"` will place the first panel in the top-right and starts filling in panels right-to-left.
`strip.position`	By default, the labels are displayed on the top of the plot. Using `strip.position` it is possible to place the labels on either of the four sides by setting `strip.position = c("top", "bottom", "left", "right")`
`axes`	Determines which axes will be drawn in case of fixed scales. When `"margins"` (default), axes will be drawn at the exterior margins. `"all_x"` and `"all_y"` will draw the respective axes at the interior panels too, whereas `"all"` will draw all axes at all panels.
`axis.labels`	Determines whether to draw labels for interior axes when the scale is fixed and the `axis` argument is not `"margins"`. When `"all"` (default), all interior axes get labels. When `"margins"`, only the exterior axes get labels, and the interior axes get none. When `"all_x"` or `"all_y"`, only draws the labels at the interior axes in the x- or y-direction respectively.

Layer layout

The `layer(layout)` argument in context of `facet_wrap()` can take the following values:

- `NULL` (default) to use the faceting variables to assign panels.
- An integer vector to include selected panels. Panel numbers not included in the integer vector are excluded.
- `"fixed"` to repeat data across every panel.

See Also

The facet wrap section of the online ggplot2 book.

Examples

```
p <- ggplot(mpg, aes(displ, hwy)) + geom_point()

# Use vars() to supply faceting variables:
p + facet_wrap(vars(class))

# Control the number of rows and columns with nrow and ncol
p + facet_wrap(vars(class), nrow = 4)


# You can facet by multiple variables
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(vars(cyl, drv))

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

```
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(vars(cyl, drv), labeller = "label_both")

# To change the order in which the panels appear, change the levels
# of the underlying factor.
mpg$class2 <- reorder(mpg$class, mpg$displ)
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(vars(class2))

# By default, the same scales are used for all panels. You can allow
# scales to vary across the panels with the `scales` argument.
# Free scales make it easier to see patterns within each panel, but
# harder to compare across panels.
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(vars(class), scales = "free")

# When scales are constant, duplicated axes can be shown with
# or without labels
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(vars(class), axes = "all", axis.labels = "all_y")

# To repeat the same data in every panel, simply construct a data frame
# that does not contain the faceting variable.
ggplot(mpg, aes(displ, hwy)) +
  geom_point(data = transform(mpg, class = NULL), colour = "grey85") +
  geom_point() +
  facet_wrap(vars(class))

# Use `strip.position` to display the facet labels at the side of your
# choice. Setting it to `bottom` makes it act as a subtitle for the axis.
# This is typically used with free scales and a theme without boxes around
# strip labels.
ggplot(economics_long, aes(date, value)) +
  geom_line() +
  facet_wrap(vars(variable), scales = "free_y", nrow = 2, strip.position = "top") +
  theme(strip.background = element_blank(), strip.placement = "outside")


# The two letters determine the starting position, so 'tr' starts
# in the top-right.
# The first letter determines direction, so 'tr' fills top-to-bottom.
# `dir = "tr"` is equivalent to `dir = "v", as.table = FALSE`
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  facet_wrap(vars(class), dir = "tr")
```

`faithfuld` *2d density estimate of Old Faithful data*

Description

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

Usage

```
faithfuld
```

Format

A data frame with 5,625 observations and 3 variables:

eruptions Eruption time in mins

waiting Waiting time to next eruption in mins

density 2d density estimate

`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

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

Arguments

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

See Also

`fortify.lm()`

Other plotting automation topics: `autolayer()`, `automatic_plotting`, `autoplot()`

geom_abline *Reference lines: horizontal, vertical, and diagonal*

Description

These geoms add reference lines (sometimes called rules) to a plot, either horizontal, vertical, or diagonal (specified by slope and intercept). These are useful for annotating plots.

Usage

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

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

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`.

A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created.

A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`...` Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`na.rm` If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`position` A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`xintercept`, `yintercept`, `slope`, `intercept`
: Parameters that control the position of the line. If these are set, `data`, `mapping` and `show.legend` are overridden.

Details

These geoms act slightly differently from 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. `geom_abline(intercept = 0, slope = 1)`, 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 x and y aesthetics which are commonly set in the plot. They also do not affect the x and y scales.

Aesthetics

These geoms are drawn using `geom_line()` so they support the same aesthetics: `alpha`, `colour`, `linetype` and `linewidth`. They also each have aesthetics that control the position of the line:

- `geom_vline()`: `xintercept`
- `geom_hline()`: `yintercept`
- `geom_abline()`: `slope` and `intercept`

See Also

See `geom_segment()` for a more general approach to adding straight line segments to a plot.

Examples

```
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 *Bar charts*

Description

There are two types of bar charts: `geom_bar()` and `geom_col()`. `geom_bar()` 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 `geom_col()` instead. `geom_bar()` uses `stat_count()` by default: it counts the number of cases at each x position. `geom_col()` uses `stat_identity()`: it leaves the data as is.

Usage

```
geom_bar(
  mapping = NULL,
  data = NULL,
  stat = "count",
  position = "stack",
  ...,
  just = 0.5,
  lineend = "butt",
  linejoin = "mitre",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_col(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "stack",
  ...,
  just = 0.5,
  lineend = "butt",
  linejoin = "mitre",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_count(
  mapping = NULL,
  data = NULL,
  geom = "bar",
  position = "stack",
  ...,
  orientation = NA,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is

technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`just`	Adjustment for column placement. Set to `0.5` by default, meaning that columns will be centered about axis breaks. Set to `0` or `1` to place columns to the left/right of axis breaks. Note that this argument may have unintended behaviour when used with alternative positions, e.g. `position_dodge()`.
`lineend`	Line end style (round, butt, square).
`linejoin`	Line join style (round, mitre, bevel).
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom, stat`	Override the default connection between `geom_bar()` and `stat_count()`. For more information about overriding these connections, see how the stat and geom arguments work.
`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.

Details

A bar chart uses height to represent a value, and so the base of the bar must always be shown to produce a valid visual comparison. Proceed with caution when using transformed scales with a bar chart. It's important to always use a meaningful reference point for the base of the bar. For example, for log transformations the reference point is 1. In fact, when using a log scale, `geom_bar()` automatically places the base of the bar at 1. Furthermore, never use stacked bars with a transformed scale, because scaling happens before stacking. As a consequence, the height of bars will be wrong when stacking occurs with a transformed scale.

By default, multiple bars occupying the same `x` position will be stacked atop one another by `position_stack()`. If you want them to be dodged side-to-side, use `position_dodge()` or `position_dodge2()`. Finally, `position_fill()` shows relative proportions at each `x` by stacking the bars and then standardising each bar to have the same height.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(count)`
  number of points in bin.
- `after_stat(prop)`
  groupwise proportion

Aesthetics

`geom_bar()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred
• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`
• `width`	→ `0.9`

`geom_col()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred

• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`
• `width`	→ `0.9`

`stat_count()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• **`x` *or* `y`**	
• `group`	→ inferred
• `weight`	→ `1`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

`geom_histogram()` for continuous data, `position_dodge()` and `position_dodge2()` for creating side-by-side bar charts.

`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

```
# 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))
# Map class to y instead to flip the orientation
ggplot(mpg) + geom_bar(aes(y = class))

# Bar charts are automatically stacked when multiple bars are placed
# at the same location. The order of the fill is designed to match
# the legend
g + geom_bar(aes(fill = drv))

# If you need to flip the order (because you've flipped the orientation)
# call position_stack() explicitly:
ggplot(mpg, aes(y = class)) +
 geom_bar(aes(fill = drv), position = position_stack(reverse = TRUE)) +
 theme(legend.position = "top")

# To show (e.g.) means, you need geom_col()
df <- data.frame(trt = c("a", "b", "c"), outcome = c(2.3, 1.9, 3.2))
ggplot(df, aes(trt, outcome)) +
  geom_col()
# But geom_point() displays 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)

# Use `just` to control how columns are aligned with axis breaks:
df <- data.frame(x = as.Date(c("2020-01-01", "2020-02-01")), y = 1:2)
# Columns centered on the first day of the month
ggplot(df, aes(x, y)) + geom_col(just = 0.5)
# Columns begin on the first day of the month
ggplot(df, aes(x, y)) + geom_col(just = 1)
```

---

geom_bin_2d *Heatmap of 2d bin counts*

Description

Divides the plane into rectangles, counts the number of cases in each rectangle, and then (by default) maps the number of cases to the rectangle's fill. This is a useful alternative to `geom_point()` in the presence of overplotting.

Usage

```
geom_bin_2d(
  mapping = NULL,
  data = NULL,
  stat = "bin2d",
  position = "identity",
  ...,
  lineend = "butt",
  linejoin = "mitre",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is

technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`lineend`	Line end style (round, butt, square).
`linejoin`	Line join style (round, mitre, bevel).
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom, stat`	Use to override the default connection between `geom_bin_2d()` and `stat_bin_2d()`. For more information about overriding these connections, see how the stat and geom arguments work.
`binwidth`	The width of the bins. Can be specified as a numeric value or as a function that takes x after scale transformation as input and returns a single numeric value. When specifying a function along with a grouping structure, the function will be called once per group. The default is to use the number of bins in `bins`, covering the range of the data. You should always override this value, exploring multiple widths to find the best to illustrate the stories in your data. The bin width of a date variable is the number of days in each time; the bin width of a time variable is the number of seconds.
`bins`	Number of bins. Overridden by `binwidth`. Defaults to 30.
`breaks`	Alternatively, you can supply a numeric vector giving the bin boundaries. Overrides `binwidth`, `bins`, `center`, and `boundary`. Can also be a function that takes group-wise values as input and returns bin boundaries.
`drop`	if `TRUE` removes all cells with 0 counts.
`closed`	One of `"right"` or `"left"` indicating whether right or left edges of bins are included in the bin.

`center`, `boundary`
: bin position specifiers. Only one, `center` or `boundary`, may be specified for a single plot. `center` specifies the center of one of the bins. `boundary` specifies the boundary between two bins. Note that if either is above or below the range of the data, things will be shifted by the appropriate integer multiple of `binwidth`. For example, to center on integers use `binwidth = 1` and `center = 0`, even if `0` is outside the range of the data. Alternatively, this same alignment can be specified with `binwidth = 1` and `boundary = 0.5`, even if `0.5` is outside the range of the data.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(count)`
  number of points in bin.
- `after_stat(density)`
  density of points in bin, scaled to integrate to 1.
- `after_stat(ncount)`
  count, scaled to maximum of 1.
- `after_stat(ndensity)`
  density, scaled to a maximum of 1.

Controlling binning parameters for the x and y directions

The arguments `bins`, `binwidth`, `breaks`, `center`, and `boundary` can be set separately for the x and y directions. When given as a scalar, one value applies to both directions. When given as a vector of length two, the first is applied to the x direction and the second to the y direction. Alternatively, these can be a named list containing `x` and `y` elements, for example `list(x = 10, y = 20)`.

Aesthetics

`geom_bin2d()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred
• `height`	→ `1`
• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`
• `width`	→ `1`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

stat_bin_hex() for hexagonal binning

Examples

```
d <- ggplot(diamonds, aes(x, y)) + xlim(4, 10) + ylim(4, 10)
d + geom_bin_2d()

# You can control the size of the bins by specifying the number of
# bins in each direction:
d + geom_bin_2d(bins = 10)
d + geom_bin_2d(bins = list(x = 30, y = 10))

# Or by specifying the width of the bins
d + geom_bin_2d(binwidth = c(0.1, 0.1))
```

---

geom_blank *Draw nothing*

Description

The blank geom draws nothing, but can be a useful way of ensuring common scales between different plots. See expand_limits() for more details.

Usage

```
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(). If specified and inherit.aes = TRUE (the default), it is combined with the default mapping at the top level of the plot. You must supply mapping if there is no plot mapping.
data	The data to be displayed in this layer. There are three options: If NULL, the default, the data is inherited from the plot data as specified in the call to ggplot(). A data.frame, or other object, will override the plot data. All objects will be fortified to produce a data frame. See fortify() for which variables will be created.

A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`stat`
: The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following:

- A `Stat` ggproto subclass, for example `StatCount`.
- A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.
- For more information and other ways to specify the stat, see the layer stat documentation.

`position`
: A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...`
: Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use TRUE. If NA, all levels are shown in legend, but unobserved levels are omitted.
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. annotation_borders().

Examples

```
ggplot(mtcars, aes(wt, mpg))
# Nothing to see here!
```

---

geom_boxplot *A box and whiskers plot (in the style of Tukey)*

Description

The boxplot compactly displays the distribution of a continuous variable. It visualises five summary statistics (the median, two hinges and two whiskers), and all "outlying" points individually.

Usage

```
geom_boxplot(
  mapping = NULL,
  data = NULL,
  stat = "boxplot",
  position = "dodge2",
  ...,
  outliers = TRUE,
  outlier.colour = NULL,
  outlier.color = NULL,
  outlier.fill = NULL,
  outlier.shape = NULL,
  outlier.size = NULL,
  outlier.stroke = 0.5,
  outlier.alpha = NULL,
  whisker.colour = NULL,
  whisker.color = NULL,
  whisker.linetype = NULL,
  whisker.linewidth = NULL,
  staple.colour = NULL,
  staple.color = NULL,
  staple.linetype = NULL,
  staple.linewidth = NULL,
  median.colour = NULL,
  median.color = NULL,
  median.linetype = NULL,
  median.linewidth = NULL,
  box.colour = NULL,
  box.color = NULL,
  box.linetype = NULL,
  box.linewidth = NULL,
  notch = FALSE,
  notchwidth = 0.5,
  staplewidth = 0,
  varwidth = FALSE,
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_boxplot(
  mapping = NULL,
  data = NULL,
  geom = "boxplot",
  position = "dodge2",
  ...,
  orientation = NA,
  coef = 1.5,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...` Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`outliers` Whether to display (`TRUE`) or discard (`FALSE`) outliers from the plot. Hiding or discarding outliers can be useful when, for example, raw data points need to be displayed on top of the boxplot. By discarding outliers, the axis limits will adapt to the box and whiskers only, not the full data range. If outliers need to be hidden and the axes needs to show the full data range, please use `outlier.shape = NA` instead.

`outlier.colour`, `outlier.color`, `outlier.fill`, `outlier.shape`, `outlier.size`, `outlier.stroke`, `outlier.alpha`

Default aesthetics for outliers. Set to `NULL` to inherit from the data's aesthetics.

`whisker.colour`, `whisker.color`, `whisker.linetype`, `whisker.linewidth`

Default aesthetics for the whiskers. Set to `NULL` to inherit from the data's aesthetics.

`staple.colour`, `staple.color`, `staple.linetype`, `staple.linewidth`

Default aesthetics for the staples. Set to `NULL` to inherit from the data's aesthetics. Note that staples don't appear unless the `staplewidth` argument is set to a non-zero size.

`median.colour`, `median.color`, `median.linetype`, `median.linewidth`

Default aesthetics for the median line. Set to `NULL` to inherit from the data's aesthetics.

`box.colour`, `box.color`, `box.linetype`, `box.linewidth`	Default aesthetics for the boxes. Set to `NULL` to inherit from the data's aesthetics.
`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 (defaults to `notchwidth = 0.5`).
`staplewidth`	The relative width of staples to the width of the box. Staples mark the ends of the whiskers with a line.
`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom`, `stat`	Use to override the default connection between `geom_boxplot()` and `stat_boxplot()`. For more information about overriding these connections, see how the stat and geom arguments work.
`coef`	Length of the whiskers as multiple of IQR. Defaults to 1.5.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Summary statistics

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()`.

The upper whisker extends from the hinge to the largest value no further than 1.5 * IQR from 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 smallest value at most 1.5 * IQR of the hinge. Data beyond the end of the whiskers are called "outlying" points and are plotted individually.

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

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation. `stat_boxplot()` provides the following variables, some of which depend on the orientation:

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

Aesthetics

`geom_boxplot()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`x` ***or*** `y`	
•	`lower` ***or*** `xlower`	
•	`upper` ***or*** `xupper`	
•	`middle` ***or*** `xmiddle`	
•	`ymin` ***or*** `xmin`	
•	`ymax` ***or*** `xmax`	
•	`alpha`	→ NA
•	`colour`	→ via `theme()`
•	`fill`	→ via `theme()`
•	`group`	→ inferred

• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`
• `shape`	→ via `theme()`
• `size`	→ via `theme()`
• `weight`	→ `1`
• `width`	→ `0.9`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Note

In the unlikely event you specify both US and UK spellings of colour, the US spelling will take precedence.

References

McGill, R., Tukey, J. W. and Larsen, W. A. (1978) Variations of box plots. The American Statistician 32, 12-16.

See Also

`geom_quantile()` for continuous `x`, `geom_violin()` for a richer display of the distribution, and `geom_jitter()` for a useful technique for small data.

Examples

```
p <- ggplot(mpg, aes(class, hwy))
p + geom_boxplot()
# Orientation follows the discrete axis
ggplot(mpg, aes(hwy, class)) + geom_boxplot()

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)
# Remove outliers when overlaying boxplot with original data points
p + geom_boxplot(outlier.shape = NA) + geom_jitter(width = 0.2)

# 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)))
# Adjust the transparency of outliers using outlier.alpha
```

```
ggplot(diamonds, aes(carat, price)) +
  geom_boxplot(aes(group = cut_width(carat, 0.25)), outlier.alpha = 0.1)


# It's possible to draw a boxplot with your own computations if you
# use stat = "identity":
set.seed(1)
y <- rnorm(100)
df <- data.frame(
  x = 1,
  y0 = min(y),
  y25 = quantile(y, 0.25),
  y50 = median(y),
  y75 = quantile(y, 0.75),
  y100 = max(y)
)
ggplot(df, aes(x)) +
  geom_boxplot(
   aes(ymin = y0, lower = y25, middle = y50, upper = y75, ymax = y100),
   stat = "identity"
 )
```

---

geom_contour *2D contours of a 3D surface*

Description

ggplot2 can not draw true 3D surfaces, but you can use `geom_contour()`, `geom_contour_filled()`, and `geom_tile()` to visualise 3D surfaces in 2D.

These functions require regular data, where the x and y coordinates form an equally spaced grid, and each combination of x and y appears once. Missing values of z are allowed, but contouring will only work for grid points where all four corners are non-missing. If you have irregular data, you'll need to first interpolate on to a grid before visualising, using `interp::interp()`, `akima::bilinear()`, or similar.

Usage

```
geom_contour(
  mapping = NULL,
  data = NULL,
  stat = "contour",
  position = "identity",
  ...,
  bins = NULL,
  binwidth = NULL,
  breaks = NULL,
  arrow = NULL,
  arrow.fill = NULL,
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_contour_filled(
  mapping = NULL,
  data = NULL,
  stat = "contour_filled",
  position = "identity",
  ...,
  bins = NULL,
  binwidth = NULL,
  breaks = NULL,
  rule = "evenodd",
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

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

stat_contour_filled(
  mapping = NULL,
  data = NULL,
  geom = "contour_filled",
  position = "identity",
  ...,
  bins = NULL,
  binwidth = NULL,
  breaks = NULL,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`bins`	Number of contour bins. Overridden by `breaks`.
`binwidth`	The width of the contour bins. Overridden by `bins`.
`breaks`	One of: • Numeric vector to set the contour breaks • A function that takes the range of the data and binwidth as input and returns breaks as output. A function can be created from a formula (e.g. ~ fullseq(.x, .y)). Overrides `binwidth` and `bins`. By default, this is a vector of length ten with `pretty()` breaks.
`arrow`	Arrow specification, as created by `grid::arrow()`.
`arrow.fill`	fill colour to use for the arrow head (if closed). `NULL` means use `colour` aesthetic.
`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

`rule`	Either `"evenodd"` or `"winding"`. If polygons with holes are being drawn (using the `subgroup` aesthetic) this argument defines how the hole coordinates are interpreted. See the examples in `grid::pathGrob()` for an explanation.
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`. • A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`. • For more information and other ways to specify the geom, see the layer geom documentation.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation. The computed variables differ somewhat for contour lines (computed by `stat_contour()`) and contour bands (filled contours, computed by `stat_contour_filled()`). The variables `nlevel` and `piece` are available for both, whereas `level_low`, `level_high`, and `level_mid` are only available for bands. The variable `level` is a numeric or a factor depending on whether lines or bands are calculated.

- `after_stat(level)`
  Height of contour. For contour lines, this is a numeric vector that represents bin boundaries. For contour bands, this is an ordered factor that represents bin ranges.
- `after_stat(level_low)`, `after_stat(level_high)`, `after_stat(level_mid)`
  (contour bands only) Lower and upper bin boundaries for each band, as well as the mid point between boundaries.
- `after_stat(nlevel)`
  Height of contour, scaled to a maximum of 1.
- `after_stat(piece)`
  Contour piece (an integer).

Dropped variables

`z` After contouring, the z values of individual data points are no longer available.

Aesthetics

`geom_contour()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`x`	
•	`y`	
•	`alpha`	→ NA
•	`colour`	→ via `theme()`

- `group` → inferred
- `linetype` → via `theme()`
- `linewidth` → via `theme()`
- `weight` → `1`

`geom_contour_filled()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`
- `y`
- `alpha` → `NA`
- `colour` → via `theme()`
- `fill` → via `theme()`
- `group` → inferred
- `linetype` → via `theme()`
- `linewidth` → via `theme()`
- `subgroup` → `NULL`

`stat_contour()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`
- `y`
- `z`
- `group` → inferred
- `order` → `after_stat(level)`

`stat_contour_filled()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`
- `y`
- `z`
- `fill` → `after_stat(level)`
- `group` → inferred
- `order` → `after_stat(level)`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

`geom_density_2d()`: 2d density contours

Examples

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

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


# use geom_contour_filled() for filled contours
v + geom_contour_filled()

# Setting bins creates evenly spaced contours in the range of the data
v + geom_contour(bins = 3)
v + geom_contour(bins = 5)

# Setting binwidth does the same thing, parameterised by the distance
# between contours
v + geom_contour(binwidth = 0.01)
v + geom_contour(binwidth = 0.001)

# Other parameters
v + geom_contour(aes(colour = after_stat(level)))
v + geom_contour(colour = "red")
v + geom_raster(aes(fill = density)) +
  geom_contour(colour = "white")
```

`geom_count` *Count overlapping points*

Description

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

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()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom, stat`	Use to override the default connection between `geom_count()` and `stat_sum()`. For more information about overriding these connections, see how the stat and geom arguments work.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(n)`
  Number of observations at position.
- `after_stat(prop)`
  Percent of points in that panel at that position.

Aesthetics

`geom_point()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `alpha`	→ NA
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred
• `shape`	→ via `theme()`
• `size`	→ via `theme()`

- `stroke` → via `theme()`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

For continuous `x` and `y`, use `geom_bin_2d()`.

Examples

```
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 = after_stat(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 = after_stat(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 = after_stat(prop), group = cut)) +
  scale_size_area(max_size = 10)
d + geom_count(aes(size = after_stat(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`. Each case draws a single graphical object.

Usage

```
geom_crossbar(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  middle.colour = NULL,
  middle.color = NULL,
  middle.linetype = NULL,
  middle.linewidth = NULL,
  box.colour = NULL,
  box.color = NULL,
  box.linetype = NULL,
  box.linewidth = NULL,
  fatten = deprecated(),
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE
)

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

geom_errorbarh(..., orientation = "y")

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

geom_pointrange(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  orientation = NA,
  fatten = deprecated(),
  lineend = "butt",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

mapping	Set of aesthetic mappings created by aes(). If specified and inherit.aes = TRUE (the default), it is combined with the default mapping at the top level of the plot. You must supply mapping if there is no plot mapping.
data	The data to be displayed in this layer. There are three options: If NULL, the default, the data is inherited from the plot data as specified in the call to ggplot(). A data.frame, or other object, will override the plot data. All objects will be fortified to produce a data frame. See fortify() for which variables will be created. A function will be called with a single argument, the plot data. The return value must be a data.frame, and will be used as the layer data. A function can be created from a formula (e.g. ~ head(.x, 10)).
stat	The statistical transformation to use on the data for this layer. When using a geom_*() function to construct a layer, the stat argument can be used to override the default coupling between geoms and stats. The stat argument accepts the following: • A Stat ggproto subclass, for example StatCount. • A string naming the stat. To give the stat as a string, strip the function name of the stat_ prefix. For example, to use stat_count(), give the stat as "count". • For more information and other ways to specify the stat, see the layer stat documentation.
position	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The position argument accepts the following: • The result of calling a position function, such as position_jitter(). This method allows for passing extra arguments to the position.

- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...` Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`middle.colour`, `middle.color`, `middle.linetype`, `middle.linewidth`
Default aesthetics for the middle line. Set to `NULL` to inherit from the data's aesthetics.

`box.colour`, `box.color`, `box.linetype`, `box.linewidth`
Default aesthetics for the boxes. Set to `NULL` to inherit from the data's aesthetics.

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

`na.rm` If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`orientation` The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`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. `annotation_borders()`.
`lineend`	Line end style (round, butt, square).

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Aesthetics

`geom_linerange()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- **`x` *or* `y`**
- **`ymin` *or* `xmin`**
- **`ymax` *or* `xmax`**
- `alpha` → NA
- `colour` → via `theme()`
- `group` → inferred
- `linetype` → via `theme()`
- `linewidth` → via `theme()`

Note that `geom_pointrange()` also understands `size` for the size of the points.

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Note

`geom_errorbarh()` is **[Deprecated]**. Use `geom_errorbar(orientation = "y")` instead.

See Also

`stat_summary()` for examples of these guys in use, `geom_smooth()` for continuous analogue

Examples

```
# Create a simple example dataset
df <- 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)

# Flip the orientation by changing mapping
ggplot(df, aes(resp, trt, colour = group)) +
  geom_linerange(aes(xmin = lower, xmax = upper))

# Draw lines connecting group means
p +
  geom_line(aes(group = group)) +
  geom_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 +
 geom_col(position = "dodge") +
 geom_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 +
  geom_col(position = dodge) +
  geom_errorbar(aes(ymin = lower, ymax = upper), position = dodge, width = 0.25)

# When using geom_errorbar() with position_dodge2(), extra padding will be
# needed between the error bars to keep them aligned with the bars.
p +
geom_col(position = "dodge2") +
geom_errorbar(
  aes(ymin = lower, ymax = upper),
  position = position_dodge2(width = 0.5, padding = 0.5)
)
```

geom_density *Smoothed density estimates*

Description

Computes and draws kernel density estimate, which is a smoothed version of the histogram. This is a useful alternative to the histogram for continuous data that comes from an underlying smooth distribution.

Usage

```
geom_density(
  mapping = NULL,
  data = NULL,
  stat = "density",
  position = "identity",
  ...,
  outline.type = "upper",
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_density(
  mapping = NULL,
  data = NULL,
  geom = "area",
  position = "stack",
  ...,
  orientation = NA,
  bw = "nrd0",
  adjust = 1,
  kernel = "gaussian",
  n = 512,
  trim = FALSE,
  bounds = c(-Inf, Inf),
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`outline.type`	Type of the outline of the area; `"both"` draws both the upper and lower lines, `"upper"`/`"lower"` draws the respective lines only. `"full"` draws a closed polygon around the area.
`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`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. `annotation_borders()`.
`geom, stat`	Use to override the default connection between `geom_density()` and `stat_density()`. For more information about overriding these connections, see how the stat and geom arguments work.
`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.
`bw`	The smoothing bandwidth to be used. If numeric, the standard deviation of the smoothing kernel. If character, a rule to choose the bandwidth, as listed in `stats::bw.nrd()`. Note that automatic calculation of the bandwidth does not take weights into account.
`adjust`	A multiplicate bandwidth adjustment. This makes it possible to adjust the bandwidth while still using the a bandwidth estimator. For example, `adjust = 1/2` means use half of the default bandwidth.
`kernel`	Kernel. See list of available kernels in `density()`.
`n`	number of equally spaced points at which the density is to be estimated, should be a power of two, see `density()` for details
`trim`	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. This parameter only matters if you are displaying multiple densities in one plot or if you are manually adjusting the scale limits.
`bounds`	Known lower and upper bounds for estimated data. Default `c(-Inf, Inf)` means that there are no (finite) bounds. If any bound is finite, boundary effect of default density estimation will be corrected by reflecting tails outside `bounds` around their closest edge. Data points outside of bounds are removed with a warning.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(density)`
  density estimate.

- `after_stat(count)`
  density * number of points - useful for stacked density plots.
- `after_stat(wdensity)`
  density * sum of weights. In absence of weights, the same as `count`.
- `after_stat(scaled)`
  density estimate, scaled to maximum of 1.
- `after_stat(n)`
  number of points.
- `after_stat(ndensity)`
  alias for `scaled`, to mirror the syntax of `stat_bin()`.

Aesthetics

`geom_density()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`x`	
•	`y`	
•	`alpha`	→ `NA`
•	`colour`	→ via `theme()`
•	`fill`	→ via `theme()`
•	`group`	→ inferred
•	`linetype`	→ via `theme()`
•	`linewidth`	→ via `theme()`
•	`weight`	→ `1`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

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

Examples

```
ggplot(diamonds, aes(carat)) +
  geom_density()
# Map the values to y to flip the orientation
ggplot(diamonds, aes(y = 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)

# Use `bounds` to adjust computation for known data limits
big_diamonds <- diamonds[diamonds$carat >= 1, ]
ggplot(big_diamonds, aes(carat)) +
  geom_density(color = 'red') +
  geom_density(bounds = c(1, Inf), color = 'blue')


# 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

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

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

geom_density_2d *Contours of a 2D density estimate*

Description

Perform a 2D kernel density estimation using MASS::kde2d() and display the results with contours. This can be useful for dealing with overplotting. This is a 2D version of geom_density(). geom_density_2d() draws contour lines, and geom_density_2d_filled() draws filled contour bands.

Usage

```
geom_density_2d(
  mapping = NULL,
  data = NULL,
  stat = "density_2d",
  position = "identity",
  ...,
  contour_var = "density",
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_density_2d_filled(
  mapping = NULL,
  data = NULL,
  stat = "density_2d_filled",
  position = "identity",
  ...,
  contour_var = "density",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_density_2d(
  mapping = NULL,
  data = NULL,
  geom = "density_2d",
  position = "identity",
  ...,
  contour = TRUE,
  contour_var = "density",
  h = NULL,
  adjust = c(1, 1),
  n = 100,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_density_2d_filled(
  mapping = NULL,
  data = NULL,
  geom = "density_2d_filled",
  position = "identity",
  ...,
  contour = TRUE,
  contour_var = "density",
  h = NULL,
  adjust = c(1, 1),
  n = 100,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
```

```
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Arguments passed on to `geom_contour` `binwidth` The width of the contour bins. Overridden by `bins`. `bins` Number of contour bins. Overridden by `breaks`. `breaks` One of: • Numeric vector to set the contour breaks • A function that takes the range of the data and binwidth as input and returns breaks as output. A function can be created from a formula (e.g. ~ fullseq(.x, .y)). Overrides `binwidth` and `bins`. By default, this is a vector of length ten with `pretty()` breaks.
`contour_var`	Character string identifying the variable to contour by. Can be one of `"density"`, `"ndensity"`, or `"count"`. See the section on computed variables for details.
`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom`, `stat`	Use to override the default connection between `geom_density_2d()` and `stat_density_2d()`. For more information at overriding these connections, see how the stat and geom arguments work.
`contour`	If `TRUE`, contour the results of the 2d density estimation.
`h`	Bandwidth (vector of length two). If `NULL`, estimated using `MASS::bandwidth.nrd()`.
`adjust`	A multiplicative bandwidth adjustment to be used if 'h' is 'NULL'. This makes it possible to adjust the bandwidth while still using the a bandwidth estimator. For example, `adjust = 1/2` means use half of the default bandwidth.
`n`	Number of grid points in each direction.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation. `stat_density_2d()` and `stat_density_2d_filled()` compute different variables depending on whether contouring is turned on or off. With contouring off (`contour = FALSE`), both stats behave the same, and the following variables are provided:

- `after_stat(density)`
  The density estimate.
- `after_stat(ndensity)`
  Density estimate, scaled to a maximum of 1.
- `after_stat(count)`
  Density estimate * number of observations in group.
- `after_stat(n)`
  Number of observations in each group.

With contouring on (`contour = TRUE`), either `stat_contour()` or `stat_contour_filled()` (for contour lines or contour bands, respectively) is run after the density estimate has been obtained, and the computed variables are determined by these stats. Contours are calculated for one of the three types of density estimates obtained before contouring, `density`, `ndensity`, and `count`. Which of those should be used is determined by the `contour_var` parameter.

Dropped variables

`z` After density estimation, the z values of individual data points are no longer available.

If contouring is enabled, then similarly `density`, `ndensity`, and `count` are no longer available after the contouring pass.

Aesthetics

`geom_density2d()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`x`	
•	`y`	
•	`alpha`	→ `NA`
•	`colour`	→ via `theme()`
•	`group`	→ inferred
•	`linetype`	→ via `theme()`
•	`linewidth`	→ via `theme()`

`geom_density2d_filled()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`x`	
•	`y`	
•	`alpha`	→ `NA`
•	`colour`	→ via `theme()`
•	`fill`	→ via `theme()`
•	`group`	→ inferred
•	`linetype`	→ via `theme()`
•	`linewidth`	→ via `theme()`
•	`subgroup`	→ `NULL`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

`geom_contour()`, `geom_contour_filled()` for information about how contours are drawn; `geom_bin_2d()` for another way of dealing with overplotting.

Examples

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

# contour lines
m + geom_density_2d()


# contour bands
m + geom_density_2d_filled(alpha = 0.5)

# contour bands and contour lines
```

```
m + geom_density_2d_filled(alpha = 0.5) +
  geom_density_2d(linewidth = 0.25, colour = "black")

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 you draw filled contours across multiple facets, the same bins are
# used across all facets
d + geom_density_2d_filled() + facet_wrap(vars(cut))
# If you want to make sure the peak intensity is the same in each facet,
# use `contour_var = "ndensity"`.
d + geom_density_2d_filled(contour_var = "ndensity") + facet_wrap(vars(cut))
# If you want to scale intensity by the number of observations in each group,
# use `contour_var = "count"`.
d + geom_density_2d_filled(contour_var = "count") + facet_wrap(vars(cut))

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

---

geom_dotplot *Dot plot*

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()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is

technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`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", "centerwhole" (centered, but with dots aligned)
`stackratio`	how close to stack the dots. Default is 1, where dots 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)
`width`	When `binaxis` is "y", the spacing of the dot stacks for dodging.
`drop`	If TRUE, remove all bins with zero counts
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Details

There are two basic approaches: *dot-density* and *histodot*. 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.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(x)`
  center of each bin, if `binaxis` is `"x"`.
- `after_stat(y)`
  center of each bin, if `binaxis` is `"x"`.
- `after_stat(binwidth)`
  maximum width of each bin if method is `"dotdensity"`; width of each bin if method is `"histodot"`.
- `after_stat(count)`
  number of points in bin.
- `after_stat(ncount)`
  count, scaled to a maximum of 1.
- `after_stat(density)`
  density of points in bin, scaled to integrate to 1, if method is `"histodot"`.
- `after_stat(ndensity)`
  density, scaled to maximum of 1, if method is `"histodot"`.

Aesthetics

`geom_dotplot()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred
• `linetype`	→ via `theme()`
• `stroke`	→ via `theme()`
• `weight`	→ `1`
• `width`	→ `0.9`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

References

Wilkinson, L. (1999) Dot plots. The American Statistician, 53(3), 276-281.

Examples

```
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")

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

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

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

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

# Change dot fill colour, stroke width
ggplot(mtcars, aes(x = mpg)) +
  geom_dotplot(binwidth = 1.5, fill = "white", stroke = 2)


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

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

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

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
ggplot(mtcars, aes(x = factor(am), y = mpg)) +
  geom_dotplot(binaxis = "y", stackdir = "center", binpositions="all")

# Stacking multiple groups, with different fill
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")

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

---

geom_freqpoly *Histograms and frequency polygons*

Description

Visualise the distribution of a single continuous variable by dividing the x axis into bins and counting the number of observations in each bin. Histograms (`geom_histogram()`) display the counts with bars; frequency polygons (`geom_freqpoly()`) display the counts with lines. Frequency polygons are more suitable when you want to compare the distribution across the levels of a categorical variable.

Usage

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

geom_histogram(
  mapping = NULL,
  data = NULL,
  stat = "bin",
  position = "stack",
  ...,
  binwidth = NULL,
  bins = NULL,
  orientation = NA,
  lineend = "butt",
  linejoin = "mitre",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_bin(
  mapping = NULL,
  data = NULL,
  geom = "bar",
  position = "stack",
  ...,
  orientation = NA,
  binwidth = NULL,
  bins = NULL,
  center = NULL,
  boundary = NULL,
  closed = c("right", "left"),
  pad = FALSE,
  breaks = NULL,
  drop = "none",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.

	• A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`binwidth`	The width of the bins. Can be specified as a numeric value or as a function that takes x after scale transformation as input and returns a single numeric value. When specifying a function along with a grouping structure, the function will be called once per group. The default is to use the number of bins in `bins`, covering the range of the data. You should always override this value, exploring multiple widths to find the best to illustrate the stories in your data. The bin width of a date variable is the number of days in each time; the bin width of a time variable is the number of seconds.
`bins`	Number of bins. Overridden by `binwidth`. Defaults to 30.

`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.
`lineend`	Line end style (round, butt, square).
`linejoin`	Line join style (round, mitre, bevel).
`geom, stat`	Use to override the default connection between `geom_histogram()`/`geom_freqpoly()` and `stat_bin()`. For more information at overriding these connections, see how the stat and geom arguments work.
`center, boundary`	bin position specifiers. Only one, `center` or `boundary`, may be specified for a single plot. `center` specifies the center of one of the bins. `boundary` specifies the boundary between two bins. Note that if either is above or below the range of the data, things will be shifted by the appropriate integer multiple of `binwidth`. For example, to center on integers use `binwidth = 1` and `center = 0`, even if `0` is outside the range of the data. Alternatively, this same alignment can be specified with `binwidth = 1` and `boundary = 0.5`, even if `0.5` is outside the range of the data.
`closed`	One of `"right"` or `"left"` indicating whether right or left edges of bins are included in the bin.
`pad`	If `TRUE`, adds empty bins at either end of x. This ensures frequency polygons touch 0. Defaults to `FALSE`.
`breaks`	Alternatively, you can supply a numeric vector giving the bin boundaries. Overrides `binwidth`, `bins`, `center`, and `boundary`. Can also be a function that takes group-wise values as input and returns bin boundaries.
`drop`	Treatment of zero count bins. If `"none"` (default), such bins are kept as-is. If `"all"`, all zero count bins are filtered out. If `"extremes"` only zero count bins at the flanks are filtered out, but not in the middle. `TRUE` is shorthand for `"all"` and `FALSE` is shorthand for `"none"`.

Details

`stat_bin()` is suitable only for continuous x data. If your x data is discrete, you probably want to use `stat_count()`.

By default, the underlying computation (`stat_bin()`) uses 30 bins; this is not a good default, but the idea is to get you experimenting with different number of bins. You can also experiment modifying the `binwidth` with `center` or `boundary` arguments. `binwidth` overrides `bins` so you should do one change at a time. You may need to look at a few options to uncover the full story behind your data.

By default, the *height* of the bars represent the counts within each bin. However, there are situations where this behavior might produce misleading plots (e.g., when non-equal-width bins are used), in which case it might be preferable to have the *area* of the bars represent the counts (by setting `aes(y = after_stat(count / width))`). See example below.

In addition to `geom_histogram()`, you can create a histogram plot by using `scale_x_binned()` with `geom_bar()`. This method by default plots tick marks in between each bar.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Aesthetics

`geom_histogram()` uses the same aesthetics as `geom_bar()`; `geom_freqpoly()` uses the same aesthetics as `geom_line()`.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(count)`
  number of points in bin.
- `after_stat(density)`
  density of points in bin, scaled to integrate to 1.
- `after_stat(ncount)`
  count, scaled to a maximum of 1.
- `after_stat(ndensity)`
  density, scaled to a maximum of 1.
- `after_stat(width)`
  widths of bins.

Dropped variables

`weight` After binning, weights of individual data points (if supplied) are no longer available.

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

```
ggplot(diamonds, aes(carat)) +
  geom_histogram()
ggplot(diamonds, aes(carat)) +
  geom_histogram(binwidth = 0.01)
ggplot(diamonds, aes(carat)) +
  geom_histogram(bins = 200)
# Map values to y to flip the orientation
ggplot(diamonds, aes(y = carat)) +
  geom_histogram()
```

```
# For histograms with tick marks between each bin, use `geom_bar()` with
# `scale_x_binned()`.
ggplot(diamonds, aes(carat)) +
  geom_bar() +
  scale_x_binned()

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

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


# When using the non-equal-width bins, we should set the area of the bars to
# represent the counts (not the height).
# Here we're using 10 equi-probable bins:
price_bins <- quantile(diamonds$price, probs = seq(0, 1, length = 11))

ggplot(diamonds, aes(price)) +
  geom_histogram(breaks = price_bins, color = "black") # misleading (height = count)

ggplot(diamonds, aes(price, after_stat(count / width))) +
  geom_histogram(breaks = price_bins, color = "black") # area = count

if (require("ggplot2movies")) {
# 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(boundary = 0) +
  coord_transform(x = "log10")
# Use boundary = 0, to make sure we don't take sqrt of negative values
m +
  geom_histogram(boundary = 0) +
  coord_transform(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()
}

# You can specify a function for calculating binwidth, which is
# particularly useful when faceting along variables with
# different ranges because the function will be called once per facet
ggplot(economics_long, aes(value)) +
  facet_wrap(~variable, scales = 'free_x') +
  geom_histogram(binwidth = \(x) 2 * IQR(x) / (length(x)^(1/3)))
```

`geom_function` *Draw a function as a continuous curve*

Description

Computes and draws a function as a continuous curve. This makes it easy to superimpose a function on top of an existing plot. The function is called with a grid of evenly spaced values along the x axis, and the results are drawn (by default) with a line.

Usage

```
geom_function(
  mapping = NULL,
  data = NULL,
  stat = "function",
  position = "identity",
  ...,
  arrow = NULL,
  arrow.fill = NULL,
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_function(
  mapping = NULL,
  data = NULL,
  geom = "function",
  position = "identity",
  ...,
  fun,
  xlim = NULL,
  n = 101,
  args = list(),
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	Ignored by `stat_function()`, do not use.
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.

- For more information and other ways to specify the position, see the layer position documentation.

`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`arrow`	Arrow specification, as created by `grid::arrow()`.
`arrow.fill`	fill colour to use for the arrow head (if closed). `NULL` means use `colour` aesthetic.
`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`.

- A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`.
- For more information and other ways to specify the geom, see the layer geom documentation.

`fun`	Function to use. Either 1) an anonymous function in the base or rlang formula syntax (see `rlang::as_function()`) or 2) a quoted or character name referencing a function; see examples. Must be vectorised.
`xlim`	Optionally, specify the range of the function.
`n`	Number of points to interpolate along the x axis.
`args`	List of additional arguments passed on to the function defined by `fun`.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(x)`
  `x` values along a grid.
- `after_stat(y)`
  values of the function evaluated at corresponding `x`.

Aesthetics

`geom_function()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `group`	→ inferred
• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

`rlang::as_function()`

Examples

```
# geom_function() is useful for overlaying functions
set.seed(1492)
ggplot(data.frame(x = rnorm(100)), aes(x)) +
  geom_density() +
  geom_function(fun = dnorm, colour = "red")
```

```
# To plot functions without data, specify range of x-axis
base <-
  ggplot() +
  xlim(-5, 5)

base + geom_function(fun = dnorm)

base + geom_function(fun = dnorm, args = list(mean = 2, sd = .5))

# The underlying mechanics evaluate the function at discrete points
# and connect the points with lines
base + stat_function(fun = dnorm, geom = "point")

base + stat_function(fun = dnorm, geom = "point", n = 20)

base + stat_function(fun = dnorm, geom = "polygon", color = "blue", fill = "blue", alpha = 0.5)

base + geom_function(fun = dnorm, n = 20)

# Two functions on the same plot
base +
  geom_function(aes(colour = "normal"), fun = dnorm) +
  geom_function(aes(colour = "t, df = 1"), fun = dt, args = list(df = 1))

# Using a custom anonymous function
base + geom_function(fun = \(x) 0.5 * exp(-abs(x)))
# or using lambda syntax:
# base + geom_function(fun = ~ 0.5 * exp(-abs(.x)))
# or using a custom named function:
# f <- function(x) 0.5 * exp(-abs(x))
# base + geom_function(fun = f)

# Using xlim to restrict the range of function
ggplot(data.frame(x = rnorm(100)), aes(x)) +
geom_density() +
geom_function(fun = dnorm, colour = "red", xlim=c(-1, 1))

# Using xlim to widen the range of function
ggplot(data.frame(x = rnorm(100)), aes(x)) +
geom_density() +
geom_function(fun = dnorm, colour = "red", xlim=c(-7, 7))
```

`geom_hex` *Hexagonal heatmap of 2d bin counts*

Description

Divides the plane into regular hexagons, counts the number of cases in each hexagon, and then (by default) maps the number of cases to the hexagon fill. Hexagon bins avoid the visual artefacts sometimes generated by the very regular alignment of `geom_bin_2d()`.

Usage

```
geom_hex(
  mapping = NULL,
  data = NULL,
  stat = "binhex",
  position = "identity",
  ...,
  lineend = "butt",
  linejoin = "mitre",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom, stat`	Override the default connection between `geom_hex()` and `stat_bin_hex()`. For more information about overriding these connections, see how the stat and geom arguments work.

`binwidth`	The width of the bins. Can be specified as a numeric value or as a function that takes x after scale transformation as input and returns a single numeric value. When specifying a function along with a grouping structure, the function will be called once per group. The default is to use the number of bins in `bins`, covering the range of the data. You should always override this value, exploring multiple widths to find the best to illustrate the stories in your data. The bin width of a date variable is the number of days in each time; the bin width of a time variable is the number of seconds.
`bins`	Number of bins. Overridden by `binwidth`. Defaults to 30.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(count)`
  number of points in bin.
- `after_stat(density)`
  density of points in bin, scaled to integrate to 1.
- `after_stat(ncount)`
  count, scaled to maximum of 1.
- `after_stat(ndensity)`
  density, scaled to maximum of 1.

Controlling binning parameters for the x and y directions

The arguments `bins` and `binwidth` can be set separately for the x and y directions. When given as a scalar, one value applies to both directions. When given as a vector of length two, the first is applied to the x direction and the second to the y direction. Alternatively, these can be a named list containing x and y elements, for example `list(x = 10, y = 20)`.

Aesthetics

`geom_hex()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`x`	
•	`y`	
•	`alpha`	→ NA
•	`colour`	→ via `theme()`
•	`fill`	→ via `theme()`
•	`group`	→ inferred
•	`linetype`	→ via `theme()`
•	`linewidth`	→ via `theme()`

`stat_binhex()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• x	
• y	
• fill	→ after_stat(count)
• group	→ inferred
• weight	→ 1

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

`stat_bin_2d()` for rectangular binning

Examples

```
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 *Jittered points*

Description

The jitter geom is a convenient shortcut for `geom_point(position = "jitter")`. It adds a small amount of random variation to the location of each point, and is a useful way of handling overplotting caused by discreteness in smaller datasets.

Usage

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

Arguments

`mapping`
: Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.

`data`
: The data to be displayed in this layer. There are three options:

  If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`.

  A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created.

  A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`stat`
: The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following:

  - A `Stat` ggproto subclass, for example `StatCount`.
  - A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.
  - For more information and other ways to specify the stat, see the layer stat documentation.

`position`
: A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

  - The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
  - A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
  - For more information and other ways to specify the position, see the layer position documentation.

`...`
: Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

  - Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`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.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Aesthetics

`geom_point()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`
- `y`
- `alpha` → NA
- `colour` → via `theme()`
- `fill` → via `theme()`
- `group` → inferred
- `shape` → via `theme()`
- `size` → via `theme()`
- `stroke` → via `theme()`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

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

Examples

```
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 *Text*

Description

Text geoms are useful for labeling plots. They can be used by themselves as scatterplots or in combination with other geoms, for example, for labeling points or for annotating the height of bars. `geom_text()` adds only text to the plot. `geom_label()` draws a rectangle behind the text, making it easier to read.

Usage

```
geom_label(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "nudge",
  ...,
  parse = FALSE,
  label.padding = unit(0.25, "lines"),
  label.r = unit(0.15, "lines"),
  label.size = deprecated(),
  border.colour = NULL,
  border.color = NULL,
  text.colour = NULL,
  text.color = NULL,
  size.unit = "mm",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_text(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "nudge",
  ...,
  parse = FALSE,
  check_overlap = FALSE,
  size.unit = "mm",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.

- For more information and other ways to specify the stat, see the layer stat documentation.

`position` A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...` Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`parse` If `TRUE`, the labels will be parsed into expressions and displayed as described in `?plotmath`.

`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` **[Deprecated]** Replaced by the `linewidth` aesthetic. Size of label border, in mm.

`border.colour`, `border.color`
Colour of label border. When `NULL` (default), the `colour` aesthetic determines the colour of the label border. `border.color` is an alias for `border.colour`.

`text.colour`, `text.color`
Colour of the text. When `NULL` (default), the `colour` aesthetic determines the colour of the text. `text.color` is an alias for `text.colour`.

`size.unit`	How the `size` aesthetic is interpreted: as millimetres (`"mm"`, default), points (`"pt"`), centimetres (`"cm"`), inches (`"in"`), or picas (`"pc"`).
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`check_overlap`	If `TRUE`, text that overlaps previous text in the same layer will not be plotted. `check_overlap` happens at draw time and in the order of the data. Therefore data should be arranged by the label column before calling `geom_text()`. Note that this argument is not supported by `geom_label()`.

Details

Note that when you resize a plot, text labels stay the same size, even though the size of the plot area changes. This happens because the "width" and "height" of a text element are 0. Obviously, text labels do have height and width, but they are physical units, not data units. For the same reason, stacking and dodging text will not work by default, and axis limits are not automatically expanded to include all text.

`geom_text()` and `geom_label()` add labels for each row in the data, even if coordinates x, y are set to single values in the call to `geom_label()` or `geom_text()`. To add labels at specified points use `annotate()` with `annotate(geom = "text", ...)` or `annotate(geom = "label", ...)`.

To automatically position non-overlapping text labels see the ggrepel package.

`geom_label()`

Currently `geom_label()` does not support the `check_overlap` argument. Also, it 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 (left/bottom) and 1 (right/top) 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.

Aesthetics

`geom_text()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`

• y	
• label	
• alpha	→ NA
• angle	→ 0
• colour	→ via theme()
• family	→ via theme()
• fontface	→ 1
• group	→ inferred
• hjust	→ 0.5
• lineheight	→ 1.2
• size	→ via theme()
• vjust	→ 0.5

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

The text labels section of the online ggplot2 book.

Examples

```
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:
# Doesn't work on all systems
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_hue(l = 40)
p + geom_label(aes(fill = factor(cyl)), colour = "white", fontface = "bold")

# Scale size of text, and change legend key glyph from a to point
p + geom_text(aes(size = wt), key_glyph = "point")
# Scale height of text, rather than sqrt(height)
p +
  geom_text(aes(size = wt), key_glyph = "point") +
  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 --------------------------------------------------
df <- data.frame(
  x = factor(c(1, 1, 2, 2)),
  y = c(1, 3, 2, 1),
  grp = c("a", "b", "a", "b")
)

# ggplot2 doesn't know you want to give the labels the same virtual width
# as the bars:
ggplot(data = df, aes(x, y, group = grp)) +
  geom_col(aes(fill = grp), position = "dodge") +
  geom_text(aes(label = y), position = "dodge")
# So tell it:
ggplot(data = df, aes(x, y, group = grp)) +
  geom_col(aes(fill = grp), position = "dodge") +
  geom_text(aes(label = y), position = position_dodge(0.9))
# You can't nudge and dodge text, so instead adjust the y position
ggplot(data = df, aes(x, y, group = grp)) +
  geom_col(aes(fill = grp), position = "dodge") +
  geom_text(
    aes(label = y, y = y + 0.05),
    position = position_dodge(0.9),
    vjust = 0
  )
```

```
# To place text in the middle of each bar in a stacked barplot, you
# need to set the vjust parameter of position_stack()
ggplot(data = df, aes(x, y, group = grp)) +
 geom_col(aes(fill = grp)) +
 geom_text(aes(label = y), position = position_stack(vjust = 0.5))

# Justification -------------------------------------------------------------
df <- data.frame(
  x = c(1, 1, 2, 2, 1.5),
  y = c(1, 2, 1, 2, 1.5),
  text = c("bottom-left", "top-left", "bottom-right", "top-right", "center")
)
ggplot(df, aes(x, y)) +
  geom_text(aes(label = text))
ggplot(df, aes(x, y)) +
  geom_text(aes(label = text), vjust = "inward", hjust = "inward")
```

geom_map *Polygons from a reference map*

Description

Display polygons as a map. This is meant as annotation, so it does not affect position scales. Note that this function predates the `geom_sf()` framework and does not work with sf geometry columns as input. However, it can be used in conjunction with `geom_sf()` layers and/or `coord_sf()` (see examples).

Usage

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.

`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`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`.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Aesthetics

`geom_map()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `map_id`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred
• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`
• `subgroup`	→ `NULL`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
# First, a made-up example containing a few polygons, to explain
# how `geom_map()` works. It requires two data frames:
# One contains the coordinates of each polygon (`positions`), and is
# provided via the `map` argument. The other contains 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)
ggplot(values, aes(fill = value)) +
  geom_map(aes(map_id = id), map = positions) +
  expand_limits(positions)
ggplot(values, aes(fill = value)) +
  geom_map(aes(map_id = id), map = positions) +
  expand_limits(positions) + ylim(0, 3)

# Now some examples with real maps
if (require(maps)) {

  crimes <- data.frame(state = tolower(rownames(USArrests)), USArrests)

  # Equivalent to crimes |> tidyr::pivot_longer(Murder:Rape)
  vars <- lapply(names(crimes)[-1], function(j) {
    data.frame(state = crimes$state, variable = j, value = crimes[[j]])
  })
  crimes_long <- do.call("rbind", vars)

  states_map <- map_data("state")

  # without geospatial coordinate system, the resulting plot
  # looks weird
  ggplot(crimes, aes(map_id = state)) +
    geom_map(aes(fill = Murder), map = states_map) +
    expand_limits(x = states_map$long, y = states_map$lat)

  # in combination with `coord_sf()` we get an appropriate result
  ggplot(crimes, aes(map_id = state)) +
    geom_map(aes(fill = Murder), map = states_map) +
    # crs = 5070 is a Conus Albers projection for North America,
    #   see: https://epsg.io/5070
    # default_crs = 4326 tells coord_sf() that the input map data
    #   are in longitude-latitude format
    coord_sf(
      crs = 5070, default_crs = 4326,
      xlim = c(-125, -70), ylim = c(25, 52)
    )

 ggplot(crimes_long, aes(map_id = state)) +
   geom_map(aes(fill = value), map = states_map) +
   coord_sf(
     crs = 5070, default_crs = 4326,
     xlim = c(-125, -70), ylim = c(25, 52)
   ) +
   facet_wrap(~variable)
}
```

geom_path *Connect observations*

Description

`geom_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. The `group` aesthetic determines which cases are connected together.

Usage

```
geom_path(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  arrow = NULL,
  arrow.fill = NULL,
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_line(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  orientation = NA,
  arrow = NULL,
  arrow.fill = NULL,
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_step(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  orientation = NA,
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  arrow = NULL,
  arrow.fill = NULL,
  direction = "hv",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.

`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`arrow`	Arrow specification, as created by `grid::arrow()`.
`arrow.fill`	fill colour to use for the arrow head (if closed). `NULL` means use `colour` aesthetic.
`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.
`direction`	direction of stairs: 'vh' for vertical then horizontal, 'hv' for horizontal then vertical, or 'mid' for step half-way between adjacent x-values.

Details

An alternative parameterisation is `geom_segment()`, where each line corresponds to a single case which provides the start and end coordinates.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Missing value handling

`geom_path()`, `geom_line()`, and `geom_step()` handle `NA` as follows:

- If an `NA` occurs in the middle of a line, it breaks the line. No warning is shown, regardless of whether `na.rm` is `TRUE` or `FALSE`.
- If an `NA` occurs at the start or the end of the line and `na.rm` is `FALSE` (default), the `NA` is removed with a warning.
- If an `NA` occurs at the start or the end of the line and `na.rm` is `TRUE`, the `NA` is removed silently, without warning.

Aesthetics

`geom_path()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`x`	
•	`y`	
•	`alpha`	→ NA
•	`colour`	→ via `theme()`
•	`group`	→ inferred
•	`linetype`	→ via `theme()`
•	`linewidth`	→ via `theme()`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

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()
# separate by colour and use "timeseries" legend key glyph
ggplot(economics_long, aes(date, value01, colour = variable)) +
  geom_line(key_glyph = "timeseries")

# You can get a timeseries that run vertically by setting the orientation
ggplot(economics, aes(unemploy, date)) + geom_line(orientation = "y")

# 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()
ggplot(recent, aes(date, unemploy)) + geom_step()

# 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 ----------------------------------------------
ggplot(economics, aes(date, unemploy)) +
  geom_line(colour = "red")

# 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")
)

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

# You can use NAs to break the line.
df <- data.frame(x = 1:5, y = c(1, 2, NA, 4, 5))
ggplot(df, aes(x, y)) + 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 *Points*

Description

The point geom is used to create scatterplots. The scatterplot is most useful for displaying the relationship between two continuous variables. It can be used to compare one continuous and one categorical variable, or two categorical variables, but a variation like `geom_jitter()`, `geom_count()`, or `geom_bin_2d()` is usually more appropriate. A *bubblechart* is a scatterplot with a third variable mapped to the size of points.

Usage

```
geom_point(
  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()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created.

A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).

`stat` The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following:

- A `Stat` ggproto subclass, for example `StatCount`.
- A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.
- For more information and other ways to specify the stat, see the layer stat documentation.

`position` A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...` Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Overplotting

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 `geom_count()`, `geom_hex()`, or `geom_density2d()`.

Another technique is to make the points transparent (e.g. `geom_point(alpha = 0.05)`) or very small (e.g. `geom_point(shape = ".")`).

Aesthetics

`geom_point()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`
- `y`
- `alpha` → `NA`
- `colour` → via `theme()`
- `fill` → via `theme()`
- `group` → inferred
- `shape` → via `theme()`
- `size` → via `theme()`
- `stroke` → via `theme()`

The `fill` aesthetic only applies to shapes 21-25.

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
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)))
# A "bubblechart":
p + geom_point(aes(size = qsec))

# 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)

# The default shape in legends is not filled, but you can override the shape
# in the guide to reflect the fill in the legend
ggplot(mtcars, aes(wt, mpg, fill = factor(carb), shape = factor(cyl))) +
  geom_point(size = 5, stroke = 1) +
  scale_shape_manual(values = 21:25) +
  scale_fill_ordinal(guide = guide_legend(override.aes = list(shape = 21)))


# 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)))

# 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
set.seed(1)
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 *Polygons*

Description

Polygons are very similar to paths (as drawn by geom_path()) except that the start and end points are connected and the inside is coloured by fill. The group aesthetic determines which cases are connected together into a polygon. From R 3.6 and onwards it is possible to draw polygons with holes by providing a subgroup aesthetic that differentiates the outer ring points from those describing holes in the polygon.

Usage

```
geom_polygon(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  rule = "evenodd",
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

mapping	Set of aesthetic mappings created by aes(). If specified and inherit.aes = TRUE (the default), it is combined with the default mapping at the top level of the plot. You must supply mapping if there is no plot mapping.
data	The data to be displayed in this layer. There are three options: If NULL, the default, the data is inherited from the plot data as specified in the call to ggplot(). A data.frame, or other object, will override the plot data. All objects will be fortified to produce a data frame. See fortify() for which variables will be created. A function will be called with a single argument, the plot data. The return value must be a data.frame, and will be used as the layer data. A function can be created from a formula (e.g. ~ head(.x, 10)).
stat	The statistical transformation to use on the data for this layer. When using a geom_*() function to construct a layer, the stat argument can be used to override the default coupling between geoms and stats. The stat argument accepts the following:

	• A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`rule`	Either `"evenodd"` or `"winding"`. If polygons with holes are being drawn (using the `subgroup` aesthetic) this argument defines how the hole coordinates are interpreted. See the examples in `grid::pathGrob()` for an explanation.
`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, missing values are removed with a warning. If TRUE, missing values are silently removed.
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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use TRUE. If NA, all levels are shown in legend, but unobserved levels are omitted.
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. annotation_borders().

Aesthetics

geom_polygon() understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• x	
• y	
• alpha	→ NA
• colour	→ via theme()
• fill	→ via theme()
• group	→ inferred
• linetype	→ via theme()
• linewidth	→ via theme()
• subgroup	→ NULL

Learn more about setting these aesthetics in vignette("ggplot2-specs").

See Also

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

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)
)

# 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))
p

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

set.seed(1)
stream <- data.frame(
  x = cumsum(runif(50, max = 0.1)),
  y = cumsum(runif(50,max = 0.1))
)

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

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

if (packageVersion("grid") >= "3.6") {
  # As of R version 3.6 geom_polygon() supports polygons with holes
  # Use the subgroup aesthetic to differentiate holes from the main polygon

  holes <- do.call(rbind, lapply(split(datapoly, datapoly$id), function(df) {
    df$x <- df$x + 0.5 * (mean(df$x) - df$x)
    df$y <- df$y + 0.5 * (mean(df$y) - df$y)
    df
  }))
  datapoly$subid <- 1L
  holes$subid <- 2L
  datapoly <- rbind(datapoly, holes)

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

geom_qq_line *A quantile-quantile plot*

Description

geom_qq() and stat_qq() produce quantile-quantile plots. geom_qq_line() and stat_qq_line()

compute the slope and intercept of the line connecting the points at specified quartiles of the theoretical and sample distributions.

Usage

```
geom_qq_line(
  mapping = NULL,
  data = NULL,
  geom = "abline",
  position = "identity",
  ...,
  distribution = stats::qnorm,
  dparams = list(),
  line.p = c(0.25, 0.75),
  fullrange = FALSE,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_qq_line(
  mapping = NULL,
  data = NULL,
  geom = "abline",
  position = "identity",
  ...,
  distribution = stats::qnorm,
  dparams = list(),
  line.p = c(0.25, 0.75),
  fullrange = FALSE,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_qq(
  mapping = NULL,
  data = NULL,
  geom = "point",
  position = "identity",
  ...,
  distribution = stats::qnorm,
  dparams = list(),
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_qq(
  mapping = NULL,
  data = NULL,
  geom = "point",
  position = "identity",
  ...,
  distribution = stats::qnorm,
  dparams = list(),
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`. • A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`. • For more information and other ways to specify the geom, see the layer geom documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.

`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`distribution`	Distribution function to use, if x not specified
`dparams`	Additional parameters passed on to `distribution` function.
`line.p`	Vector of quantiles to use when fitting the Q-Q line, defaults defaults to `c(.25, .75)`.
`fullrange`	Should the q-q line span the full range of the plot, or just the data
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation. Variables computed by `stat_qq()`:

- `after_stat(sample)`
  Sample quantiles.

- `after_stat(theoretical)`
  Theoretical quantiles.

Variables computed by `stat_qq_line()`:

- `after_stat(x)`
  x-coordinates of the endpoints of the line segment connecting the points at the chosen quantiles of the theoretical and the sample distributions.
- `after_stat(y)`
  y-coordinates of the endpoints.
- `after_stat(slope)`
  Amount of change in `y` across 1 unit of `x`.
- `after_stat(intercept)`
  Value of `y` at `x == 0`.

Aesthetics

`stat_qq()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`sample`	
•	`group`	→ inferred
•	`x`	→ `after_stat(theoretical)`
•	`y`	→ `after_stat(sample)`

`stat_qq_line()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`sample`	
•	`group`	→ inferred
•	`x`	→ `after_stat(x)`
•	`y`	→ `after_stat(y)`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
df <- data.frame(y = rt(200, df = 5))
p <- ggplot(df, aes(sample = y))
p + stat_qq() + stat_qq_line()

# Use fitdistr from MASS to estimate distribution params:
# if (requireNamespace("MASS", quietly = TRUE)) {
#   params <- as.list(MASS::fitdistr(df$y, "t")$estimate)
# }
# Here, we use pre-computed params
params <- list(m = -0.02505057194115, s = 1.122568610124, df = 6.63842653897)
```

```
ggplot(df, aes(sample = y)) +
  stat_qq(distribution = qt, dparams = params["df"]) +
  stat_qq_line(distribution = qt, dparams = params["df"])

# Using to explore the distribution of a variable
ggplot(mtcars, aes(sample = mpg)) +
  stat_qq() +
  stat_qq_line()
ggplot(mtcars, aes(sample = mpg, colour = factor(cyl))) +
  stat_qq() +
  stat_qq_line()
```

---

geom_quantile *Quantile regression*

Description

This fits a quantile regression to the data and draws the fitted quantiles with lines. This is as a continuous analogue to geom_boxplot().

Usage

```
geom_quantile(
  mapping = NULL,
  data = NULL,
  stat = "quantile",
  position = "identity",
  ...,
  arrow = NULL,
  arrow.fill = NULL,
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

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()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
data	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
position	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
...	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.

- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`arrow`	Arrow specification, as created by `grid::arrow()`.
`arrow.fill`	fill colour to use for the arrow head (if closed). `NULL` means use `colour` aesthetic.
`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom, stat`	Use to override the default connection between `geom_quantile()` and `stat_quantile()`. For more information about overriding these connections, see how the stat and geom arguments work.
`quantiles`	conditional quantiles of y to calculate and display
`formula`	formula relating y variables to x variables
`method`	Quantile regression method to use. Available options are `"rq"` (for `quantreg::rq()`) and `"rqss"` (for `quantreg::rqss()`).
`method.args`	List of additional arguments passed on to the modelling function defined by `method`.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(quantile)`
  Quantile of distribution.

Aesthetics

`geom_quantile()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• x	
• y	
• alpha	→ NA
• colour	→ via theme()
• group	→ inferred
• linetype	→ via theme()
• linewidth	→ via theme()
• weight	→ 1

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
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", linewidth = 2, alpha = 0.5)
```

`geom_raster` *Rectangles*

Description

`geom_rect()` and `geom_tile()` do the same thing, but are parameterised differently: `geom_tile()` uses the center of the tile and its size (`x`, `y`, `width`, `height`), while `geom_rect()` can use those or the locations of the corners (`xmin`, `xmax`, `ymin` and `ymax`). `geom_raster()` is a high performance special case for when all the tiles are the same size, and no pattern fills are applied.

Usage

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

geom_rect(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  lineend = "butt",
  linejoin = "mitre",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function`

can be created from a `formula` (e.g. `~ head(.x, 10)`).

`stat` The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following:

- A `Stat` ggproto subclass, for example `StatCount`.
- A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`.
- For more information and other ways to specify the stat, see the layer stat documentation.

`position` A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following:

- The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...` Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`interpolate` If `TRUE` interpolate linearly, if `FALSE` (the default) don't interpolate.

`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.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`lineend`	Line end style (round, butt, square).
`linejoin`	Line join style (round, mitre, bevel).

Details

Please note that the `width` and `height` aesthetics are not true position aesthetics and therefore are not subject to scale transformation. It is only after transformation that these aesthetics are applied.

Aesthetics

`geom_rect()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x` ***or*** `width` ***or*** `xmin` ***or*** `xmax`	
• `y` ***or*** `height` ***or*** `ymin` ***or*** `ymax`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred
• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`

`geom_tile()` understands only the `x`/`width` and `y`/`height` combinations. Note that `geom_raster()` ignores `colour`.

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

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(faithfuld, aes(waiting, eruptions)) +
 geom_raster(aes(fill = density))
```

```
# Interpolation smooths the surface & is most helpful when rendering images.
ggplot(faithfuld, 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), colour = "grey50")
ggplot(df, aes(x, y, width = w)) +
  geom_tile(aes(fill = z), colour = "grey50")
ggplot(df, aes(xmin = x - w / 2, xmax = x + w / 2, ymin = y, ymax = y + 1)) +
  geom_rect(aes(fill = z), colour = "grey50")


# Justification controls where the cells are anchored
df <- expand.grid(x = 0:5, y = 0:5)
set.seed(1)
df$z <- runif(nrow(df))
# default is compatible with geom_tile()
ggplot(df, aes(x, y, fill = z)) +
  geom_raster()
# zero padding
ggplot(df, aes(x, y, fill = z)) +
  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_bin_2d(aes(fill = after_stat(count)), binwidth = c(3,1))
cars + stat_bin_2d(aes(fill = after_stat(density)), binwidth = c(3,1))

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

geom_ribbon *Ribbons and area plots*

Description

For each x value, `geom_ribbon()` displays a y interval defined by `ymin` and `ymax`. `geom_area()` is a special case of `geom_ribbon()`, where the `ymin` is fixed to 0 and `y` is used instead of `ymax`.

Usage

```
geom_ribbon(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  orientation = NA,
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  outline.type = "both",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

geom_area(
  mapping = NULL,
  data = NULL,
  stat = "align",
  position = "stack",
  ...,
  orientation = NA,
  outline.type = "upper",
  lineend = "butt",
  linejoin = "round",
  linemitre = 10,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth`

= 3. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the params. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a stat_*() function, the ... argument can be used to pass on parameters to the geom part of the layer. An example of this is stat_density(geom = "area", outline.type = "both"). The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a geom_*() function, the ... argument can be used to pass on parameters to the stat part of the layer. An example of this is geom_area(stat = "density", adjust = 0.5). The stat's documentation lists which parameters it can accept.
- The key_glyph argument of layer() may also be passed on through .... This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

orientation	The orientation of the layer. The default (NA) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting orientation to either "x" or "y". See the *Orientation* section for more detail.
lineend	Line end style (round, butt, square).
linejoin	Line join style (round, mitre, bevel).
linemitre	Line mitre limit (number greater than 1).
outline.type	Type of the outline of the area; "both" draws both the upper and lower lines, "upper"/"lower" draws the respective lines only. "full" draws a closed polygon around the area.
na.rm	If FALSE, the default, missing values are removed with a warning. If TRUE, missing values are silently removed.
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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use TRUE. If NA, all levels are shown in legend, but unobserved levels are omitted.
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. annotation_borders().
geom	The geometric object to use to display the data for this layer. When using a stat_*() function to construct a layer, the geom argument can be used to override the default coupling between stats and geoms. The geom argument accepts the following:

- A Geom ggproto subclass, for example GeomPoint.
- A string naming the geom. To give the geom as a string, strip the function name of the geom_ prefix. For example, to use geom_point(), give the geom as "point".
- For more information and other ways to specify the geom, see the layer geom documentation.

Details

An area plot is the continuous analogue 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. See `position_stack()` for the details of stacking algorithm. To facilitate stacking, the default `stat = "align"` interpolates groups to a common set of x-coordinates. To turn off this interpolation, `stat = "identity"` can be used instead.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Aesthetics

`geom_ribbon()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x` ***or*** `y`	
• `ymin` ***or*** `xmin`	
• `ymax` ***or*** `xmax`	
• `alpha`	→ NA
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred
• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

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

Examples

```
# 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))
```

```
# Orientation cannot be deduced by mapping, so must be given explicitly for
# flipped orientation
h + geom_area(aes(x = level, y = year), orientation = "y")

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

# The underlying stat_align() takes care of unaligned data points
df <- data.frame(
  g = c("a", "a", "a", "b", "b", "b"),
  x = c(1, 3, 5, 2, 4, 6),
  y = c(2, 5, 1, 3, 6, 7)
)
a <- ggplot(df, aes(x, y, fill = g)) +
  geom_area()

# Two groups have points on different X values.
a + geom_point(size = 8) + facet_grid(g ~ .)

# stat_align() interpolates and aligns the value so that the areas can stack
# properly.
a + geom_point(stat = "align", position = "stack", size = 8)

# To turn off the alignment, the stat can be set to "identity"
ggplot(df, aes(x, y, fill = g)) +
  geom_area(stat = "identity")
```

geom_rug *Rug plots in the margins*

Description

A rug plot is a compact visualisation designed to supplement a 2d display with the two 1d marginal distributions. Rug plots display individual cases so are best used with smaller datasets.

Usage

```
geom_rug(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  lineend = "butt",
  sides = "bl",
  outside = FALSE,
  length = unit(0.03, "npc"),
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`lineend`	Line end style (round, butt, square).
`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.
`outside`	logical that controls whether to move the rug tassels outside of the plot area. Default is off (FALSE). You will also need to use `coord_cartesian(clip = "off")`. When set to TRUE, also consider changing the sides argument to "tr". See examples.
`length`	A `grid::unit()` object that sets the length of the rug lines. Use scale expansion to avoid overplotting of data.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Details

By default, the rug lines are drawn with a length that corresponds to 3% of the total plot size. Since the default scale expansion of for continuous variables is 5% at both ends of the scale, the rug will not overlap with any data points under the default settings.

Aesthetics

`geom_rug()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• alpha	→ NA
• colour	→ via theme()
• group	→ inferred
• linetype	→ via theme()
• linewidth	→ via theme()
• x	
• y	

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

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

# Use jittering to avoid overplotting for smaller datasets
ggplot(mpg, aes(displ, cty)) +
  geom_point() +
  geom_rug()

ggplot(mpg, aes(displ, cty)) +
  geom_jitter() +
  geom_rug(alpha = 1/2, position = "jitter")

# move the rug tassels to outside the plot
# remember to set clip = "off".
p +
  geom_rug(outside = TRUE) +
  coord_cartesian(clip = "off")

# set sides to top right, and then move the margins
p +
  geom_rug(outside = TRUE, sides = "tr") +
  coord_cartesian(clip = "off") +
  theme(plot.margin = margin_auto(1, unit = "cm"))

# increase the line length and
# expand axis to avoid overplotting
p +
  geom_rug(length = unit(0.05, "npc")) +
  scale_y_continuous(expand = c(0.1, 0.1))
```

geom_segment *Line segments and curves*

Description

geom_segment() draws a straight line between points (x, y) and (xend, yend). geom_curve() draws a curved line. See the underlying drawing function grid::curveGrob() for the parameters that control the curve.

Usage

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

geom_curve(
  mapping = NULL,
  data = NULL,
  stat = "identity",
  position = "identity",
  ...,
  curvature = 0.5,
  angle = 90,
  ncp = 5,
  arrow = NULL,
  arrow.fill = NULL,
  lineend = "butt",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

mapping	Set of aesthetic mappings created by aes(). If specified and inherit.aes = TRUE (the default), it is combined with the default mapping at the top level of the plot. You must supply mapping if there is no plot mapping.

`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer.

An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.

- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`arrow`	specification for arrow heads, as created by `grid::arrow()`.
`arrow.fill`	fill colour to use for the arrow head (if closed). `NULL` means use `colour` aesthetic.
`lineend`	Line end style (round, butt, square).
`linejoin`	Line join style (round, mitre, bevel).
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`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.

Details

Both geoms draw a single segment/curve per case. See `geom_path()` if you need to connect points across multiple cases.

Aesthetics

`geom_segment()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `xend` ***or*** `yend`	
• `alpha`	→ NA
• `colour`	→ via `theme()`
• `group`	→ inferred

- `linetype` → via `theme()`
- `linewidth` → via `theme()`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

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"))
)

if (requireNamespace('maps', quietly = TRUE)) {
ggplot(seals, aes(long, lat)) +
  geom_segment(aes(xend = long + delta_long, yend = lat + delta_lat),
    arrow = arrow(length = unit(0.1,"cm"))) +
  annotation_borders("state")
}

# Use lineend and linejoin to change the style of the segments
df2 <- expand.grid(
  lineend = c('round', 'butt', 'square'),
  linejoin = c('round', 'mitre', 'bevel'),
  stringsAsFactors = FALSE
)
df2 <- data.frame(df2, y = 1:9)
ggplot(df2, aes(x = 1, y = y, xend = 2, yend = y, label = paste(lineend, linejoin))) +
  geom_segment(
     lineend = df2$lineend, linejoin = df2$linejoin,
     size = 3, arrow = arrow(length = unit(0.3, "inches"))
  ) +
  geom_text(hjust = 'outside', nudge_x = -0.2) +
  xlim(0.5, 2)
```

```
# You can also use geom_segment to recreate plot(type = "h") :
set.seed(1)
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), linewidth = 10, lineend = "butt")
```

---

geom_smooth *Smoothed conditional means*

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 `stat_smooth()` if you want to display the results with a non-standard geom.

Usage

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

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.

- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`method`	Smoothing method (function) to use, accepts either `NULL` or a character vector, e.g. `"lm"`, `"glm"`, `"gam"`, `"loess"` or a function, e.g. `MASS::rlm` or `mgcv::gam`, `stats::lm`, or `stats::loess`. `"auto"` is also accepted for backwards compatibility. It is equivalent to `NULL`. For `method = NULL` the smoothing method is chosen based on the size of the largest group (across all panels). `stats::loess()` is used for less than 1,000 observations; otherwise `mgcv::gam()` is used with `formula = y ~ s(x, bs = "cs")` with `method = "REML"`. Somewhat anecdotally, `loess` gives a better appearance, but is $O(N^2)$ in memory, so does not work for larger datasets. If you have fewer than 1,000 observations but want to use the same `gam()` model that `method = NULL` would use, then set `method = "gam", formula = y ~ s(x, bs = "cs")`.
`formula`	Formula to use in smoothing function, eg. `y ~ x`, `y ~ poly(x, 2)`, `y ~ log(x)`. `NULL` by default, in which case `method = NULL` implies `formula = y ~ x` when there are fewer than 1,000 observations and `formula = y ~ s(x, bs = "cs")` otherwise.
`se`	Display confidence band around smooth? (`TRUE` by default, see `level` to control.)
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`geom, stat`	Use to override the default connection between `geom_smooth()` and `stat_smooth()`. For more information about overriding these connections, see how the stat and geom arguments work.
`n`	Number of points at which to evaluate smoother.
`span`	Controls the amount of smoothing for the default loess smoother. Smaller numbers produce wigglier lines, larger numbers produce smoother lines. Only used with loess, i.e. when `method = "loess"`, or when `method = NULL` (the default) and there are fewer than 1,000 observations.

`fullrange`	If `TRUE`, the smoothing line gets expanded to the range of the plot, potentially beyond the data. This does not extend the line into any additional padding created by `expansion`.
`xseq`	A numeric vector of values at which the smoother is evaluated. When `NULL` (default), `xseq` is internally evaluated as a sequence of `n` equally spaced points for continuous data.
`level`	Level of confidence band 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 band is constructed on the link scale and then back-transformed to the response scale.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation. `stat_smooth()` provides the following variables, some of which depend on the orientation:

- `after_stat(y)` *or* `after_stat(x)`
  Predicted value.
- `after_stat(ymin)` *or* `after_stat(xmin)`
  Lower pointwise confidence band around the mean.
- `after_stat(ymax)` *or* `after_stat(xmax)`
  Upper pointwise confidence band around the mean.
- `after_stat(se)`
  Standard error.

Aesthetics

`geom_smooth()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`
- `y`

•	`alpha`	→ `0.4`
•	`colour`	→ via `theme()`
•	`fill`	→ via `theme()`
•	`group`	→ inferred
•	`linetype`	→ via `theme()`
•	`linewidth`	→ via `theme()`
•	`weight`	→ `1`
•	`ymax`	
•	`ymin`	

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

See individual modelling functions for more details: `lm()` for linear smooths, `glm()` for generalised linear smooths, and `loess()` for local smooths.

Examples

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

# If you need the fitting to be done along the y-axis set the orientation
ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(orientation = "y")

# 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)

# Smooths 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)


binomial_smooth <- function(...) {
  geom_smooth(method = "glm", method.args = list(family = "binomial"), ...)
}
# To fit a logistic regression, you need to coerce the values to
# a numeric vector lying between 0 and 1.
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.
```

geom_spoke *Line segments parameterised by location, direction and distance*

Description

This is a polar parameterisation of `geom_segment()`. It is useful when you have variables that describe direction and distance. The angles start from east and increase counterclockwise.

Usage

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`stat`	The statistical transformation to use on the data for this layer. When using a `geom_*()` function to construct a layer, the `stat` argument can be used to override the default coupling between geoms and stats. The `stat` argument accepts the following: • A `Stat` ggproto subclass, for example `StatCount`. • A string naming the stat. To give the stat as a string, strip the function name of the `stat_` prefix. For example, to use `stat_count()`, give the stat as `"count"`. • For more information and other ways to specify the stat, see the layer stat documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is

technically possible, the order and required length is not guaranteed to be parallel to the input data.

- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`arrow`	specification for arrow heads, as created by `grid::arrow()`.
`arrow.fill`	fill colour to use for the arrow head (if closed). `NULL` means use `colour` aesthetic.
`lineend`	Line end style (round, butt, square).
`linejoin`	Line join style (round, mitre, bevel).
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Aesthetics

`geom_spoke()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `angle`	
• `radius`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `group`	→ inferred
• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
df <- expand.grid(x = 1:10, y=1:10)

set.seed(1)
df$angle <- runif(100, 0, 2*pi)
df$speed <- runif(100, 0, sqrt(0.1 * df$x))

ggplot(df, aes(x, y)) +
  geom_point() +
  geom_spoke(aes(angle = angle), radius = 0.5)

ggplot(df, aes(x, y)) +
  geom_point() +
  geom_spoke(aes(angle = angle, radius = speed))
```

`geom_violin` *Violin plot*

Description

A violin plot is a compact display of a continuous distribution. It is a blend of `geom_boxplot()` and `geom_density()`: a violin plot is a mirrored density plot displayed in the same way as a boxplot.

Usage

```
geom_violin(
  mapping = NULL,
  data = NULL,
  stat = "ydensity",
  position = "dodge",
  ...,
  trim = TRUE,
  bounds = c(-Inf, Inf),
  quantile.colour = NULL,
  quantile.color = NULL,
  quantile.linetype = 0L,
  quantile.linewidth = NULL,
  draw_quantiles = deprecated(),
  scale = "area",
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE
)

stat_ydensity(
  mapping = NULL,
  data = NULL,
  geom = "violin",
  position = "dodge",
  ...,
  orientation = NA,
  bw = "nrd0",
  adjust = 1,
  kernel = "gaussian",
  trim = TRUE,
  scale = "area",
  drop = TRUE,
  bounds = c(-Inf, Inf),
  quantiles = c(0.25, 0.5, 0.75),
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`trim` If `TRUE` (default), trim the tails of the violins to the range of the data. If `FALSE`, don't trim the tails.

`bounds` Known lower and upper bounds for estimated data. Default `c(-Inf, Inf)` means that there are no (finite) bounds. If any bound is finite, boundary effect of default density estimation will be corrected by reflecting tails outside `bounds` around their closest edge. Data points outside of bounds are removed with a warning.

`quantile.colour`, `quantile.color`, `quantile.linewidth`, `quantile.linetype`
Default aesthetics for the quantile lines. Set to `NULL` to inherit from the data's aesthetics. By default, quantile lines are hidden and can be turned on by changing `quantile.linetype`.

`draw_quantiles` **[Deprecated]** Previous specification of drawing quantiles.

`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, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`orientation` The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`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. `annotation_borders()`.
`geom, stat`	Use to override the default connection between `geom_violin()` and `stat_ydensity()`. For more information about overriding these connections, see how the stat and geom arguments work.
`bw`	The smoothing bandwidth to be used. If numeric, the standard deviation of the smoothing kernel. If character, a rule to choose the bandwidth, as listed in `stats::bw.nrd()`. Note that automatic calculation of the bandwidth does not take weights into account.
`adjust`	A multiplicate bandwidth adjustment. This makes it possible to adjust the bandwidth while still using the a bandwidth estimator. For example, `adjust = 1/2` means use half of the default bandwidth.
`kernel`	Kernel. See list of available kernels in `density()`.
`drop`	Whether to discard groups with less than 2 observations (`TRUE`, default) or keep such groups for position adjustment purposes (`FALSE`).
`quantiles`	If not `NULL` (default), compute the `quantile` variable and draw horizontal lines at the given quantiles in `geom_violin()`.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(density)`
  Density estimate.
- `after_stat(scaled)`
  Density estimate, scaled to a maximum of 1.
- `after_stat(count)`
  Density * number of points - probably useless for violin plots.
- `after_stat(violinwidth)`
  Density scaled for the violin plot, according to area, counts or to a constant maximum width.
- `after_stat(n)`
  Number of points.
- `after_stat(width)`
  Width of violin bounding box.
- `after_stat(quantile)`
  Whether the row is part of the `quantiles` computation.

Aesthetics

`geom_violin()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

• `x`	
• `y`	
• `alpha`	→ `NA`
• `colour`	→ via `theme()`
• `fill`	→ via `theme()`
• `group`	→ inferred
• `linetype`	→ via `theme()`
• `linewidth`	→ via `theme()`
• `weight`	→ `1`
• `width`	→ `0.9`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

References

Hintze, J. L., Nelson, R. D. (1998) Violin Plots: A Box Plot-Density Trace Synergism. The American Statistician 52, 181-184.

See Also

`geom_violin()` for examples, and `stat_density()` for examples with data along the x axis.

Examples

```
p <- ggplot(mtcars, aes(factor(cyl), mpg))
p + geom_violin()

# Orientation follows the discrete axis
ggplot(mtcars, aes(mpg, factor(cyl))) +
  geom_violin()


p + geom_violin() + geom_jitter(height = 0, width = 0.1)

# 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_transform(y = "log10")
m +
  geom_violin() +
  scale_y_log10() + coord_transform(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")
}
```

`get_alt_text` *Extract alt text from a plot*

Description

This function returns a text that can be used as alt-text in webpages etc. Currently it will use the `alt` label, added with `+ labs(alt = <...>)`, or a return an empty string, but in the future it might try to generate an alt text from the information stored in the plot.

Usage

```
get_alt_text(p, ...)
```

Arguments

p	a ggplot object
...	Arguments passed to methods.

Value

A text string

Examples

```
p <- ggplot(mpg, aes(displ, hwy)) +
  geom_point()

# Returns an empty string
get_alt_text(p)

# A user provided alt text
p <- p + labs(
  alt = paste("A scatterplot showing the negative correlation between engine",
              "displacement as a function of highway miles per gallon")
)

get_alt_text(p)
```

`get_theme` *Get, set, and modify the active theme*

Description

The current/active theme (see `theme()`) is automatically applied to every plot you draw. Use `get_theme()` to get the current theme, and `set_theme()` to completely override it. `update_theme()` and `replace_theme()` are shorthands for changing individual elements.

Usage

```
get_theme()

theme_get()

set_theme(new)

theme_set(new)
```

```
update_theme(...)

theme_update(...)

replace_theme(...)

theme_replace(...)

e1 %+replace% e2
```

Arguments

`new`	new theme (a list of theme elements)
`...`	named list of theme settings
`e1, e2`	Theme and element to combine

Value

`set_theme()`, `update_theme()`, and `replace_theme()` invisibly return the previous theme so you can easily save it, then later restore it.

Adding on to a theme

+ and `%+replace%` can be used to modify elements in themes.

+ 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, `%+replace%` 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.

`update_theme()` uses the `+` operator, so that any unspecified values in the theme element will default to the values they are set in the theme. `replace_theme()` uses `%+replace%` to completely replace the element, so any unspecified values will overwrite the current value in the theme with `NULL`.

In summary, the main differences between `set_theme()`, `update_theme()`, and `replace_theme()` are:

- `set_theme()` completely overrides the current theme.
- `update_theme()` modifies a particular element of the current theme using the + operator.
- `replace_theme()` modifies a particular element of the current theme using the `%+replace%` operator.

See Also

`add_gg()`

Examples

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

# Use set_theme() to completely override the current theme.
# update_theme() and replace_theme() are similar except they
# apply directly to the current/active theme.
# update_theme() modifies a particular element of the current theme.
# Here we have the old theme so we can later restore it.
# Note that the theme is applied when the plot is drawn, not
# when it is created.
old <- set_theme(theme_bw())
p

set_theme(old)
update_theme(panel.grid.minor = element_line(colour = "red"))
p

set_theme(old)
replace_theme(panel.grid.minor = element_line(colour = "red"))
p

set_theme(old)
p


# Modifying theme objects -----------------------------------------
# You can use + and %+replace% to modify a theme object.
# They differ in how they deal with missing arguments in
# the theme elements.

add_el <- theme_grey() +
  theme(text = element_text(family = "Times"))
add_el$text

rep_el <- theme_grey() %+replace%
  theme(text = element_text(family = "Times"))
rep_el$text
```

`ggplot` *Create a new ggplot*

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`	**[Deprecated]** Used prior to tidy evaluation.

Details

`ggplot()` is used to construct the initial plot object, and is almost always followed by a plus sign (+) to add components to the plot.

There are three common patterns used to invoke `ggplot()`:

- `ggplot(data = df, mapping = aes(x, y, other aesthetics))`
- `ggplot(data = df)`
- `ggplot()`

The first pattern is recommended if all layers use the same data and the same set of aesthetics, although this method can also be used when adding a layer using data from another data frame.

The second pattern 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 for the plot, but the aesthetics vary from one layer to another.

The third pattern initializes a skeleton `ggplot` object, which is fleshed out as layers are added. This is useful when multiple data frames are used to produce different layers, as is often the case in complex graphics.

The `data =` and `mapping =` specifications in the arguments are optional (and are often omitted in practice), so long as the data and the mapping values are passed into the function in the right order. In the examples below, however, they are left in place for clarity.

See Also

The first steps chapter of the online ggplot2 book.

Examples

```
# Create a data frame with some sample data, then create a data frame
# containing the mean value for each group in the sample data.
set.seed(1)

sample_df <- data.frame(
  group = factor(rep(letters[1:3], each = 10)),
  value = rnorm(30)
)

group_means_df <- setNames(
  aggregate(value ~ group, sample_df, mean),
  c("group", "group_mean")
)

# The following three code blocks create the same graphic, each using one
# of the three patterns specified above. In each graphic, the sample data
# are plotted in the first layer and the group means data frame is used to
# plot larger red points on top of the sample data in the second layer.

# Pattern 1
# Both the `data` and `mapping` arguments are passed into the `ggplot()`
# call. Those arguments are omitted in the first `geom_point()` layer
# because they get passed along from the `ggplot()` call. Note that the
# second `geom_point()` layer re-uses the `x = group` aesthetic through
# that mechanism but overrides the y-position aesthetic.
ggplot(data = sample_df, mapping = aes(x = group, y = value)) +
  geom_point() +
  geom_point(
    mapping = aes(y = group_mean), data = group_means_df,
    colour = 'red', size = 3
  )

# Pattern 2
# Same plot as above, passing only the `data` argument into the `ggplot()`
# call. The `mapping` arguments are now required in each `geom_point()`
# layer because there is no `mapping` argument passed along from the
# `ggplot()` call.
ggplot(data = sample_df) +
  geom_point(mapping = aes(x = group, y = value)) +
  geom_point(
    mapping = aes(x = group, y = group_mean), data = group_means_df,
    colour = 'red', size = 3
  )

# Pattern 3
# Same plot as above, passing neither the `data` or `mapping` arguments
# into the `ggplot()` call. Both those arguments are now required in
# each `geom_point()` layer. This pattern can be particularly useful when
# creating more complex graphics with many layers using data from multiple
# data frames.
ggplot() +
  geom_point(mapping = aes(x = group, y = value), data = sample_df) +
  geom_point(
    mapping = aes(x = group, y = group_mean), data = group_means_df,
    colour = 'red', size = 3
  )
```

ggproto   *Create a new ggproto object*

Description

Construct a new object with `ggproto()`, test with `is_ggproto()`, and access parent methods/fields with `ggproto_parent()`.

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. This is optional: 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 named members in the ggproto object. These can be functions that become methods of the class or regular objects.
`parent, self`	Access parent class `parent` of object `self`.

Details

ggproto implements a protype based OO system which blurs the lines between classes and instances. It is inspired by the proto package, but it has some important differences. Notably, it cleanly supports cross-package inheritance, and has faster performance.

In most cases, creating a new OO system to be used by a single package is not a good idea. However, it was the least-bad solution for ggplot2 because it required the fewest changes to an already complex code base.

Calling 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)`.

Working with ggproto classes

The ggproto objects constructed are build on top of environments, which has some ramifications. Environments do not follow the 'copy on modify' semantics one might be accustomed to in regular objects. Instead they have 'modify in place' semantics.

See Also

The ggproto introduction section of the online ggplot2 book.

Examples

```
Adder <- ggproto("Adder",
  x = 0,
  add = function(self, n) {
    self$x <- self$x + n
    self$x
  }
 )
is_ggproto(Adder)

Adder$add(10)
Adder$add(10)

Doubler <- ggproto("Doubler", Adder,
  add = function(self, n) {
    ggproto_parent(Adder, self)$add(n * 2)
  }
)
Doubler$x
Doubler$add(10)
```

---

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 = get_last_plot(),
  device = NULL,
  path = NULL,
  scale = 1,
  width = NA,
  height = NA,
  units = c("in", "cm", "mm", "px"),
  dpi = 300,
  limitsize = TRUE,
  bg = NULL,
  create.dir = FALSE,
  ...
)
```

Arguments

`filename`	File name to create on disk.
`plot`	Plot to save, defaults to last plot displayed.
`device`	Device to use. Can either be a device function (e.g. png), or one of "eps", "ps", "tex" (pictex), "pdf", "jpeg", "tiff", "png", "bmp", "svg" or "wmf" (windows only). If `NULL` (default), the device is guessed based on the `filename` extension.
`path`	Path of the directory to save plot to: `path` and `filename` are combined to create the fully qualified file name. Defaults to the working directory.
`scale`	Multiplicative scaling factor.
`width, height`	Plot size in units expressed by the `units` argument. If not supplied, uses the size of the current graphics device.
`units`	One of the following units in which the `width` and `height` arguments are expressed: `"in"`, `"cm"`, `"mm"` or `"px"`.
`dpi`	Plot resolution. Also accepts a string input: "retina" (320), "print" (300), or "screen" (72). Only applies when converting pixel units, as is typical for raster output types.
`limitsize`	When `TRUE` (the default), `ggsave()` will not save images larger than 50x50 inches, to prevent the common error of specifying dimensions in pixels.
`bg`	Background colour. If `NULL`, uses the `plot.background` fill value from the plot theme.
`create.dir`	Whether to create new directories if a non-existing directory is specified in the `filename` or `path` (`TRUE`) or return an error (`FALSE`, default). If `FALSE` and run in an interactive session, a prompt will appear asking to create a new directory when necessary.
`...`	Other arguments passed on to the graphics device function, as specified by `device`.

Details

Note: Filenames with page numbers can be generated by including a C integer format expression, such as `%03d` (as in the default file name for most R graphics devices, see e.g. `png()`). Thus, `filename = "figure%03d.png"` will produce successive filenames `figure001.png`, `figure002.png`, `figure003.png`, etc. To write a filename containing the `%` sign, use `%%`. For example, `filename = "figure-100%%.png"` will produce the filename `figure-100%.png`.

Saving images without ggsave()

In most cases `ggsave()` is the simplest way to save your plot, but sometimes you may wish to save the plot by writing directly to a graphics device. To do this, you can open a regular R graphics device such as `png()` or `pdf()`, print the plot, and then close the device using `dev.off()`. This technique is illustrated in the examples section.

See Also

The saving section of the online ggplot2 book.

Examples

```
## Not run:
ggplot(mtcars, aes(mpg, wt)) +
  geom_point()

# here, the device is inferred from the filename extension
ggsave("mtcars.pdf")
ggsave("mtcars.png")

# setting dimensions of the plot
ggsave("mtcars.pdf", width = 4, height = 4)
ggsave("mtcars.pdf", width = 20, height = 20, units = "cm")

# passing device-specific arguments to '...'
ggsave("mtcars.pdf", colormodel = "cmyk")

# delete files with base::unlink()
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)

# save plot to file without using ggsave
p <-
  ggplot(mtcars, aes(mpg, wt)) +
  geom_point()
png("mtcars.png")
print(p)
dev.off()


## End(Not run)
```

ggtheme *Complete themes*

Description

These are complete themes which control all non-data display. Use theme() if you just need to tweak the display of an existing theme.

Usage

```
theme_grey(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)

theme_gray(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)

theme_bw(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)

theme_linedraw(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)

theme_light(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)

theme_dark(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)

theme_minimal(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)

theme_classic(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)

theme_void(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = alpha(ink, 0),
  accent = "#3366FF"
)

theme_test(
  base_size = 11,
  base_family = "",
  header_family = NULL,
  base_line_size = base_size/22,
  base_rect_size = base_size/22,
  ink = "black",
  paper = "white",
  accent = "#3366FF"
)
```

Arguments

`base_size`	base font size, given in pts.
`base_family`	base font family
`header_family`	font family for titles and headers. The default, `NULL`, uses theme inheritance to set the font. This setting affects axis titles, legend titles, the plot title and tag text.
`base_line_size`	base size for line elements
`base_rect_size`	base size for rect elements
`ink, paper, accent`	colour for foreground, background, and accented elements respectively.

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 drawing. 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.

`theme_test()` A theme for visual unit tests. It should ideally never change except for new features.

See Also

The complete themes section of the online ggplot2 book.

Examples

```
mtcars2 <- within(mtcars, {
  vs <- factor(vs, labels = c("V-shaped", "Straight"))
  am <- factor(am, labels = c("Automatic", "Manual"))
  cyl  <- factor(cyl)
  gear <- factor(gear)
})

p1 <- ggplot(mtcars2) +
  geom_point(aes(x = wt, y = mpg, colour = gear)) +
  labs(
    title = "Fuel economy declines as weight increases",
    subtitle = "(1973-74)",
    caption = "Data from the 1974 Motor Trend US magazine.",
    tag = "Figure 1",
    x = "Weight (1000 lbs)",
    y = "Fuel economy (mpg)",
    colour = "Gears"
  )

p1 + theme_gray() # the default
p1 + theme_bw()
p1 + theme_linedraw()
p1 + theme_light()
p1 + theme_dark()
p1 + theme_minimal()
p1 + theme_classic()
p1 + theme_void()

# Theme examples with panels

p2 <- p1 + facet_grid(vs ~ am)

p2 + theme_gray() # the default
p2 + theme_bw()
p2 + theme_linedraw()
```

```
p2 + theme_light()
p2 + theme_dark()
p2 + theme_minimal()
p2 + theme_classic()
p2 + theme_void()
```

`guides` *Set guides for each scale*

Description

Guides for each scale can be set scale-by-scale with the `guide` argument, or en masse with `guides()`.

Usage

```
guides(...)
```

Arguments

`...`	List of scale name-guide pairs. The guide can either be a string (i.e. "colorbar" or "legend"), or a call to a guide function (i.e. `guide_colourbar()` or `guide_legend()`) specifying additional arguments.

Value

A list containing the mapping between scale and guide.

See Also

Other guides: `guide_bins`(), `guide_colourbar`(), `guide_coloursteps`(), `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

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)

p + theme(legend.position = "bottom")

# position of guides

# Set order for multiple guides
ggplot(mpg, aes(displ, cty)) +
  geom_point(aes(size = hwy, colour = cyl, shape = drv)) +
  guides(
   colour = guide_colourbar(order = 1),
   shape = guide_legend(order = 2),
   size = guide_legend(order = 3)
 )
```

guide_axis *Axis guide*

Description

Axis guides are the visual representation of position scales like those created with scale_(xly)_continuous() and scale_(xly)_discrete().

Usage

```
guide_axis(
  title = waiver(),
  theme = NULL,
  check.overlap = FALSE,
  angle = waiver(),
  n.dodge = 1,
  minor.ticks = FALSE,
  cap = "none",
  order = 0,
  position = waiver()
)
```

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.
`theme`	A `theme` object to style the guide individually or differently from the plot's theme settings. The `theme` argument in the guide partially overrides, and is combined with, the plot's theme.
`check.overlap`	silently remove overlapping labels, (recursively) prioritizing the first, last, and middle labels.
`angle`	Compared to setting the angle in `theme()` / `element_text()`, this also uses some heuristics to automatically pick the `hjust` and `vjust` that you probably want. Can be one of the following: • `NULL` to take the angles and `hjust`/`vjust` directly from the theme. • `waiver()` to allow reasonable defaults in special cases. • A number representing the text angle in degrees.
`n.dodge`	The number of rows (for vertical axes) or columns (for horizontal axes) that should be used to render the labels. This is useful for displaying labels that would otherwise overlap.
`minor.ticks`	Whether to draw the minor ticks (`TRUE`) or not draw minor ticks (`FALSE`, default).
`cap`	A `character` to cut the axis line back to the last breaks. Can be `"none"` (default) to draw the axis line along the whole panel, or `"upper"` and `"lower"` to draw the axis to the upper or lower break, or `"both"` to only draw the line in between the most extreme breaks. `TRUE` and `FALSE` are shorthand for `"both"` and `"none"` respectively.
`order`	A positive `integer` of length 1 that specifies the order of this guide among multiple guides. This controls in which order guides are merged if there are multiple guides for the same position. If 0 (default), the order is determined by a secret algorithm.
`position`	Where this guide should be drawn: one of top, bottom, left, or right.

Examples

```
# plot with overlapping text
p <- ggplot(mpg, aes(cty * 100, hwy * 100)) +
  geom_point() +
  facet_wrap(vars(class))
```

```
# axis guides can be customized in the scale_* functions or
# using guides()
p + scale_x_continuous(guide = guide_axis(n.dodge = 2))
p + guides(x = guide_axis(angle = 90))

# can also be used to add a duplicate guide
p + guides(x = guide_axis(n.dodge = 2), y.sec = guide_axis())
```

guide_axis_logticks *Axis with logarithmic tick marks*

Description

This axis guide replaces the placement of ticks marks at intervals in log10 space.

Usage

```
guide_axis_logticks(
  long = 2.25,
  mid = 1.5,
  short = 0.75,
  prescale.base = NULL,
  negative.small = NULL,
  short.theme = element_line(),
  expanded = TRUE,
  cap = "none",
  theme = NULL,
  prescale_base = deprecated(),
  negative_small = deprecated(),
  short_theme = deprecated(),
  ...
)
```

Arguments

`long, mid, short`	A `grid::unit()` object or `rel()` object setting the (relative) length of the long, middle and short ticks. Numeric values are interpreted as `rel()` objects. The `rel()` values are used to multiply values of the `axis.ticks.length` theme setting.
`prescale.base`	Base of logarithm used to transform data manually. The default, `NULL`, will use the scale transformation to calculate positions. Only set `prescale.base` if the data has already been log-transformed. When using a log-transform in the position scale or in `coord_transform()`, keep the default `NULL` argument.
`negative.small`	When the scale limits include 0 or negative numbers, what should be the smallest absolute value that is marked with a tick? If `NULL` (default), will be the smallest of 0.1 or 0.1 times the absolute scale maximum.

`short.theme`	A theme element for customising the display of the shortest ticks. Must be a line or blank element, and it inherits from the `axis.minor.ticks` setting for the relevant position.
`expanded`	Whether the ticks should cover the range after scale expansion (`TRUE`, default), or be restricted to the scale limits (`FALSE`).
`cap`	A `character` to cut the axis line back to the last breaks. Can be `"none"` (default) to draw the axis line along the whole panel, or `"upper"` and `"lower"` to draw the axis to the upper or lower break, or `"both"` to only draw the line in between the most extreme breaks. `TRUE` and `FALSE` are shorthand for `"both"` and `"none"` respectively.
`theme`	A `theme` object to style the guide individually or differently from the plot's theme settings. The `theme` argument in the guide partially overrides, and is combined with, the plot's theme.
`prescale_base`, `negative_small`, `short_theme`	**[Deprecated]**
`...`	Arguments passed on to `guide_axis`

`check.overlap` silently remove overlapping labels, (recursively) prioritizing the first, last, and middle labels.

`angle` Compared to setting the angle in `theme()` / `element_text()`, this also uses some heuristics to automatically pick the `hjust` and `vjust` that you probably want. Can be one of the following:

- `NULL` to take the angles and `hjust`/`vjust` directly from the theme.
- `waiver()` to allow reasonable defaults in special cases.
- A number representing the text angle in degrees.

`n.dodge` The number of rows (for vertical axes) or columns (for horizontal axes) that should be used to render the labels. This is useful for displaying labels that would otherwise overlap.

`order` A positive `integer` of length 1 that specifies the order of this guide among multiple guides. This controls in which order guides are merged if there are multiple guides for the same position. If 0 (default), the order is determined by a secret algorithm.

`position` Where this guide should be drawn: one of top, bottom, left, or right.

`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.

Examples

```
# A standard plot
p <- ggplot(msleep, aes(bodywt, brainwt)) +
  geom_point(na.rm = TRUE)

# The logticks axis works well with log scales
p + scale_x_log10(guide = "axis_logticks") +
  scale_y_log10(guide = "axis_logticks")

# Or with log-transformed coordinates
```

```
p + coord_transform(x = "log10", y = "log10") +
  guides(x = "axis_logticks", y = "axis_logticks")

# When data is transformed manually, one should provide `prescale.base`
# Keep in mind that this axis uses log10 space for placement, not log2
p + aes(x = log2(bodywt), y = log10(brainwt)) +
  guides(
    x = guide_axis_logticks(prescale.base = 2),
    y = guide_axis_logticks(prescale.base = 10)
  )

# A plot with both positive and negative extremes, pseudo-log transformed
set.seed(42)
p2 <- ggplot(data.frame(x = rcauchy(1000)), aes(x = x)) +
  geom_density() +
  scale_x_continuous(
    breaks = c(-10^(4:0), 0, 10^(0:4)),
    transform = "pseudo_log"
  )

# The log ticks are mirrored when 0 is included
p2 + guides(x = "axis_logticks")

# To control the tick density around 0, one can set `negative.small`
p2 + guides(x = guide_axis_logticks(negative.small = 1))
```

guide_axis_stack   *Stacked axis guides*

Description

This guide can stack other position guides that represent position scales, like those created with scale_(xly)_continuous() and scale_(xly)_discrete().

Usage

```
guide_axis_stack(
  first = "axis",
  ...,
  title = waiver(),
  theme = NULL,
  spacing = NULL,
  order = 0,
  position = waiver()
)
```

Arguments

first	A position guide given as one of the following:

- A string, for example "axis".
- A call to a guide function, for example guide_axis().

...	Additional guides to stack given in the same manner as first.
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.
theme	A theme object to style the guide individually or differently from the plot's theme settings. The theme argument in the guide partially overrides, and is combined with, the plot's theme.
spacing	A unit() objects that determines how far separate guides are spaced apart.
order	A positive integer of length 1 that specifies the order of this guide among multiple guides. This controls in which order guides are merged if there are multiple guides for the same position. If 0 (default), the order is determined by a secret algorithm.
position	Where this guide should be drawn: one of top, bottom, left, or right.

Details

The first guide will be placed closest to the panel and any subsequent guides provided through ... will follow in the given order.

Examples

```
# A standard plot
p <- ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  theme(axis.line = element_line())

# A normal axis first, then a capped axis
p + guides(x = guide_axis_stack("axis", guide_axis(cap = "both")))
```

---

guide_axis_theta *Angle axis guide*

Description

This is a specialised guide used in coord_radial() to represent the theta position scale.

Usage

```
guide_axis_theta(
  title = waiver(),
  theme = NULL,
  angle = waiver(),
  minor.ticks = FALSE,
  cap = "none",
  order = 0,
  position = waiver()
)
```

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.
`theme`	A `theme` object to style the guide individually or differently from the plot's theme settings. The `theme` argument in the guide partially overrides, and is combined with, the plot's theme.
`angle`	Compared to setting the angle in `theme()` / `element_text()`, this also uses some heuristics to automatically pick the `hjust` and `vjust` that you probably want. Can be one of the following: • `NULL` to take the angles and `hjust`/`vjust` directly from the theme. • `waiver()` to allow reasonable defaults in special cases. • A number representing the text angle in degrees.
`minor.ticks`	Whether to draw the minor ticks (`TRUE`) or not draw minor ticks (`FALSE`, default).
`cap`	A `character` to cut the axis line back to the last breaks. Can be `"none"` (default) to draw the axis line along the whole panel, or `"upper"` and `"lower"` to draw the axis to the upper or lower break, or `"both"` to only draw the line in between the most extreme breaks. `TRUE` and `FALSE` are shorthand for `"both"` and `"none"` respectively.
`order`	A positive `integer` of length 1 that specifies the order of this guide among multiple guides. This controls in which order guides are merged if there are multiple guides for the same position. If 0 (default), the order is determined by a secret algorithm.
`position`	Where this guide should be drawn: one of top, bottom, left, or right.

Note

The axis labels in this guide are insensitive to `hjust` and `vjust` settings. The distance from the tick marks to the labels is determined by the largest `margin` size set in the theme.

Examples

```
# A plot using coord_radial
p <- ggplot(mtcars, aes(disp, mpg)) +
  geom_point() +
  coord_radial()

# The `angle` argument can be used to set relative angles
p + guides(theta = guide_axis_theta(angle = 0))
```

guide_bins — *A binned version of guide_legend*

Description

This guide is a version of the `guide_legend()` guide for binned scales. It differs in that it places ticks correctly between the keys, and sports a small axis to better show the binning. Like `guide_legend()` it can be used for all non-position aesthetics though colour and fill defaults to `guide_coloursteps()`, and it will merge aesthetics together into the same guide if they are mapped in the same way.

Usage

```
guide_bins(
  title = waiver(),
  theme = NULL,
  angle = NULL,
  position = NULL,
  direction = NULL,
  override.aes = list(),
  reverse = FALSE,
  order = 0,
  show.limits = NULL,
  ...
)
```

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.
`theme`	A `theme` object to style the guide individually or differently from the plot's theme settings. The `theme` argument in the guide partially overrides, and is combined with, the plot's theme. Arguments that apply to a single legend are respected, most of which have the `legend`-prefix. Arguments that apply to combined legends (the legend box) are ignored, including `legend.position`, `legend.justification.*`, `legend.location` and `legend.box.*`.
`angle`	Overrules the theme settings to automatically apply appropriate `hjust` and `vjust` for angled legend text. Can be a single number representing the text angle in degrees, or `NULL` to not overrule the settings (default).
`position`	A character string indicating where the legend should be placed relative to the plot panels. One of "top", "right", "bottom", "left", or "inside".
`direction`	A character string indicating the direction of the guide. One of "horizontal" or "vertical".
`override.aes`	A list specifying aesthetic parameters of legend key. See details and examples.
`reverse`	logical. If `TRUE` the order of legends is reversed.

`order`	positive integer less than 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.
`show.limits`	Logical. Should the limits of the scale be shown with labels and ticks. Default is `NULL` meaning it will take the value from the scale. This argument is ignored if `labels` is given as a vector of values. If one or both of the limits is also given in `breaks` it will be shown irrespective of the value of `show.limits`.
`...`	ignored.

Value

A guide object

Use with discrete scale

This guide is intended to show binned data and work together with ggplot2's binning scales. However, it is sometimes desirable to perform the binning in a separate step, either as part of a stat (e.g. `stat_contour_filled()`) or prior to the visualisation. If you want to use this guide for discrete data the levels must follow the naming scheme implemented by `base::cut()`. This means that a bin must be encoded as `"(<lower>, <upper>]"` with `<lower>` giving the lower bound of the bin and `<upper>` giving the upper bound (`"[<lower>, <upper>)"` is also accepted). If you use `base::cut()` to perform the binning everything should work as expected, if not, some recoding may be needed.

See Also

Other guides: `guide_colourbar()`, `guide_coloursteps()`, `guide_legend()`, `guides()`

Examples

```
p <- ggplot(mtcars) +
  geom_point(aes(disp, mpg, size = hp)) +
  scale_size_binned()

# Standard look
p

# Remove the axis or style it
p + guides(size = guide_bins(
  theme = theme(legend.axis.line = element_blank())
))

p + guides(size = guide_bins(show.limits = TRUE))

my_arrow <- arrow(length = unit(1.5, "mm"), ends = "both")
p + guides(size = guide_bins(
  theme = theme(legend.axis.line = element_line(arrow = my_arrow))
))
```

```
# Guides are merged together if possible
ggplot(mtcars) +
  geom_point(aes(disp, mpg, size = hp, colour = hp)) +
  scale_size_binned() +
  scale_colour_binned(guide = "bins")
```

---

guide_colourbar *Continuous colour bar guide*

Description

Colour bar guide shows continuous colour scales mapped onto values. Colour bar is available with `scale_fill` and `scale_colour`.

Usage

```
guide_colourbar(
  title = waiver(),
  theme = NULL,
  nbin = NULL,
  display = "raster",
  raster = deprecated(),
  alpha = NA,
  draw.ulim = TRUE,
  draw.llim = TRUE,
  angle = NULL,
  position = NULL,
  direction = NULL,
  reverse = FALSE,
  order = 0,
  available_aes = c("colour", "color", "fill"),
  ...
)

guide_colorbar(
  title = waiver(),
  theme = NULL,
  nbin = NULL,
  display = "raster",
  raster = deprecated(),
  alpha = NA,
  draw.ulim = TRUE,
  draw.llim = TRUE,
  angle = NULL,
  position = NULL,
  direction = NULL,
  reverse = FALSE,
  order = 0,
  available_aes = c("colour", "color", "fill"),
  ...
)
```

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.
`theme`	A `theme` object to style the guide individually or differently from the plot's theme settings. The `theme` argument in the guide partially overrides, and is combined with, the plot's theme. Arguments that apply to a single legend are respected, most of which have the `legend`-prefix. Arguments that apply to combined legends (the legend box) are ignored, including `legend.position`, `legend.justification.*`, `legend.location` and `legend.box.*`.
`nbin`	A numeric specifying the number of bins for drawing the colourbar. A smoother colourbar results from a larger value.
`display`	A string indicating a method to display the colourbar. Can be one of the following: • `"raster"` to display as a bitmap image. • `"rectangles"` to display as a series of rectangles. • `"gradient"` to display as a linear gradient. Note that not all devices are able to render rasters and gradients.
`raster`	**[Deprecated]** A logical. If `TRUE` then the colourbar is rendered as a raster object. If `FALSE` then the colourbar is rendered as a set of rectangles. Note that not all graphics devices are capable of rendering raster image.
`alpha`	A numeric between 0 and 1 setting the colour transparency of the bar. Use `NA` to preserve the alpha encoded in the colour itself (default).
`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.
`angle`	Overrules the theme settings to automatically apply appropriate `hjust` and `vjust` for angled legend text. Can be a single number representing the text angle in degrees, or `NULL` to not overrule the settings (default).
`position`	A character string indicating where the legend should be placed relative to the plot panels. One of "top", "right", "bottom", "left", or "inside".
`direction`	A character string indicating the direction of the guide. One of "horizontal" or "vertical."
`reverse`	logical. If `TRUE` the colourbar is reversed. By default, the highest value is on the top and the lowest value is on the bottom
`order`	positive integer less than 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.

`available_aes`	A vector of character strings listing the aesthetics for which a colourbar can be drawn.
`...`	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_colour_manual(guide = "legend")`). As for how to specify the guide for each scale in more detail, see `guides()`.

Value

A guide object

See Also

The continuous legend section of the online ggplot2 book.

Other guides: `guide_bins()`, `guide_coloursteps()`, `guide_legend()`, `guides()`

Examples

```
df <- expand.grid(X1 = 1:10, X2 = 1:10)
df$value <- df$X1 * df$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 = "colourbar")
p1 + scale_fill_continuous(guide = guide_colourbar())
p1 + guides(fill = guide_colourbar())

# Control styles

# bar size
p1 + guides(fill = guide_colourbar(theme = theme(
  legend.key.width  = unit(0.5, "lines"),
  legend.key.height = unit(10, "lines")
)))


# no label
p1 + guides(fill = guide_colourbar(theme = theme(
  legend.text = element_blank()
)))

# no tick marks
p1 + guides(fill = guide_colourbar(theme = theme(
  legend.ticks = element_blank()
)))
```

```
# label position
p1 + guides(fill = guide_colourbar(theme = theme(
  legend.text.position = "left"
)))

# label theme
p1 + guides(fill = guide_colourbar(theme = theme(
  legend.text = element_text(colour = "blue", angle = 0)
)))

# small number of bins
p1 + guides(fill = guide_colourbar(nbin = 3))

# large number of bins
p1 + guides(fill = guide_colourbar(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_colourbar(nbin = 100, draw.ulim = FALSE, draw.llim = FALSE)
  )

# guides can be controlled independently
p2 +
  scale_fill_continuous(guide = "colourbar") +
  scale_size(guide = "legend")
p2 + guides(fill = "colourbar", size = "legend")

p2 +
  scale_fill_continuous(guide = guide_colourbar(theme = theme(
    legend.direction = "horizontal"
  ))) +
  scale_size(guide = guide_legend(theme = theme(
    legend.direction = "vertical"
  )))
```

guide_coloursteps *Discretized colourbar guide*

Description

This guide is version of guide_colourbar() for binned colour and fill scales. It shows areas between breaks as a single constant colour instead of the gradient known from the colourbar counterpart.

Usage

```
guide_coloursteps(
  title = waiver(),
  theme = NULL,
  alpha = NA,
  angle = NULL,
  even.steps = TRUE,
  show.limits = NULL,
  direction = NULL,
  position = NULL,
  reverse = FALSE,
  order = 0,
  available_aes = c("colour", "color", "fill"),
  ...
)

guide_colorsteps(
  title = waiver(),
  theme = NULL,
  alpha = NA,
  angle = NULL,
  even.steps = TRUE,
  show.limits = NULL,
  direction = NULL,
  position = NULL,
  reverse = FALSE,
  order = 0,
  available_aes = c("colour", "color", "fill"),
  ...
)
```

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.
`theme`	A `theme` object to style the guide individually or differently from the plot's theme settings. The `theme` argument in the guide partially overrides, and is combined with, the plot's theme. Arguments that apply to a single legend are respected, most of which have the `legend`-prefix. Arguments that apply to combined legends (the legend box) are ignored, including `legend.position`, `legend.justification.*`, `legend.location` and `legend.box.*`.
`alpha`	A numeric between 0 and 1 setting the colour transparency of the bar. Use `NA` to preserve the alpha encoded in the colour itself (default).
`angle`	Overrules the theme settings to automatically apply appropriate `hjust` and `vjust` for angled legend text. Can be a single number representing the text angle in degrees, or `NULL` to not overrule the settings (default).
`even.steps`	Should the rendered size of the bins be equal, or should they be proportional to their length in the data space? Defaults to `TRUE`
`show.limits`	Logical. Should the limits of the scale be shown with labels and ticks. Default is `NULL` meaning it will take the value from the scale. This argument is ignored

	if `labels` is given as a vector of values. If one or both of the limits is also given in `breaks` it will be shown irrespective of the value of `show.limits`.
`direction`	A character string indicating the direction of the guide. One of "horizontal" or "vertical."
`position`	A character string indicating where the legend should be placed relative to the plot panels. One of "top", "right", "bottom", "left", or "inside".
`reverse`	logical. If `TRUE` the colourbar is reversed. By default, the highest value is on the top and the lowest value is on the bottom
`order`	positive integer less than 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.
`available_aes`	A vector of character strings listing the aesthetics for which a colourbar can be drawn.
`...`	ignored.

Value

A guide object

Use with discrete scale

This guide is intended to show binned data and work together with ggplot2's binning scales. However, it is sometimes desirable to perform the binning in a separate step, either as part of a stat (e.g. `stat_contour_filled()`) or prior to the visualisation. If you want to use this guide for discrete data the levels must follow the naming scheme implemented by `base::cut()`. This means that a bin must be encoded as `"(<lower>, <upper>]"` with `<lower>` giving the lower bound of the bin and `<upper>` giving the upper bound (`"[<lower>, <upper>)"` is also accepted). If you use `base::cut()` to perform the binning everything should work as expected, if not, some recoding may be needed.

See Also

The binned legend section of the online ggplot2 book.

Other guides: `guide_bins()`, `guide_colourbar()`, `guide_legend()`, `guides()`

Examples

```
df <- expand.grid(X1 = 1:10, X2 = 1:10)
df$value <- df$X1 * df$X2

p <- ggplot(df, aes(X1, X2)) + geom_tile(aes(fill = value))

# Coloursteps guide is the default for binned colour scales
p + scale_fill_binned()

# By default each bin in the guide is the same size irrespectively of how
# their sizes relate in data space
```

```
p + scale_fill_binned(breaks = c(10, 25, 50))

# This can be changed with the `even.steps` argument
p + scale_fill_binned(
  breaks = c(10, 25, 50),
  guide = guide_coloursteps(even.steps = FALSE)
)

# By default the limits is not shown, but this can be changed
p + scale_fill_binned(guide = guide_coloursteps(show.limits = TRUE))

# (can also be set in the scale)
p + scale_fill_binned(show.limits = TRUE)
```

---

guide_custom *Custom guides*

Description

This is a special guide that can be used to display any graphical object (grob) along with the regular guides. This guide has no associated scale.

Usage

```
guide_custom(
  grob,
  width = grobWidth(grob),
  height = grobHeight(grob),
  title = NULL,
  theme = NULL,
  position = NULL,
  order = 0
)
```

Arguments

`grob`	A grob to display.
`width, height`	The allocated width and height to display the grob, given in `grid::unit()`s.
`title`	A character string or expression indicating the title of guide. If `NULL` (default), no title is shown.
`theme`	A `theme` object to style the guide individually or differently from the plot's theme settings. The `theme` argument in the guide partially overrides, and is combined with, the plot's theme. Arguments that apply to a single legend are respected, most of which have the `legend`-prefix. Arguments that apply to combined legends (the legend box) are ignored, including `legend.position`, `legend.justification.*`, `legend.location` and `legend.box.*`.

position	A character string indicating where the legend should be placed relative to the plot panels. One of "top", "right", "bottom", "left", or "inside".
order	positive integer less than 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.

Examples

```
# A standard plot
p <- ggplot(mpg, aes(displ, hwy)) +
  geom_point()

# Define a graphical object
circle <- grid::circleGrob()

# Rendering a grob as a guide
p + guides(custom = guide_custom(circle, title = "My circle"))

# Controlling the size of the grob defined in relative units
p + guides(custom = guide_custom(
  circle, title = "My circle",
  width = unit(2, "cm"), height = unit(2, "cm"))
)

# Size of grobs in absolute units is taken directly without the need to
# set these manually
p + guides(custom = guide_custom(
  title = "My circle",
  grob = grid::circleGrob(r = unit(1, "cm"))
))
```

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

```
guide_legend(
  title = waiver(),
  theme = NULL,
  position = NULL,
  direction = NULL,
  override.aes = list(),
  nrow = NULL,
  ncol = NULL,
  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.
`theme`	A `theme` object to style the guide individually or differently from the plot's theme settings. The `theme` argument in the guide partially overrides, and is combined with, the plot's theme. Arguments that apply to a single legend are respected, most of which have the `legend`-prefix. Arguments that apply to combined legends (the legend box) are ignored, including `legend.position`, `legend.justification.*`, `legend.location` and `legend.box.*`.
`position`	A character string indicating where the legend should be placed relative to the plot panels. One of "top", "right", "bottom", "left", or "inside".
`direction`	A character string indicating the direction of the guide. One of "horizontal" or "vertical".
`override.aes`	A list specifying aesthetic parameters of legend key. See details and examples.
`nrow, ncol`	The desired number of rows and column of legends respectively.
`reverse`	logical. If `TRUE` the order of legends is reversed.
`order`	positive integer less than 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()`.

See Also

The legends section of the online ggplot2 book.

Other guides: `guide_bins()`, `guide_colourbar()`, `guide_coloursteps()`, `guides()`

Examples

```
df <- expand.grid(X1 = 1:10, X2 = 1:10)
df$value <- df$X1 * df$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 = guide_legend())

# Control styles

# title position
p1 + guides(fill = guide_legend(
  title = "LEFT", theme(legend.title.position = "left")
))

# title text styles via element_text
p1 + guides(fill = guide_legend(theme = theme(
  legend.title = element_text(size = 15, face = "italic", colour = "red")
)))

# label position
p1 + guides(fill = guide_legend(theme = theme(
  legend.text.position = "left",
  legend.text = element_text(hjust = 1)
)))

# label styles
p1 +
  scale_fill_continuous(
    breaks = c(5, 10, 15),
    labels = paste("long", c(5, 10, 15)),
    guide = guide_legend(theme = theme(
      legend.direction = "horizontal",
      legend.title.position = "top",
      legend.text.position = "bottom",
      legend.text = element_text(hjust = 0.5, vjust = 1, angle = 90)
    ))
  )

# Set aesthetic of legend key
# very low alpha value make it difficult to see legend key
p3 <- ggplot(mtcars, aes(vs, am, colour = factor(cyl))) +
  geom_jitter(alpha = 1/5, width = 0.01, height = 0.01)
p3
# override.aes overwrites the alpha
p3 + guides(colour = guide_legend(override.aes = list(alpha = 1)))

# multiple row/col legends
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, theme = theme(legend.byrow = TRUE)))

# reversed order legend
```

```
p + guides(col = guide_legend(reverse = TRUE))
```

guide_none *Empty guide*

Description

This guide draws nothing.

Usage

```
guide_none(title = waiver(), position = waiver())
```

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.
`position`	Where this guide should be drawn: one of top, bottom, left, or right.

hmisc *A selection of summary functions from Hmisc*

Description

These are wrappers around functions from **Hmisc** designed to make them easier to use with `stat_summary()`. See the Hmisc documentation for more details:

- `Hmisc::smean.cl.boot()`
- `Hmisc::smean.cl.normal()`
- `Hmisc::smean.sdl()`
- `Hmisc::smedian.hilow()`

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.

Value

A data frame with columns y, ymin, and ymax.

Examples

```
if (requireNamespace("Hmisc", quietly = TRUE)) {
set.seed(1)
x <- rnorm(100)
mean_cl_boot(x)
mean_cl_normal(x)
mean_sdl(x)
median_hilow(x)
}
```

---

labeller *Construct labelling specification*

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 vector that will serve as a lookup table.

Usage

```
labeller(
  ...,
  .rows = NULL,
  .cols = NULL,
  keep.as.numeric = deprecated(),
  .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 `Hmisc::capitalize()`.

Value

A labeller function to supply to `facet_grid()` or `facet_wrap()` for the argument `labeller`.

See Also

`as_labeller()`, labellers

Examples

```
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()
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"
)
## Source: http://en.wikipedia.org/wiki/Wikipedia:Conservation_status

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
idx <- match(msleep$conservation, names(conservation_status))
msleep$conservation2 <- conservation_status[idx]

p3 <- ggplot(msleep, aes(x = sleep_total, y = awake)) + geom_point()
p3 +
  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)
p3 + facet_wrap(~conservation2, labeller = global_labeller)
```

labellers *Useful 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 faceting 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 faceting and `label_both()` when multiple factors are involved. `label_wrap_gen()` uses `base::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

```
mtcars$cyl2 <- factor(mtcars$cyl, labels = c("alpha", "beta", "gamma"))
p <- ggplot(mtcars, aes(wt, mpg)) + geom_point()

# The default is label_value
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)

# Include optional argument in label function
p + facet_grid(. ~ cyl, labeller = \(x) label_both(x, sep = "="))
```

---

label_bquote *Label with mathematical expressions*

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

```
label_bquote(rows = NULL, cols = NULL, default)
```

Arguments

rows	Backquoted labelling expression for rows.
cols	Backquoted labelling expression for columns.
default	Unused, kept for compatibility.

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(wt, 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)))
```

`labs` *Modify axis, legend, and plot labels*

Description

Good labels are critical for making your plots accessible to a wider audience. Always ensure the axis and legend labels display the full variable name. Use the plot `title` and `subtitle` to explain the main findings. It's common to use the `caption` to provide information about the data source. `tag` can be used for adding identification tags to differentiate between multiple plots.

[Superseded]: `xlab()`, `ylab()` and `ggtitle()` are superseded. It is recommended to use the `labs(x, y, title, subtitle)` arguments instead.

`get_labs()` retrieves completed labels from a plot.

Usage

```
labs(
  ...,
  title = waiver(),
  subtitle = waiver(),
  caption = waiver(),
  tag = waiver(),
  dictionary = waiver(),
  alt = waiver(),
  alt_insight = waiver()
)

xlab(label)

ylab(label)

ggtitle(label, subtitle = waiver())

get_labs(plot = get_last_plot())
```

Arguments

`...`	A list of new name-value pairs. The name should be an aesthetic.
`title`	The text for the title.
`subtitle`	The text for the subtitle for the plot which will be displayed below the title.
`caption`	The text for the caption which will be displayed in the bottom-right of the plot by default.
`tag`	The text for the tag label which will be displayed at the top-left of the plot by default.
`dictionary`	A named character vector to serve as dictionary. Automatically derived labels, such as those based on variables will be matched with `names(dictionary)` and replaced by the matching entry in `dictionary`.
`alt, alt_insight`	Text used for the generation of alt-text for the plot. See get_alt_text for examples. `alt` can also be a function that takes the plot as input and returns text as output. `alt` also accepts rlang lambda function notation.
`label`	The title of the respective axis (for `xlab()` or `ylab()`) or of the plot (for `ggtitle()`).
`plot`	A ggplot object

Details

You can also set axis and legend labels in the individual scales (using the first argument, the `name`). If you're changing other scale options, this is recommended.

If a plot already has a title, subtitle, caption, etc., and you want to remove it, you can do so by setting the respective argument to `NULL`. For example, if plot `p` has a subtitle, then `p + labs(subtitle = NULL)` will remove the subtitle from the plot.

See Also

The plot and axis titles section of the online ggplot2 book.

Examples

```
p <- ggplot(mtcars, aes(mpg, wt, colour = cyl)) + geom_point()
p + labs(colour = "Cylinders")
p + labs(x = "New x label")

# Set labels by variable name instead of aesthetic
p + labs(dict = c(
  disp = "Displacment", # Not in use
  cyl  = "Number of cylinders",
  mpg  = "Miles per gallon",
  wt   = "Weight (1000 lbs)"
))

# The plot title appears at the top-left, with the subtitle
# display in smaller text underneath it
p + labs(title = "New plot title")
```

```
p + labs(title = "New plot title", subtitle = "A subtitle")

# The caption appears in the bottom-right, and is often used for
# sources, notes or copyright
p + labs(caption = "(based on data from ...)")

# The plot tag appears at the top-left, and is typically used
# for labelling a subplot with a letter.
p + labs(title = "title", tag = "A")

# If you want to remove a label, set it to NULL.
p +
 labs(title = "title") +
 labs(title = NULL)
```

`layer_geoms` *Layer geometry display*

Description

In ggplot2, a plot in constructed by adding layers to it. A layer consists of two important parts: the geometry (geoms), and statistical transformations (stats). The 'geom' part of a layer is important because it determines the looks of the data. Geoms determine *how* something is displayed, not *what* is displayed.

Specifying geoms

There are five ways in which the 'geom' part of a layer can be specified.

```
# 1. The geom can have a layer constructor
geom_area()

# 2. A stat can default to a particular geom
stat_density() # has `geom = "area"` as default

# 3. It can be given to a stat as a string
stat_function(geom = "area")

# 4. The ggproto object of a geom can be given
stat_bin(geom = GeomArea)

# 5. It can be given to `layer()` directly
layer(
  geom = "area",
  stat = "smooth",
  position = "identity"
)
```

Many of these ways are absolutely equivalent. Using `stat_density(geom = "line")` is identical to using `geom_line(stat = "density")`. Note that for `layer()`, you need to provide the `"position"` argument as well. To give geoms as a string, take the function name, and remove the `geom_` prefix, such that `geom_point` becomes `"point"`.

Some of the more well known geoms that can be used for the `geom` argument are: `"point"`, `"line"`, `"area"`, `"bar"` and `"polygon"`.

Graphical display

A ggplot is build on top of the grid package. This package understands various graphical primitives, such as points, lines, rectangles and polygons and their positions, as well as graphical attributes, also termed aesthetics, such as colours, fills, linewidths and linetypes. The job of the geom part of a layer, is to translate data to grid graphics that can be plotted.

To see how aesthetics are specified, run `vignette("ggplot2-specs")`. To see what geom uses what aesthetics, you can find the **Aesthetics** section in their documentation, for example in `?geom_line`.

While almost anything can be represented by polygons if you try hard enough, it is not always convenient to do so manually. For this reason, the geoms provide abstractions that take most of this hassle away. `geom_ribbon()` for example is a special case of `geom_polygon()`, where two sets of y-positions have a shared x-position. In turn, `geom_area()` is a special case of a ribbon, where one of the two sets of y-positions is set at 0.

```
# A hassle to build a polygon
my_polygon <- data.frame(
  x = c(economics$date,    rev(economics$date)),
  y = c(economics$uempmed, rev(economics$psavert))
)
ggplot(my_polygon, aes(x, y)) +
  geom_polygon()

# More succinctly
ggplot(economics, aes(date)) +
  geom_ribbon(aes(ymin = uempmed, ymax = psavert))
```

In addition to abstraction, geoms sometimes also perform composition. A boxplot is a particular arrangement of lines, rectangles and points that people have agreed upon is a summary of some data, which is performed by `geom_boxplot()`.

```
Boxplot data
value <- fivenum(rnorm(100))
df <- data.frame(
  min = value[1], lower = value[2], middle = value[3],
  upper = value[4], max = value[5]
)

# Drawing a boxplot manually
ggplot(df, aes(x = 1, xend = 1)) +
  geom_rect(
    aes(
      xmin = 0.55, xmax = 1.45,
      ymin = lower, ymax = upper
    ),
    colour = "black", fill = "white"
  ) +
  geom_segment(
    aes(
      x = 0.55, xend = 1.45,
      y = middle, yend = middle
    ),
    size = 1
  ) +
  geom_segment(aes(y = lower, yend = min)) +
  geom_segment(aes(y = upper, yend = max))

# More succinctly
ggplot(df, aes(x = 1)) +
  geom_boxplot(
    aes(ymin = min, ymax = max,
        lower = lower, upper = upper,
        middle = middle),
    stat = "identity"
  )
```

Under the hood

Internally, geoms are represented as `ggproto` classes that occupy a slot in a layer. All these classes inherit from the parental `Geom` ggproto object that orchestrates how geoms work. Briefly, geoms are given the opportunity to draw the data of the layer as a whole, a facet panel, or of individual groups. For more information on extending geoms, see the **Creating a new geom** section after running `vignette("extending-ggplot2")`. Additionally, see the **New geoms** section of the online book.

See Also

For an overview of all geom layers, see the online reference.

Other layer documentation: `layer()`, `layer_positions`, `layer_stats`

`layer_positions` *Layer position adjustments*

Description

In ggplot2, a plot is constructed by adding layers to it. In addition to geoms and stats, position adjustments are the third required part of a layer. The 'position' part of a layer is responsible for dodging, jittering and nudging groups of data to minimise their overlap, or otherwise tweaking their positions.

For example if you add `position = position_nudge(x = 1)` to a layer, you can offset every x-position by 1. For many layers, the default position adjustment is `position_identity()`, which performs no adjustment.

Specifying positions

There are 4 ways in which the 'position' part of a layer can be specified.

```
1. A layer can have default position adjustments
geom_jitter() # has `position = "jitter"`

2. It can be given to a layer as a string
geom_point(position = "jitter")

3. The position function can be used to pass extra arguments
geom_point(position = position_jitter(width = 1))

4. It can be given to `layer()` directly
layer(
  geom = "point",
  stat = "identity",
  position = "jitter"
)
```

These ways are not always equivalent. Some layers may not understand what to do with a position adjustment, and require additional parameters passed through the `position_*()` function, or may not work correctly. For example `position_dodge()` requires non-overlapping x intervals, whereas `geom_point()` doesn't have dimensions to calculate intervals for. To give positions as a string, take the function name, and remove the `position_` prefix, such that `position_fill` becomes `"fill"`.

Pairing geoms with positions

Some geoms work better with some positions than others. Below follows a brief overview of geoms and position adjustments that work well together.

Identity:
`position_identity()` can work with virtually any geom.

Dodging:
`position_dodge()` pushes overlapping objects away from one another and requires a `group` variable. `position_dodge2()` can work without group variables and can handle variable widths. As a rule of thumb, layers where groups occupy a range on the x-axis pair well with dodging. If layers have no width, you may be required to specify it manually with `position_dodge(width = ...)`. Some geoms that pair well with dodging are `geom_bar()`, `geom_boxplot()`, `geom_linerange()`, `geom_errorbar()` and `geom_text()`.

Jittering:
`position_jitter()` adds a some random noise to every point, which can help with overplotting. `position_jitterdodge()` does the same, but also dodges the points. As a rule of thumb, jittering

works best when points have discrete x-positions. Jittering is most useful for `geom_point()`, but can also be used in `geom_path()` for example.

Nudging:

`position_nudge()` can add offsets to x- and y-positions. This can be useful for discrete positions where you don't want to put an object exactly in the middle. While most useful for `geom_text()`, it can be used with virtually all geoms.

Stacking:

`position_stack()` is useful for displaying data on top of one another. It can be used for geoms that are usually anchored to the x-axis, for example `geom_bar()`, `geom_area()` or `geom_histogram()`.

Filling:

`position_fill()` can be used to give proportions at every x-position. Like stacking, filling is most useful for geoms that are anchored to the x-axis, like `geom_bar()`, `geom_area()` or `geom_histogram()`.

Under the hood

Internally, positions are represented as `ggproto` classes that occupy a slot in a layer. All these classes inherit from the parental `Position` ggproto object that orchestrates how positions work. Briefly, positions are given the opportunity to adjust the data of each facet panel. For more information about extending positions, see the **New positions** section of the online book.

See Also

For an overview of all position adjustments, see the online reference.

Other layer documentation: `layer()`, `layer_geoms`, `layer_stats`

`layer_stats` *Layer statistical transformations*

Description

In ggplot2, a plot is constructed by adding layers to it. A layer consists of two important parts: the geometry (geoms), and statistical transformations (stats). The 'stat' part of a layer is important because it performs a computation on the data before it is displayed. Stats determine *what* is displayed, not *how* it is displayed.

For example, if you add `stat_density()` to a plot, a kernel density estimation is performed, which can be displayed with the 'geom' part of a layer. For many `geom_*()` functions, `stat_identity()` is used, which performs no extra computation on the data.

Specifying stats

There are five ways in which the 'stat' part of a layer can be specified.

```
# 1. The stat can have a layer constructor
stat_density()

# 2. A geom can default to a particular stat
geom_density() # has `stat = "density"` as default

# 3. It can be given to a geom as a string
geom_line(stat = "density")

# 4. The ggproto object of a stat can be given
geom_area(stat = StatDensity)

# 5. It can be given to `layer()` directly:
layer(
  geom = "line",
  stat = "density",
  position = "identity"
)
```

Many of these ways are absolutely equivalent. Using `stat_density(geom = "line")` is identical to using `geom_line(stat = "density")`. Note that for `layer()`, you need to provide the `"position"` argument as well. To give stats as a string, take the function name, and remove the `stat_` prefix, such that `stat_bin` becomes `"bin"`.

Some of the more well known stats that can be used for the `stat` argument are: `"density"`, `"bin"`, `"count"`, `"function"` and `"smooth"`.

Paired geoms and stats

Some geoms have paired stats. In some cases, like `geom_density()`, it is just a variant of another geom, `geom_area()`, with slightly different defaults.

In other cases, the relationship is more complex. In the case of boxplots for example, the stat and the geom have distinct roles. The role of the stat is to compute the five-number summary of the data. In addition to just displaying the box of the five-number summary, the geom also provides display options for the outliers and widths of boxplots. In such cases, you cannot freely exchange geoms and stats: using `stat_boxplot(geom = "line")` or `geom_area(stat = "boxplot")` give errors.

Some stats and geoms that are paired are:

- `geom_violin()` and `stat_ydensity()`
- `geom_histogram()` and `stat_bin()`
- `geom_contour()` and `stat_contour()`
- `geom_function()` and `stat_function()`
- `geom_bin_2d()` and `stat_bin_2d()`
- `geom_boxplot()` and `stat_boxplot()`

- geom_count() and stat_sum()
- geom_density() and stat_density()
- geom_density_2d() and stat_density_2d()
- geom_hex() and stat_binhex()
- geom_quantile() and stat_quantile()
- geom_smooth() and stat_smooth()

Using computed variables

As mentioned above, the role of stats is to perform computation on the data. As a result, stats have 'computed variables' that determine compatibility with geoms. These computed variables are documented in the **Computed variables** sections of the documentation, for example in ?stat_bin. While more thoroughly documented in after_stat(), it should briefly be mentioned that these computed stats can be accessed in aes().

For example, the ?stat_density documentation states that, in addition to a variable called density, the stat computes a variable named count. Instead of scaling such that the area integrates to 1, the count variable scales the computed density such that the values can be interpreted as counts. If stat_density(aes(y = after_stat(count))) is used, we can display these count-scaled densities instead of the regular densities.

The computed variables offer flexibility in that arbitrary geom-stat pairings can be made. While not necessarily recommended, geom_line() *can* be paired with stat = "boxplot" if the line is instructed on how to use the boxplot computed variables:

```
ggplot(mpg, aes(factor(cyl))) +
  geom_line(
    # Stage gives 'displ' to the stat, and afterwards chooses 'middle' as
    # the y-variable to display
    aes(y = stage(displ, after_stat = middle),
        # Regroup after computing the stats to display a single line
        group = after_stat(1)),
    stat = "boxplot"
  )
```

Under the hood

Internally, stats are represented as ggproto classes that occupy a slot in a layer. All these classes inherit from the parental Stat ggproto object that orchestrates how stats work. Briefly, stats are given the opportunity to perform computation either on the layer as a whole, a facet panel, or on individual groups. For more information on extending stats, see the **Creating a new stat** section after running vignette("extending-ggplot2"). Additionally, see the **New stats** section of the online book.

See Also

For an overview of all stat layers, see the online reference.

How computed aesthetics work.

Other layer documentation: layer(), layer_geoms, layer_positions

lims — *Set scale limits*

Description

This is a shortcut for supplying the `limits` argument to the individual scales. By default, any values outside the limits specified are replaced with `NA`. Be warned that this will remove data outside the limits and this can produce unintended results. For changing x or y axis limits **without** dropping data observations, see `coord_cartesian(xlim, ylim)`, or use a full scale with `oob = scales::oob_keep`.

Usage

```
lims(...)

xlim(...)

ylim(...)
```

Arguments

`...`	For `xlim()` and `ylim()`: Two numeric values, specifying the left/lower limit and the right/upper limit of the scale. If the larger value is given first, the scale will be reversed. You can leave one value as `NA` if you want to compute the corresponding limit from the range of the data. For `lims()`: A name–value pair. The name must be an aesthetic, and the value must be either a length-2 numeric, a character, a factor, or a date/time. A numeric value will create a continuous scale. If the larger value comes first, the scale will be reversed. You can leave one value as `NA` if you want to compute the corresponding limit from the range of the data. A character or factor value will create a discrete scale. A date-time value will create a continuous date/time scale.

See Also

To expand the range of a plot to always include certain values, see `expand_limits()`. For other types of data, see `scale_x_discrete()`, `scale_x_continuous()`, `scale_x_date()`.

Examples

```
# Zoom into a specified area
ggplot(mtcars, aes(mpg, wt)) +
  geom_point() +
  xlim(15, 20)

# reverse scale
ggplot(mtcars, aes(mpg, wt)) +
  geom_point() +
  xlim(20, 15)

# with automatic lower limit
ggplot(mtcars, aes(mpg, wt)) +
  geom_point() +
  xlim(NA, 20)

# You can also supply limits that are larger than the data.
# This is useful if you want to match scales across different plots
small <- subset(mtcars, cyl == 4)
big <- subset(mtcars, cyl > 4)

ggplot(small, aes(mpg, wt, colour = factor(cyl))) +
  geom_point() +
  lims(colour = c("4", "6", "8"))

ggplot(big, aes(mpg, wt, colour = factor(cyl))) +
  geom_point() +
  lims(colour = c("4", "6", "8"))

# There are two ways of setting the axis limits: with limits or
# with coordinate systems. They work in two rather different ways.

set.seed(1)
last_month <- Sys.Date() - 0:59
df <- data.frame(
  date = last_month,
  price = c(rnorm(30, mean = 15), runif(30) + 0.2 * (1:30))
)

p <- ggplot(df, aes(date, price)) +
  geom_line() +
  stat_smooth()

p

# Setting the limits with the scale discards all data outside the range.
p + lims(x= c(Sys.Date() - 30, NA), y = c(10, 20))

# For changing x or y axis limits **without** dropping data
# observations use [coord_cartesian()]. Setting the limits on the
# coordinate system performs a visual zoom.
p + coord_cartesian(xlim =c(Sys.Date() - 30, NA), ylim = c(10, 20))
```

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,v Position in Luv colour space

col Colour name

`margin` *Theme elements*

Description

In conjunction with the theme system, the `element_` functions specify the display of how non-data components of the plot are drawn.

- `element_blank()`: draws nothing, and assigns no space.
- `element_rect()`: borders and backgrounds.
- `element_line()`: lines.
- `element_text()`: text.
- `element_polygon()`: polygons.
- `element_point()`: points.
- `element_geom()`: defaults for drawing layers.

`rel()` is used to specify sizes relative to the parent, `margin()`, `margin_part()` and `margin_auto()` are all used to specify the margins of elements.

Usage

```
margin(t = 0, r = 0, b = 0, l = 0, unit = "pt", ...)

margin_part(t = NA, r = NA, b = NA, l = NA, unit = "pt")

margin_auto(t = 0, r = t, b = t, l = r, unit = "pt")

element()

element_blank()

element_rect(
  fill = NULL,
  colour = NULL,
  linewidth = NULL,
  linetype = NULL,
  color = NULL,
  linejoin = NULL,
  inherit.blank = FALSE,
  size = deprecated(),
  ...
)

element_line(
  colour = NULL,
  linewidth = NULL,
  linetype = NULL,
  lineend = NULL,
  color = NULL,
  linejoin = NULL,
  arrow = NULL,
  arrow.fill = NULL,
  inherit.blank = FALSE,
  size = deprecated(),
  ...
)

element_text(
  family = NULL,
  face = NULL,
  colour = NULL,
  size = NULL,
  hjust = NULL,
  vjust = NULL,
  angle = NULL,
  lineheight = NULL,
  color = NULL,
  margin = NULL,
  debug = NULL,
  inherit.blank = FALSE,
  ...
)

element_polygon(
  fill = NULL,
  colour = NULL,
  linewidth = NULL,
  linetype = NULL,
  color = NULL,
  linejoin = NULL,
  inherit.blank = FALSE,
  ...
)
```

```
element_point(
  colour = NULL,
  shape = NULL,
  size = NULL,
  fill = NULL,
  stroke = NULL,
  color = NULL,
  inherit.blank = FALSE,
  ...
)

element_geom(
  ink = NULL,
  paper = NULL,
  accent = NULL,
  linewidth = NULL,
  borderwidth = NULL,
  linetype = NULL,
  bordertype = NULL,
  family = NULL,
  fontsize = NULL,
  pointsize = NULL,
  pointshape = NULL,
  colour = NULL,
  color = NULL,
  fill = NULL,
  ...
)

rel(x)
```

Arguments

`t, r, b, 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.
`...`	Reserved for future expansion.
`fill`	Fill colour. `fill_alpha()` can be used to set the transparency of the fill.
`colour, color`	Line/border colour. Color is an alias for colour. `alpha()` can be used to set the transparency of the colour.
`linewidth, borderwidth, stroke`	Line/border size in mm.
`linetype, bordertype`	Line type for lines and borders respectively. An integer (0:8), a name (blank, solid, dashed, dotted, dotdash, longdash, twodash), or a string with an even number (up to eight) of hexadecimal digits which give the lengths in consecutive positions in the string.

`linejoin`	Line join style, one of `"round"`, `"mitre"` or `"bevel"`.
`inherit.blank`	Should this element inherit the existence of an `element_blank` among its parents? If `TRUE` the existence of a blank element among its parents will cause this element to be blank as well. If `FALSE` any blank parent element will be ignored when calculating final element state.
`size, fontsize, pointsize`	text size in pts, point size in mm.
`lineend`	Line end style, one of `"round"`, `"butt"` or `"square"`.
`arrow`	Arrow specification, as created by `grid::arrow()`
`arrow.fill`	Fill colour for arrows.
`family`	The typeface to use. The validity of this value will depend on the graphics device being used for rendering the plot. See the systemfonts vignette for guidance on the best way to access fonts installed on your computer. The values `"sans"`, `"serif"`, and `"mono"` should always be valid and will select the default typeface for the respective styles. However, what is considered default is dependant on the graphics device and the operating system.
`face`	Font face ("plain", "italic", "bold", "bold.italic")
`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.
`shape, pointshape`	Shape for points (1-25).
`ink`	Foreground colour.
`paper`	Background colour.
`accent`	Accent colour.
`x`	A single number specifying size relative to parent element.

Details

The `element_polygon()` and `element_point()` functions are not rendered in standard plots and just serve as extension points.

Value

An object of class `element`, `rel`, or `margin`.

Examples

```
# A standard plot
plot <- ggplot(mpg, aes(displ, hwy)) + geom_point()

# Turning off theme elements by setting them to blank
plot + theme(
  panel.background = element_blank(),
  axis.text = element_blank()
)

# Text adjustments
plot + theme(
  axis.text = element_text(colour = "red", size = rel(1.5))
)

# Turning on the axis line with an arrow
plot + theme(
  axis.line = element_line(arrow = arrow())
)

plot + theme(
  panel.background = element_rect(fill = "white"),
  plot.margin = margin_auto(2, unit = "cm"),
  plot.background = element_rect(
    fill = "grey90",
    colour = "black",
    linewidth = 1
  )
)

ggplot(mpg, aes(displ, hwy)) +
  geom_point() +
  geom_smooth(formula = y ~ x, method = "lm") +
  theme(geom = element_geom(
    ink = "red", accent = "black",
    pointsize = 1, linewidth = 2
  ))
```

`mean_se` *Calculate mean and standard error of the mean*

Description

For use with `stat_summary()`

Usage

```
mean_se(x, mult = 1)
```

Arguments

`x`	numeric vector.
`mult`	number of multiples of standard error.

Value

A data frame with three columns:

`y` The mean.

`ymin` The mean minus the multiples of the standard error.

`ymax` The mean plus the multiples of the standard error.

Examples

```
set.seed(1)
x <- rnorm(100)
mean_se(x)
```

`midwest` *Midwest demographics*

Description

Demographic information of midwest counties from 2000 US census

Usage

```
midwest
```

Format

A data frame with 437 rows and 28 variables:

PID Unique county identifier.

county County name.

state State to which county belongs to.

area Area of county (units unknown).

poptotal Total population.

popdensity Population density (person/unit area).

popwhite Number of whites.

popblack Number of blacks.

popamerindian Number of American Indians.

popasian Number of Asians.

popother Number of other races.

percwhite Percent white.

percblack Percent black.

percamerindan Percent American Indian.

percasian Percent Asian.

percother Percent other races.

popadults Number of adults.

perchsd Percent with high school diploma.

percollege Percent college educated.

percprof Percent with professional degree.

poppovertyknown Population with known poverty status.

percpovertyknown Percent of population with known poverty status.

percbelowpoverty Percent of people below poverty line.

percchildbelowpovert Percent of children below poverty line.

percadultpoverty Percent of adults below poverty line.

percelderlypoverty Percent of elderly below poverty line.

inmetro County considered in a metro area.

category Miscellaneous.

Details

Note: this dataset is included for illustrative purposes. The original descriptions were not documented and the current descriptions here are based on speculation. For more accurate and up-to-date US census data, see the `acs` package.

`mpg` *Fuel economy data from 1999 to 2008 for 38 popular models of cars*

Description

This dataset contains a subset of the fuel economy data that the EPA makes available on `https://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 manufacturer name

model model name

displ engine displacement, in litres

year year of manufacture

cyl number of cylinders

trans type of transmission

drv the type of drive train, where f = front-wheel drive, r = rear wheel drive, 4 = 4wd

cty city miles per gallon

hwy highway miles per gallon

fl fuel type

class "type" of car

`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 *Dodge overlapping objects side-to-side*

Description

Dodging preserves the vertical position of an geom while adjusting the horizontal position. `position_dodge()` requires the grouping variable to be be specified in the global or `geom_*` layer. Unlike `position_dodge()`, `position_dodge2()` works without a grouping variable in a layer. `position_dodge2()` works with bars and rectangles, but is particularly useful for arranging box plots, which can have variable widths.

Usage

```
position_dodge(
  width = NULL,
  preserve = "total",
  orientation = "x",
  reverse = FALSE
)

position_dodge2(
  width = NULL,
  preserve = "total",
  padding = 0.1,
  reverse = FALSE
)
```

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.
`preserve`	Should dodging preserve the `"total"` width of all elements at a position, or the width of a `"single"` element?
`orientation`	Fallback orientation when the layer or the data does not indicate an explicit orientation, like `geom_point()`. Can be `"x"` (default) or `"y"`.
`reverse`	If `TRUE`, will reverse the default stacking order. This is useful if you're rotating both the plot and legend.
`padding`	Padding between elements at the same position. Elements are shrunk by this proportion to allow space between them. Defaults to 0.1.

Aesthetics

`position_dodge()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `order` → NULL

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

Other position adjustments: `position_identity()`, `position_jitter()`, `position_jitterdodge()`, `position_nudge()`, `position_stack()`

Examples

```
ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) +
  geom_bar(position = "dodge2")

# By default, dodging with `position_dodge2()` preserves the total width of
# the elements. You can choose to preserve the width of each element with:
ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) +
  geom_bar(position = position_dodge2(preserve = "single"))


ggplot(diamonds, aes(price, fill = cut)) +
  geom_histogram(position="dodge2")
# see ?geom_bar for more examples

# In this case a frequency polygon is probably a better choice
ggplot(diamonds, aes(price, colour = cut)) +
  geom_freqpoly()


# Dodging with various widths -------------------------------------
# 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)) +
  geom_col(position = "dodge", fill = "grey50", colour = "black")
p

# A line range has no width:
p + geom_linerange(aes(ymin = y - 1, ymax = y + 1), position = "dodge")

# So you must explicitly specify the width
p + geom_linerange(
  aes(ymin = y - 1, ymax = y + 1),
  position = position_dodge(width = 0.9)
)

# The same principle applies to error bars, which are usually
# narrower than the 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)
)

# Box plots use position_dodge2 by default, and bars can use it too
ggplot(mpg, aes(factor(year), displ)) +
  geom_boxplot(aes(colour = hwy < 30))

ggplot(mpg, aes(factor(year), displ)) +
  geom_boxplot(aes(colour = hwy < 30), varwidth = TRUE)

ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) +
  geom_bar(position = position_dodge2(preserve = "single"))

ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) +
  geom_bar(position = position_dodge2(preserve = "total"))
```

---

position_identity *Don't adjust position*

Description

Don't adjust position

Usage

```
position_identity()
```

See Also

Other position adjustments: position_dodge(), position_jitter(), position_jitterdodge(), position_nudge(), position_stack()

position_jitter *Jitter points to avoid overplotting*

Description

Counterintuitively adding random noise to a plot can sometimes make it easier to read. Jittering is particularly useful for small datasets with at least one discrete position.

Usage

```
position_jitter(width = NULL, height = NULL, seed = NA)
```

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.
`seed`	A random seed to make the jitter reproducible. Useful if you need to apply the same jitter twice, e.g., for a point and a corresponding label. The random seed is reset after jittering. If `NA` (the default value), the seed is initialised with a random value; this makes sure that two subsequent calls start with a different seed. Use `NULL` to use the current random seed and also avoid resetting (the behaviour of **ggplot** 2.2.1 and earlier).

See Also

Other position adjustments: `position_dodge()`, `position_identity()`, `position_jitterdodge()`, `position_nudge()`, `position_stack()`

Examples

```
# Jittering is useful when you have a discrete position, and a relatively
# small number of points
# take up as much space as a boxplot or a bar
ggplot(mpg, aes(class, hwy)) +
  geom_boxplot(colour = "grey50") +
  geom_jitter()

# If the default jittering is too much, as in this plot:
ggplot(mtcars, aes(am, vs)) +
  geom_jitter()

# You can adjust it in two ways
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))

# Create a jitter object for reproducible jitter:
jitter <- position_jitter(width = 0.1, height = 0.1, seed = 0)
ggplot(mtcars, aes(am, vs)) +
  geom_point(position = jitter) +
  geom_point(position = jitter, color = "red", aes(am + 0.2, vs + 0.2))
```

---

position_jitterdodge *Simultaneously dodge and jitter*

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

```
position_jitterdodge(
  jitter.width = NULL,
  jitter.height = 0,
  dodge.width = 0.75,
  reverse = FALSE,
  seed = NA
)
```

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.
reverse	If TRUE, will reverse the default stacking order. This is useful if you're rotating both the plot and legend.
seed	A random seed to make the jitter reproducible. Useful if you need to apply the same jitter twice, e.g., for a point and a corresponding label. The random seed is reset after jittering. If NA (the default value), the seed is initialised with a random value; this makes sure that two subsequent calls start with a different seed. Use NULL to use the current random seed and also avoid resetting (the behaviour of **ggplot** 2.2.1 and earlier).

See Also

Other position adjustments: position_dodge(), position_identity(), position_jitter(), position_nudge(), position_stack()

Examples

```
set.seed(596)
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 a fixed distance*

Description

`position_nudge()` is generally useful for adjusting the position of items on discrete scales by a small amount. Nudging is built in to `geom_text()` because it's so useful for moving labels a small distance from what they're labelling.

Usage

```
position_nudge(x = NULL, y = NULL)
```

Arguments

`x, y`	Amount of vertical and horizontal distance to move.

Aesthetics

`position_nudge()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `nudge_x` → `0`
- `nudge_y` → `0`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

Other position adjustments: `position_dodge()`, `position_identity()`, `position_jitter()`, `position_jitterdodge()`, `position_stack()`

Examples

```
df <- data.frame(
  x = c(1,3,2,5),
  y = c("a","c","d","c")
)

ggplot(df, aes(x, y)) +
  geom_point() +
  geom_text(aes(label = y))

ggplot(df, aes(x, y)) +
  geom_point() +
  geom_text(aes(label = y), position = position_nudge(y = -0.1))

# Or, in brief
ggplot(df, aes(x, y)) +
  geom_point() +
  geom_text(aes(label = y), nudge_y = -0.1)

# For each text individually
ggplot(df, aes(x, y)) +
  geom_point() +
  geom_text(aes(label = y, nudge_y = c(-0.1, 0.1, -0.1, 0.1)))
```

`position_stack` *Stack overlapping objects on top of each another*

Description

`position_stack()` stacks bars on top of each other; `position_fill()` stacks bars and standardises each stack to have constant height.

Usage

```
position_stack(vjust = 1, reverse = FALSE)

position_fill(vjust = 1, reverse = FALSE)
```

Arguments

`vjust`	Vertical adjustment for geoms that have a position (like points or lines), not a dimension (like bars or areas). Set to `0` to align with the bottom, `0.5` for the middle, and `1` (the default) for the top.
`reverse`	If `TRUE`, will reverse the default stacking order. This is useful if you're rotating both the plot and legend.

Details

`position_fill()` and `position_stack()` automatically stack values in reverse order of the group aesthetic, which for bar charts is usually defined by the fill aesthetic (the default group aesthetic is formed by the combination of all discrete aesthetics except for x and y). This default ensures that bar colours align with the default legend.

There are three ways to override the defaults depending on what you want:

1. Change the order of the levels in the underlying factor. This will change the stacking order, and the order of keys in the legend.
2. Set the legend `breaks` to change the order of the keys without affecting the stacking.
3. Manually set the group aesthetic to change the stacking order without affecting the legend.

Stacking of positive and negative values are performed separately so that positive values stack upwards from the x-axis and negative values stack downward.

Because stacking is performed after scale transformations, stacking with non-linear scales gives distortions that easily lead to misinterpretations of the data. It is therefore *discouraged* to use these position adjustments in combination with scale transformations, such as logarithmic or square root scales.

See Also

See `geom_bar()` and `geom_area()` for more examples.

Other position adjustments: `position_dodge()`, `position_identity()`, `position_jitter()`, `position_jitterdodge()`, `position_nudge()`

Examples

```
# Stacking and filling ------------------------------------------------------

# Stacking is the default behaviour for most area plots.
# Fill makes it easier to compare proportions
ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) +
  geom_bar()
ggplot(mtcars, aes(factor(cyl), fill = factor(vs))) +
  geom_bar(position = "fill")

ggplot(diamonds, aes(price, fill = cut)) +
  geom_histogram(binwidth = 500)
ggplot(diamonds, aes(price, fill = cut)) +
  geom_histogram(binwidth = 500, position = "fill")

# Stacking is also useful for time series
set.seed(1)
series <- 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(series, aes(time, value)) +
  geom_area(aes(fill = type))

# Stacking order --------------------------------------------------------------
# The stacking order is carefully designed so that the plot matches
# the legend.

# You control the stacking order by setting the levels of the underlying
# factor. See the forcats package for convenient helpers.
series$type2 <- factor(series$type, levels = c('c', 'b', 'd', 'a'))
ggplot(series, aes(time, value)) +
  geom_area(aes(fill = type2))

# You can change the order of the levels in the legend using the scale
ggplot(series, aes(time, value)) +
  geom_area(aes(fill = type)) +
  scale_fill_discrete(breaks = c('a', 'b', 'c', 'd'))

# If you've flipped the plot, use reverse = TRUE so the levels
# continue to match
ggplot(series, aes(time, value)) +
  geom_area(aes(fill = type2), position = position_stack(reverse = TRUE)) +
  coord_flip() +
  theme(legend.position = "top")

# Non-area plots --------------------------------------------------------------

# When stacking across multiple layers it's a good idea to always set
# the `group` aesthetic in the ggplot() call. This ensures that all layers
# are stacked in the same way.
ggplot(series, aes(time, value, group = type)) +
  geom_line(aes(colour = type), position = "stack") +
  geom_point(aes(colour = type), position = "stack")

ggplot(series, aes(time, value, group = type)) +
  geom_area(aes(fill = type)) +
  geom_line(aes(group = type), position = "stack")

# You can also stack labels, but the default position is suboptimal.
ggplot(series, aes(time, value, group = type)) +
  geom_area(aes(fill = type)) +
  geom_text(aes(label = type), position = "stack")

# You can override this with the vjust parameter. A vjust of 0.5
# will center the labels inside the corresponding area
ggplot(series, aes(time, value, group = type)) +
  geom_area(aes(fill = type)) +
  geom_text(aes(label = type), position = position_stack(vjust = 0.5))

# Negative values -------------------------------------------------------------

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

ggplot(data = df, aes(x, y, group = grp)) +
  geom_col(aes(fill = grp), position = position_stack(reverse = TRUE)) +
  geom_hline(yintercept = 0)

ggplot(data = df, aes(x, y, group = grp)) +
  geom_col(aes(fill = grp)) +
  geom_hline(yintercept = 0) +
  geom_text(aes(label = grp), position = position_stack(vjust = 0.5))
```

`presidential` *Terms of 12 presidents from Eisenhower to Trump*

Description

The names of each president, the start and end date of their term, and their party of 12 US presidents from Eisenhower to Trump. This data is in the public domain.

Usage

```
presidential
```

Format

A data frame with 12 rows and 4 variables:

name Last name of president

start Presidency start date

end Presidency end date

party Party of president

`print.ggplot` *Explicitly draw plot*

Description

Generally, you do not need to print or plot a ggplot2 plot explicitly: the default top-level print method will do it for you. You will, however, need to call `print()` explicitly if you want to draw a plot inside a function or for loop.

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 original plot.

Examples

```
colours <- c("class", "drv", "fl")

# Doesn't seem to do anything!
for (colour in colours) {
  ggplot(mpg, aes(displ, hwy, colour = .data[[colour]])) +
    geom_point()
}

for (colour in colours) {
  print(ggplot(mpg, aes(displ, hwy, colour = .data[[colour]])) +
          geom_point())
}
```

`print.ggproto` *Format or print a ggproto object*

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

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

## 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.

Examples

```
Dog <- ggproto(
  print = function(self, n) {
    cat("Woof!\n")
  }
 )
Dog
cat(format(Dog), "\n")
```

`qplot` *Quick plot*

Description

`qplot()` is now deprecated in order to encourage the users to learn `ggplot()` as it makes it easier to create complex graphics.

Usage

```
qplot(
  x,
  y,
  ...,
  data,
  facets = NULL,
  margins = FALSE,
  geom = "auto",
  xlim = c(NA, NA),
  ylim = c(NA, NA),
  log = "",
  main = NULL,
  xlab = NULL,
  ylab = NULL,
  asp = NA,
  stat = deprecated(),
  position = deprecated()
)

quickplot(
  x,
  y,
  ...,
  data,
  facets = NULL,
  margins = FALSE,
  geom = "auto",
  xlim = c(NA, NA),
  ylim = c(NA, NA),
  log = "",
  main = NULL,
  xlab = NULL,
  ylab = NULL,
  asp = NA,
  stat = deprecated(),
  position = deprecated()
)
```

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)
qplot(mpg, wt, data = mtcars, facets = vs ~ am)


set.seed(1)
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")
```

resolution — *Compute the "resolution" of a numeric vector*

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

Usage

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

Arguments

x	numeric vector
zero	should a zero value be automatically included in the computation of resolution
discrete	should vectors mapped with a discrete scale be treated as having a resolution of 1?

Examples

```
resolution(1:10)
resolution((1:10) - 0.5)
resolution((1:10) - 0.5, FALSE)

# Note the difference between numeric and integer vectors
resolution(c(2, 10, 20, 50))
resolution(c(2L, 10L, 20L, 50L))
```

`scale_alpha` *Alpha transparency scales*

Description

Alpha-transparency scales are not tremendously useful, but can be a convenient way to visually down-weight less important observations. `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.

Usage

```
scale_alpha(name = waiver(), ..., range = NULL, aesthetics = "alpha")

scale_alpha_continuous(
  name = waiver(),
  ...,
  range = NULL,
  aesthetics = "alpha"
)

scale_alpha_binned(name = waiver(), ..., range = NULL, aesthetics = "alpha")

scale_alpha_discrete(...)

scale_alpha_ordinal(name = waiver(), ..., range = NULL, aesthetics = "alpha")
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Other arguments passed on to `continuous_scale()`, `binned_scale()`, or `discrete_scale()` as appropriate, to control name, limits, breaks, labels and so forth.
`range`	Output range of alpha values. Must lie between 0 and 1.
`aesthetics`	The names of the aesthetics that this scale works with.

See Also

The documentation on colour aesthetics.

Other alpha scales: `scale_alpha_manual()`, `scale_alpha_identity()`.

The alpha scales section of the online ggplot2 book.

Other colour scales: `scale_colour_brewer()`, `scale_colour_continuous()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_hue()`, `scale_colour_identity()`, `scale_colour_manual()`, `scale_colour_steps()`, `scale_colour_viridis_d()`

Examples

```
p <- ggplot(mpg, aes(displ, hwy)) +
  geom_point(aes(alpha = year))

# The default range of 0.1-1.0 leaves all data visible
p

# Include 0 in the range to make data invisible
p + scale_alpha(range = c(0, 1))

# Changing the title
p + scale_alpha("cylinders")
```

`scale_binned` *Positional scales for binning continuous data (x & y)*

Description

`scale_x_binned()` and `scale_y_binned()` are scales that discretize continuous position data. You can use these scales to transform continuous inputs before using it with a geom that requires discrete positions. An example is using `scale_x_binned()` with `geom_bar()` to create a histogram.

Usage

```
scale_x_binned(
  name = waiver(),
  n.breaks = 10,
  nice.breaks = TRUE,
  breaks = waiver(),
  labels = waiver(),
  limits = NULL,
  expand = waiver(),
  oob = squish,
  na.value = NA_real_,
  right = TRUE,
  show.limits = FALSE,
  transform = "identity",
  trans = deprecated(),
  guide = waiver(),
  position = "bottom"
)

scale_y_binned(
  name = waiver(),
  n.breaks = 10,
  nice.breaks = TRUE,
  breaks = waiver(),
  labels = waiver(),
  limits = NULL,
  expand = waiver(),
  oob = squish,
  na.value = NA_real_,
  right = TRUE,
  show.limits = FALSE,
  transform = "identity",
  trans = deprecated(),
  guide = waiver(),
  position = "left"
)
```

Arguments

`name` The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.

`n.breaks` The number of break points to create if breaks are not given directly.

`nice.breaks` Logical. Should breaks be attempted placed at nice values instead of exactly evenly spaced between the limits. If `TRUE` (default) the scale will ask the transformation object to create breaks, and this may result in a different number of breaks than requested. Ignored if breaks are given explicitly.

`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 (e.g., a function returned by `scales::extended_breaks()`). Note that for position scales, limits are provided after scale expansion. Also accepts rlang lambda function notation.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `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`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`limits` One of:

- `NULL` to use the default scale range
- A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum

	• A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).
`expand`	For position scales, a vector of range expansion constants used to add some padding around the data to ensure that they are placed some distance away from the axes. Use the convenience function `expansion()` to generate the values for the `expand` argument. The defaults are to expand the scale by 5% on each side for continuous variables, and by 0.6 units on each side for discrete variables.
`oob`	One of: • Function that handles limits outside of the scale limits (out of bounds). Also accepts rlang lambda function notation. • The default (`scales::squish()`) squishes out of bounds values into range. • scales::censor for replacing out of bounds values with `NA`. • `scales::squish_infinite()` for squishing infinite values into range.
`na.value`	Missing values will be replaced with this value.
`right`	Should the intervals be closed on the right (`TRUE`, default) or should the intervals be closed on the left (`FALSE`)? 'Closed on the right' means that values at break positions are part of the lower bin (open on the left), whereas they are part of the upper bin when intervals are closed on the left (open on the right).
`show.limits`	should the limits of the scale appear as ticks
`transform`	For continuous scales, the name of a transformation object or the object itself. Built-in transformations include "asn", "atanh", "boxcox", "date", "exp", "hms", "identity", "log", "log10", "log1p", "log2", "logit", "modulus", "probability", "probit", "pseudo_log", "reciprocal", "reverse", "sqrt" and "time". 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 `transform_<name>`. If transformations require arguments, you can call them from the scales package, e.g. `scales::transform_boxcox(p = 2)`. You can create your own transformation with `scales::new_transform()`.
`trans`	**[Deprecated]** Deprecated in favour of `transform`.
`guide`	A function used to create a guide or its name. See `guides()` for more information.
`position`	For position scales, The position of the axis. `left` or `right` for y axes, `top` or `bottom` for x axes.

See Also

The position documentation.

The binned position scales section of the online ggplot2 book.

Other position scales: `scale_x_continuous()`, `scale_x_date()`, `scale_x_discrete()`

Examples

```
# Create a histogram by binning the x-axis
ggplot(mtcars) +
  geom_bar(aes(mpg)) +
  scale_x_binned()
```

`scale_colour_brewer` *Sequential, diverging and qualitative colour scales from ColorBrewer*

Description

The `brewer` scales provide sequential, diverging and qualitative colour schemes from ColorBrewer. These are particularly well suited to display discrete values on a map. See `https://colorbrewer2.org` for more information.

Usage

```
scale_colour_brewer(
  name = waiver(),
  ...,
  type = "seq",
  palette = 1,
  direction = 1,
  aesthetics = "colour"
)

scale_fill_brewer(
  name = waiver(),
  ...,
  type = "seq",
  palette = 1,
  direction = 1,
  aesthetics = "fill"
)

scale_colour_distiller(
  name = waiver(),
  ...,
  type = "seq",
  palette = 1,
  direction = -1,
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "colour"
)
```

```
scale_fill_distiller(
  name = waiver(),
  ...,
  type = "seq",
  palette = 1,
  direction = -1,
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "fill"
)

scale_colour_fermenter(
  name = waiver(),
  ...,
  type = "seq",
  palette = 1,
  direction = -1,
  na.value = "grey50",
  guide = "coloursteps",
  aesthetics = "colour"
)

scale_fill_fermenter(
  name = waiver(),
  ...,
  type = "seq",
  palette = 1,
  direction = -1,
  na.value = "grey50",
  guide = "coloursteps",
  aesthetics = "fill"
)
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Other arguments passed on to `discrete_scale()`, `continuous_scale()`, or `binned_scale()`, for `brewer`, `distiller`, and `fermenter` variants respectively, 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`. The list of available palettes can found in the Palettes section.

`direction`	Sets the order of colours in the scale. If 1, the default, colours are as output by `RColorBrewer::brewer.pal()`. If -1, the order of colours is reversed.
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the `colour` and `fill` aesthetics at the same time, via `aesthetics = c("colour", "fill")`.
`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.
`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.

Details

The `brewer` scales were carefully designed and tested on discrete data. They were not designed to be extended to continuous data, but results often look good. Your mileage may vary.

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, Greys, Oranges, OrRd, PuBu, PuBuGn, PuRd, Purples, RdPu, Reds, YlGn, YlGnBu, YlOrBr, YlOrRd

Modify the palette through the `palette` argument.

Note

The `distiller` scales extend `brewer` scales by smoothly interpolating 7 colours from any palette to a continuous scale. The `distiller` scales have a default direction = -1. To reverse, use direction = 1. The `fermenter` scales provide binned versions of the `brewer` scales.

See Also

The documentation on colour aesthetics.

The brewer scales section of the online ggplot2 book.

Other colour scales: `scale_alpha()`, `scale_colour_continuous()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_hue()`, `scale_colour_identity()`, `scale_colour_manual()`, `scale_colour_steps()`, `scale_colour_viridis_d()`

Examples

```
set.seed(596)
dsamp <- diamonds[sample(nrow(diamonds), 1000), ]
(d <- ggplot(dsamp, aes(carat, price)) +
  geom_point(aes(colour = clarity)))
d + scale_colour_brewer()

# Change scale label
d + scale_colour_brewer("Diamond\nclarity")

# Select brewer palette to use, see ?scales::pal_brewer 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 continuous data
v <- ggplot(faithfuld) +
  geom_tile(aes(waiting, eruptions, fill = density))
v
v + scale_fill_distiller()
v + scale_fill_distiller(palette = "Spectral")
# the order of colour can be reversed, but with scale_*_distiller(),
# the default direction = -1, so to reverse, use direction = 1.
v + scale_fill_distiller(palette = "Spectral", direction = 1)

# or use blender variants to discretise continuous data
v + scale_fill_fermenter()
```

`scale_colour_continuous`

Continuous and binned colour scales

Description

The scales `scale_colour_continuous()` and `scale_fill_continuous()` are the default colour scales ggplot2 uses when continuous data values are mapped onto the `colour` or `fill` aesthetics, respectively. The scales `scale_colour_binned()` and `scale_fill_binned()` are equivalent scale functions that assign discrete color bins to the continuous values instead of using a continuous color spectrum.

Usage

```
scale_colour_continuous(
  ...,
  palette = NULL,
  aesthetics = "colour",
  guide = "colourbar",
  na.value = "grey50",
  type = getOption("ggplot2.continuous.colour")
)

scale_fill_continuous(
  ...,
  palette = NULL,
  aesthetics = "fill",
  guide = "colourbar",
  na.value = "grey50",
  type = getOption("ggplot2.continuous.fill")
)

scale_colour_binned(
  ...,
  palette = NULL,
  aesthetics = "colour",
  guide = "coloursteps",
  na.value = "grey50",
  type = getOption("ggplot2.binned.colour")
)

scale_fill_binned(
  ...,
  palette = NULL,
  aesthetics = "fill",
  guide = "coloursteps",
  na.value = "grey50",
  type = getOption("ggplot2.binned.fill")
)
```

Arguments

`...`	Additional parameters passed on to the scale type
`palette`	One of the following: • `NULL` for the default palette stored in the theme. • a character vector of colours.

- a single string naming a palette.
- a palette function that when called with a numeric vector with values between 0 and 1 returns the corresponding output values.

`aesthetics`	The names of the aesthetics that this scale works with.
`guide`	A function used to create a guide or its name. See `guides()` for more information.
`na.value`	Missing values will be replaced with this value.
`type`	**[Superseded]** One of the following: • "gradient" (the default) • "viridis" • A function that returns a continuous colour scale.

Details

[Superseded]: The mechanism of setting defaults via `options()` is superseded by theme settings. The preferred method to change the default palette of scales is via the theme, for example: `theme(palette.colour.continuous = scales::pal_viridis())`. The `ggplot2.continuous.colour` and `ggplot2.continuous.fill` options could be used to set default continuous scales and `ggplot2.binned.colour` and `ggplot2.binned.fill` options to set default binned scales.

These scale functions are meant to provide simple defaults. If you want to manually set the colors of a scale, consider using `scale_colour_gradient()` or `scale_colour_steps()`.

Color Blindness

Many color palettes derived from RGB combinations (like the "rainbow" color palette) are not suitable to support all viewers, especially those with color vision deficiencies. Using `viridis` type, which is perceptually uniform in both colour and black-and-white display is an easy option to ensure good perceptive properties of your visualizations. The colorspace package offers functionalities

- to generate color palettes with good perceptive properties,
- to analyse a given color palette, like emulating color blindness,
- and to modify a given color palette for better perceptivity.

For more information on color vision deficiencies and suitable color choices see the paper on the colorspace package and references therein.

See Also

`scale_colour_gradient()`, `scale_colour_viridis_c()`, `scale_colour_steps()`, `scale_colour_viridis_b()`, `scale_fill_gradient()`, `scale_fill_viridis_c()`, `scale_fill_steps()`, and `scale_fill_viridis_b()`

The documentation on colour aesthetics.

The continuous colour scales section of the online ggplot2 book.

Other colour scales: `scale_alpha()`, `scale_colour_brewer()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_hue()`, `scale_colour_identity()`, `scale_colour_manual()`, `scale_colour_steps()`, `scale_colour_viridis_d()`

Examples

```
# A standard plot
p <- ggplot(mpg, aes(displ, hwy, colour = cty)) +
  geom_point()

# You can use the scale to give a palette directly
p + scale_colour_continuous(palette = c("#FEE0D2", "#FC9272", "#DE2D26"))

# The default colours are encoded into the theme
p + theme(palette.colour.continuous = c("#DEEBF7", "#9ECAE1", "#3182BD"))

# You can globally set default colour palette via the theme
old <- update_theme(palette.colour.continuous = c("#E5F5E0", "#A1D99B", "#31A354"))

# Plot now shows new global default
p

# The default binned colour scale uses the continuous palette
p + scale_colour_binned() +
  theme(palette.colour.continuous = c("#EFEDF5", "#BCBDDC", "#756BB1"))

# Restoring the previous theme
theme_set(old)
```

---

scale_colour_discrete *Discrete colour scales*

Description

The default discrete colour scale.

Usage

```
scale_colour_discrete(
  ...,
  palette = NULL,
  aesthetics = "colour",
  na.value = "grey50",
  type = getOption("ggplot2.discrete.colour")
)

scale_fill_discrete(
  ...,
  palette = NULL,
  aesthetics = "fill",
  na.value = "grey50",
  type = getOption("ggplot2.discrete.fill")
)
```

Arguments

`...`	Additional parameters passed on to the scale type,
`palette`	One of the following: • `NULL` for the default palette stored in the theme. • a character vector of colours. • a single string naming a palette. • a palette function that when called with a single integer argument (the number of levels in the scale) returns the values that they should take.
`aesthetics`	The names of the aesthetics that this scale works with.
`na.value`	If `na.translate = TRUE`, what aesthetic value should the missing values be displayed as? Does not apply to position scales where `NA` is always placed at the far right.
`type`	**[Superseded]** The preferred mechanism for setting the default palette is by using the theme. For example: `theme(palette.colour.discrete = "Okabe-Ito")`. One of the following: • A character vector of color codes. The codes are used for a 'manual' color scale as long as the number of codes exceeds the number of data levels (if there are more levels than codes, `scale_colour_hue()`/`scale_fill_hue()` are used to construct the default scale). If this is a named vector, then the color values will be matched to levels based on the names of the vectors. Data values that don't match will be set as `na.value`. • A list of character vectors of color codes. The minimum length vector that exceeds the number of data levels is chosen for the color scaling. This is useful if you want to change the color palette based on the number of levels. • A function that returns a discrete colour/fill scale (e.g., `scale_fill_hue()`, `scale_fill_brewer()`, etc).

See Also

The discrete colour scales section of the online ggplot2 book.

Examples

```
# A standard plot
p <- ggplot(mpg, aes(displ, hwy, colour = class)) +
  geom_point()

# You can use the scale to give a palette directly
p + scale_colour_discrete(palette = scales::pal_brewer(palette = "Dark2"))

# The default colours are encoded into the theme
p + theme(palette.colour.discrete = scales::pal_grey())

# You can globally set default colour palette via the theme
old <- update_theme(palette.colour.discrete = scales::pal_viridis())

# Plot now shows new global default
```

```
p

# Restoring the previous theme
theme_set(old)
```

---

scale_colour_gradient *Gradient colour scales*

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. For binned variants of these scales, see the color steps scales.

Usage

```
scale_colour_gradient(
  name = waiver(),
  ...,
  low = "#132B43",
  high = "#56B1F7",
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "colour"
)

scale_fill_gradient(
  name = waiver(),
  ...,
  low = "#132B43",
  high = "#56B1F7",
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "fill"
)

scale_colour_gradient2(
  name = waiver(),
  ...,
  low = muted("red"),
  mid = "white",
  high = muted("blue"),
  midpoint = 0,
  space = "Lab",
  na.value = "grey50",
  transform = "identity",
  guide = "colourbar",
  aesthetics = "colour"
)

scale_fill_gradient2(
  name = waiver(),
  ...,
  low = muted("red"),
  mid = "white",
  high = muted("blue"),
  midpoint = 0,
  space = "Lab",
  na.value = "grey50",
  transform = "identity",
  guide = "colourbar",
  aesthetics = "fill"
)

scale_colour_gradientn(
  name = waiver(),
  ...,
  colours,
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "colour",
  colors
)

scale_fill_gradientn(
  name = waiver(),
  ...,
  colours,
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "fill",
  colors
)
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Arguments passed on to `continuous_scale`

`scale_name` **[Deprecated]** The name of the scale that should be used for error messages associated with this scale.

`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 (e.g., a function returned by `scales::extended_breaks()`). Note that for position scales, limits are provided after scale expansion. Also accepts rlang lambda function notation.

`minor_breaks` One of:

- `NULL` for no minor breaks
- `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous)
- A numeric vector of positions
- A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.

`n.breaks` An integer guiding the number of major breaks. The algorithm may choose a slightly different number to ensure nice break labels. Will only have an effect if `breaks = waiver()`. Use `NULL` to use the default number of breaks given by the transformation.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `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`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`limits` One of:

- `NULL` to use the default scale range
- A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum
- A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).

`rescaler` A function used to scale the input values to the range [0, 1]. This is always `scales::rescale()`, except for diverging and n colour gradients (i.e., `scale_colour_gradient2()`, `scale_colour_gradientn()`). The `rescaler` is ignored by position scales, which always use `scales::rescale()`. Also accepts rlang lambda function notation.

	`oob` One of: • Function that handles limits outside of the scale limits (out of bounds). Also accepts rlang lambda function notation. • The default (`scales::censor()`) replaces out of bounds values with `NA`. • `scales::squish()` for squishing out of bounds values into range. • `scales::squish_infinite()` for squishing infinite values into range. `trans` **[Deprecated]** Deprecated in favour of `transform`. `call` The `call` used to construct the scale for reporting messages. `super` The super class to use for the constructed scale
`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.
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the `colour` and `fill` aesthetics at the same time, via `aesthetics = c("colour", "fill")`.
`mid`	colour for mid point
`midpoint`	The midpoint (in data value) of the diverging scale. Defaults to 0.
`transform`	For continuous scales, the name of a transformation object or the object itself. Built-in transformations include "asn", "atanh", "boxcox", "date", "exp", "hms", "identity", "log", "log10", "log1p", "log2", "logit", "modulus", "probability", "probit", "pseudo_log", "reciprocal", "reverse", "sqrt" and "time". 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 `transform_<name>`. If transformations require arguments, you can call them from the scales package, e.g. `scales::transform_boxcox(p = 2)`. You can create your own transformation with `scales::new_transform()`.
`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 `mnsl(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

scales::pal_seq_gradient() for details on underlying palette, scale_colour_steps() for binned variants of these scales.

The documentation on colour aesthetics.

Other colour scales: scale_alpha(), scale_colour_brewer(), scale_colour_continuous(), scale_colour_grey(), scale_colour_hue(), scale_colour_identity(), scale_colour_manual(), scale_colour_steps(), scale_colour_viridis_d()

Examples

```
set.seed(1)
df <- data.frame(
  x = runif(100),
  y = runif(100),
  z1 = rnorm(100),
  z2 = abs(rnorm(100))
)

df_na <- data.frame(
  value = seq(1, 20),
  x = runif(20),
  y = runif(20),
  z1 = c(rep(NA, 10), rnorm(10))
)

# 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 = 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))

# The gradientn scale can be centered by using a rescaler
ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z1)) +
  scale_colour_gradientn(
    colours = c("blue", "dodgerblue", "white", "orange", "red"),
    rescaler = ~ scales::rescale_mid(.x, mid = 0)
  )

# Equivalent fill scales do the same job for the fill aesthetic
ggplot(faithfuld, 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_gradient(low = "white", high = "black")
# Avoid red-green colour contrasts because ~10% of men have difficulty
# seeing them

# Use `na.value = NA` to hide missing values but keep the original axis range
ggplot(df_na, aes(x = value, y)) +
  geom_bar(aes(fill = z1), stat = "identity") +
  scale_fill_gradient(low = "yellow", high = "red", na.value = NA)

 ggplot(df_na, aes(x, y)) +
   geom_point(aes(colour = z1)) +
   scale_colour_gradient(low = "yellow", high = "red", na.value = NA)
```

---

scale_colour_grey *Sequential grey colour scales*

Description

Based on `gray.colors()`. This is black and white equivalent of `scale_colour_gradient()`.

Usage

```
scale_colour_grey(
  name = waiver(),
  ...,
  start = 0.2,
  end = 0.8,
  na.value = "red",
  aesthetics = "colour"
)

scale_fill_grey(
  name = waiver(),
  ...,
  start = 0.2,
  end = 0.8,
  na.value = "red",
  aesthetics = "fill"
)
```

Arguments

name	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.

...	Arguments passed on to discrete_scale

breaks One of:

- NULL for no breaks
- waiver() for the default breaks (the scale limits)
- A character vector of breaks
- A function that takes the limits as input and returns breaks as output. Also accepts rlang lambda function notation.

limits One of:

- NULL to use the default scale values
- A character vector that defines possible values of the scale and their order
- A function that accepts the existing (automatic) values and returns new ones. Also accepts rlang lambda function notation.

drop Should unused factor levels be omitted from the scale? The default, TRUE, uses the levels that appear in the data; FALSE includes the levels in the factor. Please note that to display every level in a legend, the layer should use show.legend = TRUE.

na.translate Unlike continuous scales, discrete scales can easily show missing values, and do so by default. If you want to remove missing values from a discrete scale, specify na.translate = FALSE.

minor_breaks One of:

- NULL for no minor breaks
- waiver() for the default breaks (none for discrete, one minor break between each major break for continuous)
- A numeric vector of positions
- A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.

labels One of the options below. Please note that when labels is a vector, it is highly recommended to also set the breaks argument as a vector to protect against unintended mismatches.

- 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)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

guide A function used to create a guide or its name. See guides() for more information.

call The call used to construct the scale for reporting messages.

super The super class to use for the constructed scale

start	grey value at low end of palette
end	grey value at high end of palette

`na.value`	Colour to use for missing values
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the `colour` and `fill` aesthetics at the same time, via `aesthetics = c("colour", "fill")`.

See Also

The documentation on colour aesthetics.

The hue and grey scales section of the online ggplot2 book.

Other colour scales: `scale_alpha()`, `scale_colour_brewer()`, `scale_colour_continuous()`, `scale_colour_gradient()`, `scale_colour_hue()`, `scale_colour_identity()`, `scale_colour_manual()`, `scale_colour_steps()`, `scale_colour_viridis_d()`

Examples

```
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` *Evenly spaced colours for discrete data*

Description

Maps each level to an evenly spaced hue on the colour wheel. It does not generate colour-blind safe palettes.

Usage

```
scale_colour_hue(
  name = waiver(),
  ...,
  h = c(0, 360) + 15,
  c = 100,
  l = 65,
  h.start = 0,
  direction = 1,
  na.value = "grey50",
  aesthetics = "colour"
)

scale_fill_hue(
  name = waiver(),
  ...,
  h = c(0, 360) + 15,
  c = 100,
  l = 65,
  h.start = 0,
  direction = 1,
  na.value = "grey50",
  aesthetics = "fill"
)
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Arguments passed on to `discrete_scale`

`breaks` One of:

- `NULL` for no breaks
- `waiver()` for the default breaks (the scale limits)
- A character vector of breaks
- A function that takes the limits as input and returns breaks as output. Also accepts rlang lambda function notation.

`limits` One of:

- `NULL` to use the default scale values
- A character vector that defines possible values of the scale and their order
- A function that accepts the existing (automatic) values and returns new ones. Also accepts rlang lambda function notation.

`drop` Should unused factor levels be omitted from the scale? The default, `TRUE`, uses the levels that appear in the data; `FALSE` includes the levels in the factor. Please note that to display every level in a legend, the layer should use `show.legend = TRUE`.

`na.translate` Unlike continuous scales, discrete scales can easily show missing values, and do so by default. If you want to remove missing values from a discrete scale, specify `na.translate = FALSE`.

`minor_breaks` One of:

- `NULL` for no minor breaks
- `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous)

- A numeric vector of positions
- A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `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`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`guide` A function used to create a guide or its name. See `guides()` for more information.

`call` The `call` used to construct the scale for reporting messages.

`super` The super class to use for the constructed scale

`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
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the `colour` and `fill` aesthetics at the same time, via `aesthetics = c("colour", "fill")`.

See Also

The documentation on colour aesthetics.

The hue and grey scales section of the online ggplot2 book.

Other colour scales: `scale_alpha()`, `scale_colour_brewer()`, `scale_colour_continuous()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_identity()`, `scale_colour_manual()`, `scale_colour_steps()`, `scale_colour_viridis_d()`

Examples

```
set.seed(596)
dsamp <- diamonds[sample(nrow(diamonds), 1000), ]
(d <- ggplot(dsamp, aes(carat, price)) + geom_point(aes(colour = clarity)))

# Change scale label
```

```
d + scale_colour_hue()
d + scale_colour_hue("clarity")
d + scale_colour_hue(expression(clarity[beta]))

# Adjust luminosity and chroma
d + scale_colour_hue(l = 40, c = 30)
d + scale_colour_hue(l = 70, c = 30)
d + scale_colour_hue(l = 70, c = 150)
d + scale_colour_hue(l = 80, c = 150)

# Change range of hues used
d + scale_colour_hue(h = c(0, 90))
d + scale_colour_hue(h = c(90, 180))
d + scale_colour_hue(h = c(180, 270))
d + scale_colour_hue(h = c(270, 360))

# Vary opacity
# (only works with pdf, quartz and cairo devices)
d <- ggplot(dsamp, aes(carat, price, colour = clarity))
d + geom_point(alpha = 0.9)
d + geom_point(alpha = 0.5)
d + geom_point(alpha = 0.2)

# 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))
ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(colour = miss)) +
  scale_colour_hue(na.value = "black")
```

scale_colour_steps *Binned gradient colour scales*

Description

scale_*_steps creates a two colour binned gradient (low-high), scale_*_steps2 creates a diverging binned colour gradient (low-mid-high), and scale_*_stepsn creates a n-colour binned gradient. These scales are binned variants of the gradient scale family and works in the same way.

Usage

```
scale_colour_steps(
  name = waiver(),
  ...,
  low = "#132B43",
  high = "#56B1F7",
  space = "Lab",
  na.value = "grey50",
  guide = "coloursteps",
  aesthetics = "colour"
)

scale_colour_steps2(
  name = waiver(),
  ...,
  low = muted("red"),
  mid = "white",
  high = muted("blue"),
  midpoint = 0,
  space = "Lab",
  na.value = "grey50",
  transform = "identity",
  guide = "coloursteps",
  aesthetics = "colour"
)

scale_colour_stepsn(
  name = waiver(),
  ...,
  colours,
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "coloursteps",
  aesthetics = "colour",
  colors
)

scale_fill_steps(
  name = waiver(),
  ...,
  low = "#132B43",
  high = "#56B1F7",
  space = "Lab",
  na.value = "grey50",
  guide = "coloursteps",
  aesthetics = "fill"
)

scale_fill_steps2(
  name = waiver(),
  ...,
  low = muted("red"),
  mid = "white",
  high = muted("blue"),
  midpoint = 0,
  space = "Lab",
  na.value = "grey50",
  transform = "identity",
  guide = "coloursteps",
  aesthetics = "fill"
)

scale_fill_stepsn(
  name = waiver(),
  ...,
  colours,
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "coloursteps",
  aesthetics = "fill",
  colors
)
```

Arguments

`name` The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.

`...` Arguments passed on to `binned_scale`

`n.breaks` The number of break points to create if breaks are not given directly.

`nice.breaks` Logical. Should breaks be attempted placed at nice values instead of exactly evenly spaced between the limits. If `TRUE` (default) the scale will ask the transformation object to create breaks, and this may result in a different number of breaks than requested. Ignored if breaks are given explicitly.

`oob` One of:

- Function that handles limits outside of the scale limits (out of bounds). Also accepts rlang lambda function notation.
- The default (`scales::squish()`) squishes out of bounds values into range.
- scales::censor for replacing out of bounds values with `NA`.
- `scales::squish_infinite()` for squishing infinite values into range.

`right` Should the intervals be closed on the right (`TRUE`, default) or should the intervals be closed on the left (`FALSE`)? 'Closed on the right' means that values at break positions are part of the lower bin (open on the left), whereas they are part of the upper bin when intervals are closed on the left (open on the right).

`show.limits` should the limits of the scale appear as ticks

`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 (e.g., a function returned by `scales::extended_breaks()`). Note that for position scales, limits are provided after scale expansion. Also accepts rlang lambda function notation.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `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`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`limits` One of:

- `NULL` to use the default scale range
- A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum
- A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).

`trans` **[Deprecated]** Deprecated in favour of `transform`.

`call` The `call` used to construct the scale for reporting messages.

`super` The super class to use for the constructed scale

`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.
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the `colour` and `fill` aesthetics at the same time, via `aesthetics = c("colour", "fill")`.
`mid`	colour for mid point
`midpoint`	The midpoint (in data value) of the diverging scale. Defaults to 0.
`transform`	For continuous scales, the name of a transformation object or the object itself. Built-in transformations include "asn", "atanh", "boxcox", "date", "exp", "hms", "identity", "log", "log10", "log1p", "log2", "logit", "modulus", "probability", "probit", "pseudo_log", "reciprocal", "reverse", "sqrt" and "time".

	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 `transform_<name>`. If transformations require arguments, you can call them from the scales package, e.g. `scales::transform_boxcox(p = 2)`. You can create your own transformation with `scales::new_transform()`.
`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 `mnsl(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

`scales::pal_seq_gradient()` for details on underlying palette, `scale_colour_gradient()` for continuous scales without binning.

The documentation on colour aesthetics.

The binned colour scales section of the online ggplot2 book.

Other colour scales: `scale_alpha()`, `scale_colour_brewer()`, `scale_colour_continuous()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_hue()`, `scale_colour_identity()`, `scale_colour_manual()`, `scale_colour_viridis_d()`

Examples

```
set.seed(1)
df <- data.frame(
  x = runif(100),
  y = runif(100),
  z1 = rnorm(100)
)

# Use scale_colour_steps for a standard binned gradient
ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z1)) +
  scale_colour_steps()

# Get a divergent binned scale with the *2 variant
ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z1)) +
  scale_colour_steps2()

# Define your own colour ramp to extract binned colours from
ggplot(df, aes(x, y)) +
  geom_point(aes(colour = z1)) +
  scale_colour_stepsn(colours = terrain.colors(10))
```

`scale_colour_viridis_d` *Viridis colour scales from viridisLite*

Description

The `viridis` scales provide colour maps that are perceptually uniform in both colour and black-and-white. They are also designed to be perceived by viewers with common forms of colour blindness. See also `https://bids.github.io/colormap/`.

Usage

```
scale_colour_viridis_d(
  name = waiver(),
  ...,
  alpha = 1,
  begin = 0,
  end = 1,
  direction = 1,
  option = "D",
  aesthetics = "colour"
)

scale_fill_viridis_d(
  name = waiver(),
  ...,
  alpha = 1,
  begin = 0,
  end = 1,
  direction = 1,
  option = "D",
  aesthetics = "fill"
)

scale_colour_viridis_c(
  name = waiver(),
  ...,
  alpha = 1,
  begin = 0,
  end = 1,
  direction = 1,
  option = "D",
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "colour"
)

scale_fill_viridis_c(
  name = waiver(),
  ...,
  alpha = 1,
  begin = 0,
  end = 1,
  direction = 1,
  option = "D",
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "colourbar",
  aesthetics = "fill"
)

scale_colour_viridis_b(
  name = waiver(),
  ...,
  alpha = 1,
  begin = 0,
  end = 1,
  direction = 1,
  option = "D",
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "coloursteps",
  aesthetics = "colour"
)

scale_fill_viridis_b(
  name = waiver(),
  ...,
  alpha = 1,
  begin = 0,
  end = 1,
  direction = 1,
  option = "D",
  values = NULL,
  space = "Lab",
  na.value = "grey50",
  guide = "coloursteps",
  aesthetics = "fill"
)
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Other arguments passed on to `discrete_scale()`, `continuous_scale()`, or `binned_scale()` to control name, limits, breaks, labels and so forth.
`alpha`	The alpha transparency, a number in [0,1], see argument alpha in `hsv`.
`begin, end`	The (corrected) hue in `[0,1]` at which the color map begins and ends.
`direction`	Sets the order of colors in the scale. If 1, the default, colors are ordered from darkest to lightest. If -1, the order of colors is reversed.
`option`	A character string indicating the color map option to use. Eight options are available: • `"magma"` (or `"A"`) • `"inferno"` (or `"B"`) • `"plasma"` (or `"C"`) • `"viridis"` (or `"D"`) • `"cividis"` (or `"E"`) • `"rocket"` (or `"F"`) • `"mako"` (or `"G"`) • `"turbo"` (or `"H"`)
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the `colour` and `fill` aesthetics at the same time, via `aesthetics = c("colour", "fill")`.
`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.
`space`	colour space in which to calculate gradient. Must be "Lab" - other values are deprecated.
`na.value`	Missing values will be replaced with this value.
`guide`	A function used to create a guide or its name. See `guides()` for more information.

See Also

The documentation on colour aesthetics.

Other colour scales: `scale_alpha()`, `scale_colour_brewer()`, `scale_colour_continuous()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_hue()`, `scale_colour_identity()`, `scale_colour_manual()`, `scale_colour_steps()`

Examples

```
# viridis is the default colour/fill scale for ordered factors
set.seed(596)
dsamp <- diamonds[sample(nrow(diamonds), 1000), ]
ggplot(dsamp, aes(carat, price)) +
  geom_point(aes(colour = clarity))

# Use viridis_d with discrete data
txsamp <- subset(txhousing, city %in%
  c("Houston", "Fort Worth", "San Antonio", "Dallas", "Austin"))
(d <- ggplot(data = txsamp, aes(x = sales, y = median)) +
   geom_point(aes(colour = city)))
d + scale_colour_viridis_d()

# Change scale label
d + scale_colour_viridis_d("City\nCenter")

# Select palette to use, see ?scales::pal_viridis for more details
d + scale_colour_viridis_d(option = "plasma")
d + scale_colour_viridis_d(option = "inferno")

# scale_fill_viridis_d works just the same as
# scale_colour_viridis_d but for fill colours
p <- ggplot(txsamp, aes(x = median, fill = city)) +
  geom_histogram(position = "dodge", binwidth = 15000)
p + scale_fill_viridis_d()
# the order of colour can be reversed
p + scale_fill_viridis_d(direction = -1)

# Use viridis_c with continuous data
(v <- ggplot(faithfuld) +
  geom_tile(aes(waiting, eruptions, fill = density)))
v + scale_fill_viridis_c()
v + scale_fill_viridis_c(option = "plasma")

# Use viridis_b to bin continuous data before mapping
v + scale_fill_viridis_b()
```

`scale_continuous` *Position scales for continuous data (x & y)*

Description

`scale_x_continuous()` and `scale_y_continuous()` are the default scales for continuous x and y aesthetics. There are three variants that set the `transform` argument for commonly used transformations: `scale_*_log10()`, `scale_*_sqrt()` and `scale_*_reverse()`.

Usage

```
scale_x_continuous(
  name = waiver(),
  breaks = waiver(),
  minor_breaks = waiver(),
  n.breaks = NULL,
  labels = waiver(),
  limits = NULL,
  expand = waiver(),
  oob = censor,
  na.value = NA_real_,
  transform = "identity",
  trans = deprecated(),
  guide = waiver(),
  position = "bottom",
  sec.axis = waiver()
)

scale_y_continuous(
  name = waiver(),
  breaks = waiver(),
  minor_breaks = waiver(),
  n.breaks = NULL,
  labels = waiver(),
  limits = NULL,
  expand = waiver(),
  oob = censor,
  na.value = NA_real_,
  transform = "identity",
  trans = deprecated(),
  guide = waiver(),
  position = "left",
  sec.axis = waiver()
)

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 the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`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 (e.g., a function returned by `scales::extended_breaks()`). Note that for position scales, limits are provided after scale expansion. Also accepts rlang lambda function notation.
`minor_breaks`	One of: • `NULL` for no minor breaks • `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous) • A numeric vector of positions • A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.
`n.breaks`	An integer guiding the number of major breaks. The algorithm may choose a slightly different number to ensure nice break labels. Will only have an effect if `breaks = waiver()`. Use `NULL` to use the default number of breaks given by the transformation.
`labels`	One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches. • `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`) • An expression vector (must be the same length as breaks). See ?plotmath for details. • A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.
`limits`	One of: • `NULL` to use the default scale range • A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum • A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).

`expand`	For position scales, a vector of range expansion constants used to add some padding around the data to ensure that they are placed some distance away from the axes. Use the convenience function `expansion()` to generate the values for the `expand` argument. The defaults are to expand the scale by 5% on each side for continuous variables, and by 0.6 units on each side for discrete variables.
`oob`	One of: • Function that handles limits outside of the scale limits (out of bounds). Also accepts rlang lambda function notation. • The default (`scales::censor()`) replaces out of bounds values with `NA`. • `scales::squish()` for squishing out of bounds values into range. • `scales::squish_infinite()` for squishing infinite values into range.
`na.value`	Missing values will be replaced with this value.
`transform`	For continuous scales, the name of a transformation object or the object itself. Built-in transformations include "asn", "atanh", "boxcox", "date", "exp", "hms", "identity", "log", "log10", "log1p", "log2", "logit", "modulus", "probability", "probit", "pseudo_log", "reciprocal", "reverse", "sqrt" and "time". 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 `transform_<name>`. If transformations require arguments, you can call them from the scales package, e.g. `scales::transform_boxcox(p = 2)`. You can create your own transformation with `scales::new_transform()`.
`trans`	**[Deprecated]** Deprecated in favour of `transform`.
`guide`	A function used to create a guide or its name. See `guides()` for more information.
`position`	For position scales, The position of the axis. `left` or `right` for y axes, `top` or `bottom` for x axes.
`sec.axis`	`sec_axis()` is used to specify a secondary axis.
`...`	Other arguments passed on to `scale_(x|y)_continuous()`

Details

For simple manipulation of labels and limits, you may wish to use `labs()` and `lims()` instead.

See Also

The position documentation.

The numeric position scales section of the online ggplot2 book.

Other position scales: `scale_x_binned()`, `scale_x_date()`, `scale_x_discrete()`

Examples

```
p1 <- ggplot(mpg, aes(displ, hwy)) +
  geom_point()
p1
```

```
# Manipulating the default position scales lets you:
#  * change the axis labels
p1 +
  scale_x_continuous("Engine displacement (L)") +
  scale_y_continuous("Highway MPG")

# You can also use the short-cut labs().
# Use NULL to suppress axis labels
p1 + labs(x = NULL, y = NULL)

#  * modify the axis limits
p1 + scale_x_continuous(limits = c(2, 6))
p1 + scale_x_continuous(limits = c(0, 10))

# you can also use the short hand functions `xlim()` and `ylim()`
p1 + xlim(2, 6)

#  * choose where the ticks appear
p1 + scale_x_continuous(breaks = c(2, 4, 6))

#  * choose your own labels
p1 + scale_x_continuous(
  breaks = c(2, 4, 6),
  label = c("two", "four", "six")
)

# Typically you'll pass a function to the `labels` argument.
# Some common formats are built into the scales package:
set.seed(1)
df <- data.frame(
  x = rnorm(10) * 100000,
  y = seq(0, 1, length.out = 10)
)
p2 <- ggplot(df, aes(x, y)) + geom_point()
p2 + scale_y_continuous(labels = scales::label_percent())
p2 + scale_y_continuous(labels = scales::label_dollar())
p2 + scale_x_continuous(labels = scales::label_comma())

# You can also override the default linear mapping by using a
# transformation. There are three shortcuts:
p1 + scale_y_log10()
p1 + scale_y_sqrt()
p1 + scale_y_reverse()

# Or you can supply a transformation in the `trans` argument:
p1 + scale_y_continuous(transform = scales::transform_reciprocal())

# You can also create your own. See ?scales::new_transform
```

scale_date *Position scales for date/time data*

Description

These are the default scales for the three date/time class. These will usually be added automatically. To override manually, use `scale_*_date` for dates (class `Date`), `scale_*_datetime` for datetimes (class `POSIXct`), and `scale_*_time` for times (class `hms`).

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(),
  oob = censor,
  guide = waiver(),
  position = "bottom",
  sec.axis = 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(),
  oob = censor,
  guide = waiver(),
  position = "left",
  sec.axis = waiver()
)

scale_x_datetime(
  name = waiver(),
  breaks = waiver(),
  date_breaks = waiver(),
  labels = waiver(),
  date_labels = waiver(),
  minor_breaks = waiver(),
  date_minor_breaks = waiver(),
  timezone = NULL,
  limits = NULL,
  expand = waiver(),
  oob = censor,
  guide = waiver(),
  position = "bottom",
  sec.axis = waiver()
)

scale_y_datetime(
  name = waiver(),
  breaks = waiver(),
  date_breaks = waiver(),
  labels = waiver(),
  date_labels = waiver(),
  minor_breaks = waiver(),
  date_minor_breaks = waiver(),
  timezone = NULL,
  limits = NULL,
  expand = waiver(),
  oob = censor,
  guide = waiver(),
  position = "left",
  sec.axis = waiver()
)

scale_x_time(
  name = waiver(),
  breaks = waiver(),
  date_breaks = waiver(),
  minor_breaks = waiver(),
  date_minor_breaks = waiver(),
  labels = waiver(),
  date_labels = waiver(),
  limits = NULL,
  expand = waiver(),
  oob = censor,
  na.value = NA_real_,
  guide = waiver(),
  position = "bottom",
  sec.axis = waiver()
)

scale_y_time(
  name = waiver(),
  breaks = waiver(),
  date_breaks = waiver(),
  minor_breaks = waiver(),
  date_minor_breaks = waiver(),
  labels = waiver(),
  date_labels = waiver(),
  limits = NULL,
  expand = waiver(),
  oob = censor,
  na.value = NA_real_,
  guide = waiver(),
  position = "left",
  sec.axis = waiver()
)
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`breaks`	One of: • `NULL` for no breaks • `waiver()` for the breaks specified by `date_breaks` • A `Date`/`POSIXct` vector giving positions of breaks • 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. Valid specifications are 'sec', 'min', 'hour', 'day', 'week', 'month' or 'year', optionally followed by 's'.
`labels`	One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches. • `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`) • An expression vector (must be the same length as breaks). See ?plotmath for details. • A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.
`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 breaks • `waiver()` for the breaks specified by `date_minor_breaks` • A `Date`/`POSIXct` vector giving positions of minor breaks • A function that takes the limits as input and returns minor breaks as output

`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. Valid specifications are 'sec', 'min', 'hour', 'day', 'week', 'month' or 'year', optionally followed by 's'.

`limits`
: One of:

  - `NULL` to use the default scale range
  - A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum
  - A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).

`expand`
: For position scales, a vector of range expansion constants used to add some padding around the data to ensure that they are placed some distance away from the axes. Use the convenience function `expansion()` to generate the values for the `expand` argument. The defaults are to expand the scale by 5% on each side for continuous variables, and by 0.6 units on each side for discrete variables.

`oob`
: One of:

  - Function that handles limits outside of the scale limits (out of bounds). Also accepts rlang lambda function notation.
  - The default (`scales::censor()`) replaces out of bounds values with `NA`.
  - `scales::squish()` for squishing out of bounds values into range.
  - `scales::squish_infinite()` for squishing infinite values into range.

`guide`
: A function used to create a guide or its name. See `guides()` for more information.

`position`
: For position scales, The position of the axis. `left` or `right` for y axes, `top` or `bottom` for x axes.

`sec.axis`
: `sec_axis()` is used to specify a secondary axis.

`timezone`
: The timezone to use for display on the axes. The default (`NULL`) uses the timezone encoded in the data.

`na.value`
: Missing values will be replaced with this value.

See Also

`sec_axis()` for how to specify secondary axes.

The date-time position scales section of the online ggplot2 book.

The position documentation.

Other position scales: `scale_x_binned()`, `scale_x_continuous()`, `scale_x_discrete()`

Examples

```
last_month <- Sys.Date() - 0:29
set.seed(1)
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` *Use values without scaling*

Description

Use this set of scales when your data has already been scaled, i.e. it already represents aesthetic values that ggplot2 can handle directly. These scales will not produce a legend unless you also supply the `breaks`, `labels`, and type of `guide` you want.

Usage

```
scale_colour_identity(
  name = waiver(),
  ...,
  guide = "none",
  aesthetics = "colour"
)

scale_fill_identity(name = waiver(), ..., guide = "none", aesthetics = "fill")

scale_shape_identity(
  name = waiver(),
  ...,
  guide = "none",
  aesthetics = "shape"
)
```

```
scale_linetype_identity(
  name = waiver(),
  ...,
  guide = "none",
  aesthetics = "linetype"
)

scale_linewidth_identity(
  name = waiver(),
  ...,
  guide = "none",
  aesthetics = "linewidth"
)

scale_alpha_identity(
  name = waiver(),
  ...,
  guide = "none",
  aesthetics = "alpha"
)

scale_size_identity(name = waiver(), ..., guide = "none", aesthetics = "size")

scale_discrete_identity(aesthetics, name = waiver(), ..., guide = "none")

scale_continuous_identity(aesthetics, name = waiver(), ..., guide = "none")
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Other arguments passed on to `discrete_scale()` or `continuous_scale()`
`guide`	Guide to use for this scale. Defaults to `"none"`.
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the `colour` and `fill` aesthetics at the same time, via `aesthetics = c("colour", "fill")`.

Details

The functions `scale_colour_identity()`, `scale_fill_identity()`, `scale_size_identity()`, etc. work on the aesthetics specified in the scale name: `colour`, `fill`, `size`, etc. However, the functions `scale_colour_identity()` and `scale_fill_identity()` also have an optional `aesthetics` argument that can be used to define both `colour` and `fill` aesthetic mappings via a single function call. The functions `scale_discrete_identity()` and `scale_continuous_identity()` are generic scales that can work with any aesthetic or set of aesthetics provided via the `aesthetics` argument.

See Also

The identity scales section of the online ggplot2 book.

Other shape scales: `scale_shape()`, `scale_shape_manual()`.

Other linetype scales: `scale_linetype()`, `scale_linetype_manual()`.

Other alpha scales: `scale_alpha()`, `scale_alpha_manual()`.

Other size scales: `scale_size()`, `scale_size_manual()`.

Other colour scales: `scale_alpha()`, `scale_colour_brewer()`, `scale_colour_continuous()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_hue()`, `scale_colour_manual()`, `scale_colour_steps()`, `scale_colour_viridis_d()`

Examples

```
ggplot(luv_colours, aes(u, v)) +
  geom_point(aes(colour = col), size = 3) +
  scale_color_identity() +
  coord_fixed()

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. Continuous values can not be mapped to line types unless `scale_linetype_binned()` is used. Still, as linetypes has no inherent order, this use is not advised.

Usage

```
scale_linetype(name = waiver(), ..., aesthetics = "linetype")

scale_linetype_binned(name = waiver(), ..., aesthetics = "linetype")

scale_linetype_continuous(...)

scale_linetype_discrete(name = waiver(), ..., aesthetics = "linetype")
```

Arguments

`name` The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.

`...` Arguments passed on to `discrete_scale`

`breaks` One of:

- `NULL` for no breaks
- `waiver()` for the default breaks (the scale limits)
- A character vector of breaks
- A function that takes the limits as input and returns breaks as output. Also accepts rlang lambda function notation.

`limits` One of:

- `NULL` to use the default scale values
- A character vector that defines possible values of the scale and their order
- A function that accepts the existing (automatic) values and returns new ones. Also accepts rlang lambda function notation.

`drop` Should unused factor levels be omitted from the scale? The default, `TRUE`, uses the levels that appear in the data; `FALSE` includes the levels in the factor. Please note that to display every level in a legend, the layer should use `show.legend = TRUE`.

`na.translate` Unlike continuous scales, discrete scales can easily show missing values, and do so by default. If you want to remove missing values from a discrete scale, specify `na.translate = FALSE`.

`minor_breaks` One of:

- `NULL` for no minor breaks
- `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous)
- A numeric vector of positions
- A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `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`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`guide` A function used to create a guide or its name. See `guides()` for more information.

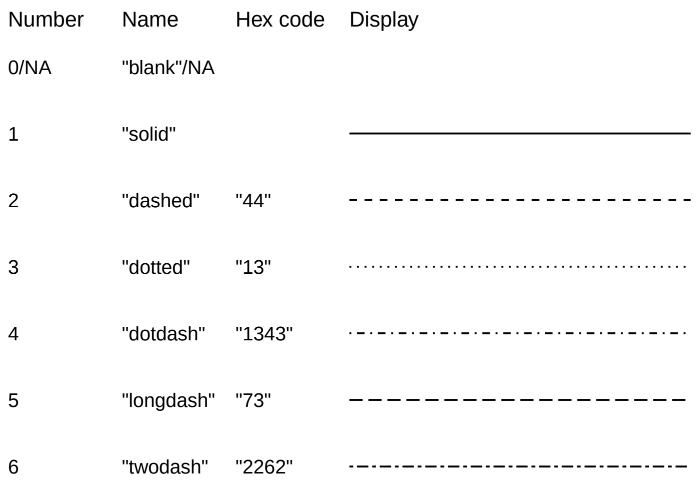
`call` The `call` used to construct the scale for reporting messages.

`super` The super class to use for the constructed scale

`aesthetics` The names of the aesthetics that this scale works with.

Details

Lines can be referred to by number, name or hex code. Contrary to base R graphics, `NA`s are interpreted as blanks.



Number	Name	Hex code	Display
0/NA	"blank"/NA		
1	"solid"		
2	"dashed"	"44"	
3	"dotted"	"13"	
4	"dotdash"	"1343"	
5	"longdash"	"73"	
6	"twodash"	"2262"	

See Also

The documentation for differentiation related aesthetics.

Other linetype scales: `scale_linetype_manual()`, `scale_linetype_identity()`.

The line type section of the online ggplot2 book.

Examples

```
base <- ggplot(economics_long, aes(date, value01))
base + geom_line(aes(group = variable))
base + geom_line(aes(linetype = variable))

# See scale_manual for more flexibility

# Common line types ----------------------------
df_lines <- data.frame(
  linetype = factor(
    1:4,
    labels = c("solid", "longdash", "dashed", "dotted")
  )
)
ggplot(df_lines) +
  geom_hline(aes(linetype = linetype, yintercept = 0), linewidth = 2) +
  scale_linetype_identity() +
  facet_grid(linetype ~ .) +
  theme_void(20)
```

`scale_linewidth` *Scales for line width*

Description

`scale_linewidth` scales the width of lines and polygon strokes. Due to historical reasons, it is also possible to control this with the `size` aesthetic, but using `linewidth` is encouraged to clearly differentiate area aesthetics from stroke width aesthetics.

Usage

```
scale_linewidth(
  name = waiver(),
  breaks = waiver(),
  labels = waiver(),
  limits = NULL,
  range = NULL,
  transform = "identity",
  trans = deprecated(),
  guide = "legend",
  aesthetics = "linewidth"
)

scale_linewidth_binned(
  name = waiver(),
  breaks = waiver(),
  labels = waiver(),
  limits = NULL,
  range = NULL,
  n.breaks = NULL,
  nice.breaks = TRUE,
  transform = "identity",
  trans = deprecated(),
  guide = "bins",
  aesthetics = "linewidth"
)
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`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 (e.g., a function returned by `scales::extended_breaks()`). Note that for position scales, limits are provided after scale expansion. Also accepts rlang lambda function notation.
`labels`	One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches. • `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`) • An expression vector (must be the same length as breaks). See ?plotmath for details. • A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.
`limits`	One of: • `NULL` to use the default scale range • A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum • A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).

`range`	a numeric vector of length 2 that specifies the minimum and maximum size of the plotting symbol after transformation.
`transform`	For continuous scales, the name of a transformation object or the object itself. Built-in transformations include "asn", "atanh", "boxcox", "date", "exp", "hms", "identity", "log", "log10", "log1p", "log2", "logit", "modulus", "probability", "probit", "pseudo_log", "reciprocal", "reverse", "sqrt" and "time". 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 `transform_<name>`. If transformations require arguments, you can call them from the scales package, e.g. `scales::transform_boxcox(p = 2)`. You can create your own transformation with `scales::new_transform()`.
`trans`	**[Deprecated]** Deprecated in favour of `transform`.
`guide`	A function used to create a guide or its name. See `guides()` for more information.
`aesthetics`	The names of the aesthetics that this scale works with.
`n.breaks`	An integer guiding the number of major breaks. The algorithm may choose a slightly different number to ensure nice break labels. Will only have an effect if `breaks = waiver()`. Use `NULL` to use the default number of breaks given by the transformation.
`nice.breaks`	Logical. Should breaks be attempted placed at nice values instead of exactly evenly spaced between the limits. If `TRUE` (default) the scale will ask the transformation object to create breaks, and this may result in a different number of breaks than requested. Ignored if breaks are given explicitly.

See Also

The documentation for differentiation related aesthetics.

The line width section of the online ggplot2 book.

Examples

```
p <- ggplot(economics, aes(date, unemploy, linewidth = uempmed)) +
  geom_line(lineend = "round")
p
p + scale_linewidth("Duration of\nunemployment")
p + scale_linewidth(range = c(0, 4))

# Binning can sometimes make it easier to match the scaled data to the legend
p + scale_linewidth_binned()
```

scale_manual *Create your own discrete scale*

Description

These functions allow you to specify your own set of mappings from levels in the data to aesthetic values.

Usage

```
scale_colour_manual(
  ...,
  values,
  aesthetics = "colour",
  breaks = waiver(),
  na.value = "grey50"
)

scale_fill_manual(
  ...,
  values,
  aesthetics = "fill",
  breaks = waiver(),
  na.value = "grey50"
)

scale_size_manual(
  ...,
  values,
  breaks = waiver(),
  na.value = NA,
  aesthetics = "size"
)

scale_shape_manual(
  ...,
  values,
  breaks = waiver(),
  na.value = NA,
  aesthetics = "shape"
)

scale_linetype_manual(
  ...,
  values,
  breaks = waiver(),
  na.value = NA,
  aesthetics = "linetype"
)

scale_linewidth_manual(
  ...,
  values,
  breaks = waiver(),
  na.value = NA,
  aesthetics = "linewidth"
)

scale_alpha_manual(
  ...,
  values,
  breaks = waiver(),
  na.value = NA,
  aesthetics = "alpha"
)

scale_discrete_manual(aesthetics, ..., values, breaks = waiver())
```

Arguments

`...` Arguments passed on to `discrete_scale`

`limits` One of:
- `NULL` to use the default scale values
- A character vector that defines possible values of the scale and their order
- A function that accepts the existing (automatic) values and returns new ones. Also accepts rlang lambda function notation.

`drop` Should unused factor levels be omitted from the scale? The default, `TRUE`, uses the levels that appear in the data; `FALSE` includes the levels in the factor. Please note that to display every level in a legend, the layer should use `show.legend = TRUE`.

`na.translate` Unlike continuous scales, discrete scales can easily show missing values, and do so by default. If you want to remove missing values from a discrete scale, specify `na.translate = FALSE`.

`name` The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.

`minor_breaks` One of:
- `NULL` for no minor breaks
- `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous)
- A numeric vector of positions
- A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `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`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`guide` A function used to create a guide or its name. See `guides()` for more information.

`call` The `call` used to construct the scale for reporting messages.

`super` The super class to use for the constructed scale

`values`	a set of aesthetic values to map data values to. The values will be matched in order (usually alphabetical) with the limits of the scale, or with `breaks` if provided. If this is a named vector, then the values will be matched based on the names instead. Data values that don't match will be given `na.value`.
`aesthetics`	Character string or vector of character strings listing the name(s) of the aesthetic(s) that this scale works with. This can be useful, for example, to apply colour settings to the `colour` and `fill` aesthetics at the same time, via `aesthetics = c("colour", "fill")`.
`breaks`	One of: • `NULL` for no breaks • `waiver()` for the default breaks (the scale limits) • A character vector of breaks • A function that takes the limits as input and returns breaks as output
`na.value`	The aesthetic value to use for missing (`NA`) values

Details

The functions `scale_colour_manual()`, `scale_fill_manual()`, `scale_size_manual()`, etc. work on the aesthetics specified in the scale name: `colour`, `fill`, `size`, etc. However, the functions `scale_colour_manual()` and `scale_fill_manual()` also have an optional `aesthetics` argument that can be used to define both `colour` and `fill` aesthetic mappings via a single function call (see examples). The function `scale_discrete_manual()` is a generic scale that can work with any aesthetic or set of aesthetics provided via the `aesthetics` argument.

Color Blindness

Many color palettes derived from RGB combinations (like the "rainbow" color palette) are not suitable to support all viewers, especially those with color vision deficiencies. Using `viridis` type, which is perceptually uniform in both colour and black-and-white display is an easy option to ensure good perceptive properties of your visualizations. The colorspace package offers functionalities

- to generate color palettes with good perceptive properties,

- to analyse a given color palette, like emulating color blindness,
- and to modify a given color palette for better perceptivity.

For more information on color vision deficiencies and suitable color choices see the paper on the colorspace package and references therein.

See Also

The documentation for differentiation related aesthetics.

The documentation on colour aesthetics.

The manual scales and manual colour scales sections of the online ggplot2 book.

Other size scales: `scale_size()`, `scale_size_identity()`.

Other shape scales: `scale_shape()`, `scale_shape_identity()`.

Other linetype scales: `scale_linetype()`, `scale_linetype_identity()`.

Other alpha scales: `scale_alpha()`, `scale_alpha_identity()`.

Other colour scales: `scale_alpha()`, `scale_colour_brewer()`, `scale_colour_continuous()`, `scale_colour_gradient()`, `scale_colour_grey()`, `scale_colour_hue()`, `scale_colour_identity()`, `scale_colour_steps()`, `scale_colour_viridis_d()`

Examples

```
p <- ggplot(mtcars, aes(mpg, wt)) +
  geom_point(aes(colour = factor(cyl)))
p + scale_colour_manual(values = c("red", "blue", "green"))

# It's recommended to use a named vector
cols <- c("8" = "red", "4" = "blue", "6" = "darkgreen", "10" = "orange")
p + scale_colour_manual(values = cols)

# You can set color and fill aesthetics at the same time
ggplot(
  mtcars,
  aes(mpg, wt, colour = factor(cyl), fill = factor(cyl))
) +
  geom_point(shape = 21, alpha = 0.5, size = 2) +
  scale_colour_manual(
    values = cols,
    aesthetics = c("colour", "fill")
  )

# As with other scales you can use breaks to control the appearance
# of the legend.
p + scale_colour_manual(values = cols)
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"))
```

`scale_shape` *Scales for shapes, aka glyphs*

Description

`scale_shape()` maps discrete variables to six easily discernible shapes. If you have more than six levels, you will get a warning message, and the seventh and subsequent levels will not appear on the plot. Use `scale_shape_manual()` to supply your own values. You can not map a continuous variable to shape unless `scale_shape_binned()` is used. Still, as shape has no inherent order, this use is not advised.

Usage

```
scale_shape(name = waiver(), ..., solid = NULL, aesthetics = "shape")

scale_shape_binned(name = waiver(), ..., solid = TRUE, aesthetics = "shape")
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`...`	Arguments passed on to `discrete_scale`

`breaks` One of:
- `NULL` for no breaks
- `waiver()` for the default breaks (the scale limits)
- A character vector of breaks
- A function that takes the limits as input and returns breaks as output. Also accepts rlang lambda function notation.

`limits` One of:
- `NULL` to use the default scale values
- A character vector that defines possible values of the scale and their order
- A function that accepts the existing (automatic) values and returns new ones. Also accepts rlang lambda function notation.

`drop` Should unused factor levels be omitted from the scale? The default, `TRUE`, uses the levels that appear in the data; `FALSE` includes the levels in the factor. Please note that to display every level in a legend, the layer should use `show.legend = TRUE`.

`na.translate` Unlike continuous scales, discrete scales can easily show missing values, and do so by default. If you want to remove missing values from a discrete scale, specify `na.translate = FALSE`.

`na.value` If `na.translate = TRUE`, what aesthetic value should the missing values be displayed as? Does not apply to position scales where `NA` is always placed at the far right.

`minor_breaks` One of:

- `NULL` for no minor breaks
- `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous)
- A numeric vector of positions
- A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `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`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`guide` A function used to create a guide or its name. See `guides()` for more information.

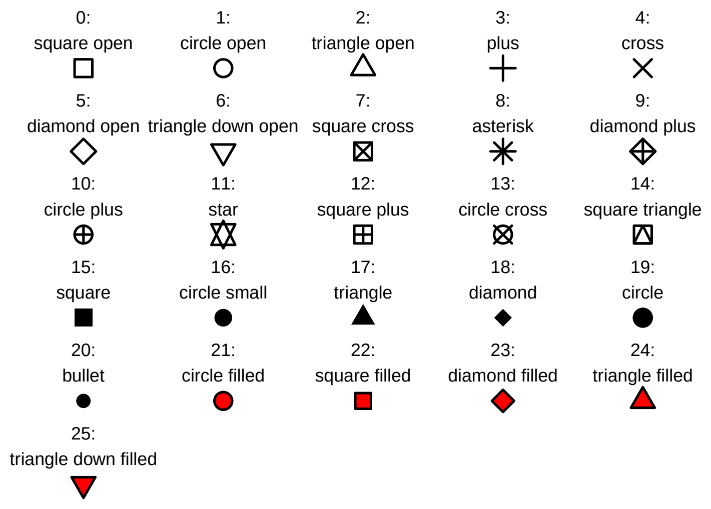
`call` The `call` used to construct the scale for reporting messages.

`super` The super class to use for the constructed scale

`solid`	Should the shapes be solid, `TRUE`, or hollow, `FALSE`?
`aesthetics`	The names of the aesthetics that this scale works with.

Details

Shapes can be referred to by number or name. Shapes in [0, 20] do not support a fill aesthetic, whereas shapes in [21, 25] do.



See Also

The documentation for differentiation related aesthetics.

Other shape scales: `scale_shape_manual()`, `scale_shape_identity()`.

The shape section of the online ggplot2 book.

Examples

```
set.seed(596)
dsmall <- diamonds[sample(nrow(diamonds), 100), ]

(d <- 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")

# To change order of levels, change order of
# underlying factor
levels(dsmall$cut) <- 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))

# Show a list of available shapes
df_shapes <- data.frame(shape = 0:24)
ggplot(df_shapes, aes(0, 0, shape = shape)) +
  geom_point(aes(shape = shape), size = 5, fill = 'red') +
  scale_shape_identity() +
  facet_wrap(~shape) +
  theme_void()
```

`scale_size` *Scales for 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. `scale_size_binned()` is a binned version of `scale_size()` that scales by area (but does not ensure 0 equals an area of zero). For a binned equivalent of `scale_size_area()` use `scale_size_binned_area()`.

Usage

```
scale_size(
  name = waiver(),
  breaks = waiver(),
  labels = waiver(),
  limits = NULL,
  range = NULL,
  transform = "identity",
  trans = deprecated(),
  guide = "legend",
  aesthetics = "size"
)

scale_radius(
  name = waiver(),
  breaks = waiver(),
  labels = waiver(),
  limits = NULL,
  range = c(1, 6),
  transform = "identity",
  trans = deprecated(),
  guide = "legend",
  aesthetics = "size"
)

scale_size_binned(
  name = waiver(),
  breaks = waiver(),
  labels = waiver(),
  limits = NULL,
  range = NULL,
  n.breaks = NULL,
  nice.breaks = TRUE,
  transform = "identity",
  trans = deprecated(),
  guide = "bins",
  aesthetics = "size"
)

scale_size_area(name = waiver(), ..., max_size = 6, aesthetics = "size")

scale_size_binned_area(name = waiver(), ..., max_size = 6, aesthetics = "size")
```

Arguments

`name`	The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.
`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 (e.g., a function returned by `scales::extended_breaks()`). Note that for position scales, limits are provided after scale expansion. Also accepts rlang lambda function notation.
`labels`	One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches. • `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`) • An expression vector (must be the same length as breaks). See ?plotmath for details. • A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.
`limits`	One of: • `NULL` to use the default scale range • A numeric vector of length two providing limits of the scale. Use `NA` to refer to the existing minimum or maximum • A function that accepts the existing (automatic) limits and returns new limits. Also accepts rlang lambda function notation. Note that setting limits on positional scales will **remove** data outside of the limits. If the purpose is to zoom, use the limit argument in the coordinate system (see `coord_cartesian()`).
`range`	a numeric vector of length 2 that specifies the minimum and maximum size of the plotting symbol after transformation.
`transform`	For continuous scales, the name of a transformation object or the object itself. Built-in transformations include "asn", "atanh", "boxcox", "date", "exp", "hms",

	"identity", "log", "log10", "log1p", "log2", "logit", "modulus", "probability", "probit", "pseudo_log", "reciprocal", "reverse", "sqrt" and "time". 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 `transform_<name>`. If transformations require arguments, you can call them from the scales package, e.g. `scales::transform_boxcox(p = 2)`. You can create your own transformation with `scales::new_transform()`.
`trans`	**[Deprecated]** Deprecated in favour of `transform`.
`guide`	A function used to create a guide or its name. See `guides()` for more information.
`aesthetics`	The names of the aesthetics that this scale works with.
`n.breaks`	An integer guiding the number of major breaks. The algorithm may choose a slightly different number to ensure nice break labels. Will only have an effect if `breaks = waiver()`. Use `NULL` to use the default number of breaks given by the transformation.
`nice.breaks`	Logical. Should breaks be attempted placed at nice values instead of exactly evenly spaced between the limits. If `TRUE` (default) the scale will ask the transformation object to create breaks, and this may result in a different number of breaks than requested. Ignored if breaks are given explicitly.
`...`	Arguments passed on to `continuous_scale` `minor_breaks` One of: • `NULL` for no minor breaks • `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous) • A numeric vector of positions • A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions. `oob` One of: • Function that handles limits outside of the scale limits (out of bounds). Also accepts rlang lambda function notation. • The default (`scales::censor()`) replaces out of bounds values with `NA`. • `scales::squish()` for squishing out of bounds values into range. • `scales::squish_infinite()` for squishing infinite values into range. `na.value` Missing values will be replaced with this value. `call` The `call` used to construct the scale for reporting messages. `super` The super class to use for the constructed scale
`max_size`	Size of largest points.

Note

Historically the size aesthetic was used for two different things: Scaling the size of object (like points and glyphs) and scaling the width of lines. From ggplot2 3.4.0 the latter has been moved to its own linewidth aesthetic. For backwards compatibility using size is still possible, but it is highly advised to switch to the new linewidth aesthetic for these cases.

See Also

`scale_size_area()` if you want 0 values to be mapped to points with size 0. `scale_linewidth()` if you want to scale the width of lines.

The documentation for differentiation related aesthetics.

The size section of the online ggplot2 book.

Examples

```
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()

# Binning can sometimes make it easier to match the scaled data to the legend
p + scale_size_binned()

# 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` *Position scales for discrete data*

Description

`scale_x_discrete()` and `scale_y_discrete()` are used to set the values for discrete x and y scale aesthetics. For simple manipulation of scale labels and limits, you may wish to use `labs()` and `lims()` instead.

Usage

```
scale_x_discrete(
  name = waiver(),
  ...,
  palette = seq_len,
  expand = waiver(),
  guide = waiver(),
  position = "bottom",
  sec.axis = waiver(),
  continuous.limits = NULL
)

scale_y_discrete(
  name = waiver(),
  ...,
  palette = seq_len,
  expand = waiver(),
  guide = waiver(),
  position = "left",
  sec.axis = waiver(),
  continuous.limits = NULL
)
```

Arguments

`name` The name of the scale. Used as the axis or legend title. If `waiver()`, the default, the name of the scale is taken from the first mapping used for that aesthetic. If `NULL`, the legend title will be omitted.

`...` Arguments passed on to `discrete_scale`

`breaks` One of:

- `NULL` for no breaks
- `waiver()` for the default breaks (the scale limits)
- A character vector of breaks
- A function that takes the limits as input and returns breaks as output. Also accepts rlang lambda function notation.

`limits` One of:

- `NULL` to use the default scale values
- A character vector that defines possible values of the scale and their order
- A function that accepts the existing (automatic) values and returns new ones. Also accepts rlang lambda function notation.

`drop` Should unused factor levels be omitted from the scale? The default, `TRUE`, uses the levels that appear in the data; `FALSE` includes the levels in the factor. Please note that to display every level in a legend, the layer should use `show.legend = TRUE`.

`na.translate` Unlike continuous scales, discrete scales can easily show missing values, and do so by default. If you want to remove missing values from a discrete scale, specify `na.translate = FALSE`.

`na.value` If `na.translate = TRUE`, what aesthetic value should the missing values be displayed as? Does not apply to position scales where `NA` is always placed at the far right.

`aesthetics` The names of the aesthetics that this scale works with.

`minor_breaks` One of:

- `NULL` for no minor breaks

- `waiver()` for the default breaks (none for discrete, one minor break between each major break for continuous)
- A numeric vector of positions
- A function that given the limits returns a vector of minor breaks. Also accepts rlang lambda function notation. When the function has two arguments, it will be given the limits and major break positions.

`labels` One of the options below. Please note that when `labels` is a vector, it is highly recommended to also set the `breaks` argument as a vector to protect against unintended mismatches.

- `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`)
- An expression vector (must be the same length as breaks). See ?plotmath for details.
- A function that takes the breaks as input and returns labels as output. Also accepts rlang lambda function notation.

`call` The `call` used to construct the scale for reporting messages.

`super` The super class to use for the constructed scale

`palette` A palette function that when called with a single integer argument (the number of levels in the scale) returns the numerical values that they should take.

`expand` For position scales, a vector of range expansion constants used to add some padding around the data to ensure that they are placed some distance away from the axes. Use the convenience function `expansion()` to generate the values for the `expand` argument. The defaults are to expand the scale by 5% on each side for continuous variables, and by 0.6 units on each side for discrete variables.

`guide` A function used to create a guide or its name. See `guides()` for more information.

`position` For position scales, The position of the axis. `left` or `right` for y axes, `top` or `bottom` for x axes.

`sec.axis` `dup_axis()` is used to specify a secondary axis.

`continuous.limits`
One of:

- `NULL` to use the default scale range
- A numeric vector of length two providing a display range for the scale. Use `NA` to refer to the existing minimum or maximum.
- A function that accepts the limits and returns a numeric vector of length two.

Details

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.

See Also

The position documentation.

The discrete position scales section of the online ggplot2 book.

Other position scales: `scale_x_binned()`, `scale_x_continuous()`, `scale_x_date()`

Examples

```
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"
    )
  )

# 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")
d + ylim("I1", "IF")

# See ?reorder to reorder based on the values of another variable
ggplot(mpg, aes(manufacturer, cty)) +
  geom_point()
ggplot(mpg, aes(reorder(manufacturer, cty), cty)) +
  geom_point()
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

This vector field was produced from the data described in Brillinger, D.R., Preisler, H.K., Ager, A.A. and Kie, J.G. "An exploratory data analysis (EDA) of the paths of moving animals". J. Statistical Planning and Inference 122 (2004), 43-63, using the methods of Brillinger, D.R., "Learning a potential function from a trajectory", Signal Processing Letters. December (2007).

Usage

```
seals
```

Format

A data frame with 1155 rows and 4 variables

References

https://www.stat.berkeley.edu/~brill/Papers/jspifinal.pdf

`sec_axis` *Specify a secondary axis*

Description

This function is used in conjunction with a position scale to create a secondary axis, positioned opposite of the primary axis. All secondary axes must be based on a one-to-one transformation of the primary axes.

Usage

```
sec_axis(
  transform = NULL,
  name = waiver(),
  breaks = waiver(),
  labels = waiver(),
  guide = waiver(),
  trans = deprecated()
)

dup_axis(
  transform = identity,
  name = derive(),
  breaks = derive(),
  labels = derive(),
  guide = derive(),
  trans = deprecated()
)

derive()
```

Arguments

`transform`	A formula or function of a strictly monotonic transformation
`name`	The name of the secondary axis
`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
`guide`	A position guide that will be used to render the axis on the plot. Usually this is `guide_axis()`.
`trans`	**[Deprecated]**

Details

`sec_axis()` is used to create the specifications for a secondary axis. Except for the `trans` argument any of the arguments can be set to `derive()` which would result in the secondary axis inheriting the settings from the primary axis.

`dup_axis()` is provide as a shorthand for creating a secondary axis that is a duplication of the primary axis, effectively mirroring the primary axis.

As of v3.1, date and datetime scales have limited secondary axis capabilities. Unlike other continuous scales, secondary axis transformations for date and datetime scales must respect their primary POSIX data structure. This means they may only be transformed via addition or subtraction, e.g. `~ . + hms::hms(days = 8)`, or `~ . - 8*60*60`. Nonlinear transformations will return an error. To produce a time-since-event secondary axis in this context, users may consider adapting secondary axis labels.

Examples

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

# Create a simple secondary axis
p + scale_y_continuous(sec.axis = sec_axis(~ . + 10))
```

```
# Inherit the name from the primary axis
p + scale_y_continuous("Miles/gallon", sec.axis = sec_axis(~ . + 10, name = derive()))

# Duplicate the primary axis
p + scale_y_continuous(sec.axis = dup_axis())

# You can pass in a formula as a shorthand
p + scale_y_continuous(sec.axis = ~ .^2)

# Secondary axes work for date and datetime scales too:
df <- data.frame(
  dx = seq(
    as.POSIXct("2012-02-29 12:00:00", tz = "UTC"),
    length.out = 10,
    by = "4 hour"
  ),
  price = seq(20, 200000, length.out = 10)
 )

# This may useful for labelling different time scales in the same plot
ggplot(df, aes(x = dx, y = price)) +
  geom_line() +
  scale_x_datetime(
    "Date",
    date_labels = "%b %d",
    date_breaks = "6 hour",
    sec.axis = dup_axis(
      name = "Time of Day",
      labels = scales::label_time("%I %p")
    )
  )

# or to transform axes for different timezones
ggplot(df, aes(x = dx, y = price)) +
  geom_line() +
  scale_x_datetime("
    GMT",
    date_labels = "%b %d %I %p",
    sec.axis = sec_axis(
      ~ . + 8 * 3600,
      name = "GMT+8",
      labels = scales::label_time("%b %d %I %p")
    )
  )
```

`stat_connect` *Connect observations*

Description

Connect successive points with lines of different shapes.

Usage

```
stat_connect(
  mapping = NULL,
  data = NULL,
  geom = "path",
  position = "identity",
  ...,
  connection = "hv",
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`. • A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`. • For more information and other ways to specify the geom, see the layer geom documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position.

- A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`.
- For more information and other ways to specify the position, see the layer position documentation.

`...`
: Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

  - Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
  - When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
  - Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
  - The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`connection`
: A specification of how two points are connected. Can be one of the folloing:

  - A string giving a named connection. These options are:
    - `"hv"` to first jump horizontally, then vertically.
    - `"vh"` to first jump vertically, then horizontally.
    - `"mid"` to step half-way between adjacent x-values.
    - `"linear"` to use a straight segment.
  - A numeric matrix with two columns giving x and y coordinates respectively. The coordinates should describe points on a path that connect point A at location (0, 0) and point B at location (1, 1). At least one of these two points is expected to be included in the coordinates.

`na.rm`
: If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`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. `annotation_borders()`.

Aesthetics

`stat_connect()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x` ***or*** `xmin` ***or*** `xmax`
- `y` ***or*** `ymin` ***or*** `ymax`
- `group` → inferred

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
ggplot(head(economics, 20), aes(date, unemploy)) +
  stat_connect(connection = "hv")

# Setup custom connections
x <- seq(0, 1, length.out = 20)[-1]
smooth <- cbind(x, scales::rescale(1 / (1 + exp(-(x * 10 - 5)))))
zigzag <- cbind(c(0.4, 0.6, 1), c(0.75, 0.25, 1))

ggplot(head(economics, 10), aes(date, unemploy)) +
  geom_point() +
  stat_connect(aes(colour = "zigzag"), connection = zigzag) +
  stat_connect(aes(colour = "smooth"), connection = smooth)
```

`stat_ecdf` *Compute empirical cumulative distribution*

Description

The empirical cumulative distribution function (ECDF) provides an alternative visualisation of distribution. Compared to other visualisations that rely on density (like `geom_histogram()`), the ECDF doesn't require any tuning parameters and handles both continuous and categorical variables. The downside is that it requires more training to accurately interpret, and the underlying visual tasks are somewhat more challenging.

Usage

```
stat_ecdf(
  mapping = NULL,
  data = NULL,
  geom = "step",
  position = "identity",
  ...,
  n = NULL,
  pad = TRUE,
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`. • A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`. • For more information and other ways to specify the geom, see the layer geom documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

`n`	if NULL, do not interpolate. If not NULL, this is the number of points to interpolate with.
`pad`	If `TRUE`, pad the ecdf with additional points (-Inf, 0) and (Inf, 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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Details

The statistic relies on the aesthetics assignment to guess which variable to use as the input and which to use as the output. Either x or y must be provided and one of them must be unused. The ECDF will be calculated on the given aesthetic and will be output on the unused one.

If the `weight` aesthetic is provided, a weighted ECDF will be computed. In this case, the ECDF is incremented by `weight / sum(weight)` instead of `1 / length(x)` for each observation.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(ecdf)`
  Cumulative density corresponding to `x`.
- `after_stat(y)`
  [Superseded] For backward compatibility.

Dropped variables

weight After calculation, weights of individual observations (if supplied), are no longer available.

Aesthetics

`stat_ecdf()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

•	`x` ***or*** `y`	
•	`group`	→ inferred
•	`weight`	→ `NULL`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
set.seed(1)
df <- data.frame(
  x = c(rnorm(100, 0, 3), rnorm(100, 0, 10)),
  g = gl(2, 100)
)
ggplot(df, aes(x)) +
  stat_ecdf(geom = "step")

# Don't go to positive/negative infinity
ggplot(df, aes(x)) +
  stat_ecdf(geom = "step", pad = FALSE)

# Multiple ECDFs
ggplot(df, aes(x, colour = g)) +
  stat_ecdf()

# Using weighted eCDF
weighted <- data.frame(x = 1:10, weights = c(1:5, 5:1))
plain <- data.frame(x = rep(weighted$x, weighted$weights))

ggplot(plain, aes(x)) +
  stat_ecdf(linewidth = 1) +
  stat_ecdf(
    aes(weight = weights),
    data = weighted, colour = "green"
  )
```

stat_ellipse *Compute normal data ellipses*

Description

The method for calculating the ellipses has been modified from `car::dataEllipse` (Fox and Weisberg 2011, Friendly and Monette 2013)

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`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`. • A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`.

- For more information and other ways to specify the geom, see the layer geom documentation.

`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`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 `coord_fixed()` is applied.
`level`	The level at which to draw an ellipse, or, if `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 `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Aesthetics

`stat_ellipse()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`
- `y`
- `group` → inferred
- `weight`

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

References

John Fox and Sanford Weisberg (2011). An R Companion to Applied Regression, Second Edition. Thousand Oaks CA: Sage. URL: `https://uk.sagepub.com/en-gb/eur/an-r-companion-to-applied-regression/book246125`

Michael Friendly. Georges Monette. John Fox. "Elliptical Insights: Understanding Statistical Methods through Elliptical Geometry." Statist. Sci. 28 (1) 1 - 39, February 2013. URL: `https://projecteuclid.org/journals/statistical-science/volume-28/issue-1/Elliptical-Insights-Understanding-10.1214/12-STS402.full`

Examples

```
ggplot(faithful, aes(waiting, eruptions)) +
  geom_point() +
  stat_ellipse()

ggplot(faithful, aes(waiting, eruptions, color = eruptions > 3)) +
  geom_point() +
  stat_ellipse()

ggplot(faithful, aes(waiting, eruptions, color = eruptions > 3)) +
  geom_point() +
  stat_ellipse(type = "norm", linetype = 2) +
  stat_ellipse(type = "t")

ggplot(faithful, aes(waiting, eruptions, color = eruptions > 3)) +
  geom_point() +
  stat_ellipse(type = "norm", linetype = 2) +
    stat_ellipse(type = "euclid", level = 3) +
    coord_fixed()

  ggplot(faithful, aes(waiting, eruptions, fill = eruptions > 3)) +
    stat_ellipse(geom = "polygon")
```

`stat_identity` *Leave data as is*

Description

The identity statistic leaves the data unchanged.

Usage

```
stat_identity(
  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()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`.

- A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`.
- For more information and other ways to specify the geom, see the layer geom documentation.

`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Examples

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

`stat_manual` *Manually compute transformations*

Description

`stat_manual()` takes a function that computes a data transformation for every group.

Usage

```
stat_manual(
  mapping = NULL,
  data = NULL,
  geom = "point",
  position = "identity",
  ...,
  fun = identity,
  args = list(),
  na.rm = FALSE,
  show.legend = NA,
  inherit.aes = TRUE
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`.

- A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`.
- For more information and other ways to specify the geom, see the layer geom documentation.

`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`fun`	Function that takes a data frame as input and returns a data frame or data frame-like list as output. The default (`identity()`) returns the data unchanged.
`args`	A list of arguments to pass to the function given in `fun`.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To

include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.

`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. `annotation_borders()`.

Aesthetics

Input aesthetics are determined by the `fun` argument. Output aesthetics must include those required by `geom`. Any aesthetic that is constant within a group will be preserved even if dropped by `fun`.

Aesthetics

`stat_manual()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `group` $\rightarrow$ inferred

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
# A standard scatterplot
p <- ggplot(mtcars, aes(disp, mpg, colour = factor(cyl))) +
  geom_point()

# The default just displays points as-is
p + stat_manual()

# Using a custom function
make_hull <- function(data) {
  hull <- chull(x = data$x, y = data$y)
  data.frame(x = data$x[hull], y = data$y[hull])
}

p + stat_manual(
  geom = "polygon",
  fun  = make_hull,
  fill = NA
)

# Using the `with` function with quoting
p + stat_manual(
  fun  = with,
  args = list(expr = quote({
    hull <- chull(x, y)
    list(x = x[hull], y = y[hull])
  })),
  geom = "polygon", fill = NA
)
```

```
# Using the `transform` function with quoting
p + stat_manual(
  geom = "segment",
  fun  = transform,
  args = list(
    xend = quote(mean(x)),
    yend = quote(mean(y))
  )
)

# Using dplyr verbs with `vars()`
if (requireNamespace("dplyr", quietly = TRUE)) {

  # Get centroids with `summarise()`
  p + stat_manual(
    size = 10, shape = 21,
    fun  = dplyr::summarise,
    args = vars(x = mean(x), y = mean(y))
  )

  # Connect to centroid with `mutate`
  p + stat_manual(
    geom = "segment",
    fun  = dplyr::mutate,
    args = vars(xend = mean(x), yend = mean(y))
  )

  # Computing hull with `reframe()`
  p + stat_manual(
    geom = "polygon", fill = NA,
    fun  = dplyr::reframe,
    args = vars(hull = chull(x, y), x = x[hull], y = y[hull])
  )
}
```

`stat_sf_coordinates` *Extract coordinates from 'sf' objects*

Description

`stat_sf_coordinates()` extracts the coordinates from 'sf' objects and summarises them to one pair of coordinates (x and y) per geometry. This is convenient when you draw an sf object as geoms like text and labels (so `geom_sf_text()` and `geom_sf_label()` relies on this).

Usage

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

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`. • A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`. • For more information and other ways to specify the geom, see the layer geom documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.

`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`fun.geometry`	A function that takes a `sfc` object and returns a `sfc_POINT` with the same length as the input. If `NULL`, `function(x) sf::st_point_on_surface(sf::st_zm(x))` will be used. Note that the function may warn about the incorrectness of the result if the data is not projected, but you can ignore this except when you really care about the exact locations.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept.
- The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

Details

coordinates of an `sf` object can be retrieved by `sf::st_coordinates()`. But, we cannot simply use `sf::st_coordinates()` because, whereas text and labels require exactly one coordinate per geometry, it returns multiple ones for a polygon or a line. Thus, these two steps are needed:

1. Choose one point per geometry by some function like `sf::st_centroid()` or `sf::st_point_on_surface()`.
2. Retrieve coordinates from the points by `sf::st_coordinates()`.

For the first step, you can use an arbitrary function via `fun.geometry`. By default, `function(x) sf::st_point_on_surface(sf::st_zm(x))` is used; `sf::st_point_on_surface()` seems more appropriate than `sf::st_centroid()` since labels and text usually are intended to be put within the polygon or the line. `sf::st_zm()` is needed to drop Z and M dimension beforehand, otherwise `sf::st_point_on_surface()` may fail when the geometries have M dimension.

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(x)`
  X dimension of the simple feature.
- `after_stat(y)`
  Y dimension of the simple feature.

Examples

```
if (requireNamespace("sf", quietly = TRUE)) {
nc <- sf::st_read(system.file("shape/nc.shp", package="sf"))

ggplot(nc) +
  stat_sf_coordinates()

ggplot(nc) +
  geom_errorbarh(
    aes(geometry = geometry,
        xmin = after_stat(x) - 0.1,
        xmax = after_stat(x) + 0.1,
        y = after_stat(y),
        height = 0.04),
    stat = "sf_coordinates"
  )
}
```

`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 is are summarised with `fun`.

Usage

```
stat_summary_2d(
  mapping = NULL,
  data = NULL,
  geom = "tile",
  position = "identity",
    ...,
    binwidth = NULL,
    bins = 30,
    breaks = NULL,
    drop = TRUE,
    fun = "mean",
    fun.args = list(),
    boundary = 0,
    closed = NULL,
    center = NULL,
    na.rm = FALSE,
    show.legend = NA,
    inherit.aes = TRUE
  )

  stat_summary_hex(
    mapping = NULL,
    data = NULL,
    geom = "hex",
    position = "identity",
    ...,
    binwidth = NULL,
    bins = 30,
    drop = TRUE,
    fun = "mean",
    fun.args = list(),
    na.rm = FALSE,
    show.legend = NA,
    inherit.aes = TRUE
  )
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to over-

ride the default coupling between stats and geoms. The geom argument accepts the following:

- A Geom ggproto subclass, for example GeomPoint.
- A string naming the geom. To give the geom as a string, strip the function name of the geom_ prefix. For example, to use geom_point(), give the geom as "point".
- For more information and other ways to specify the geom, see the layer geom documentation.

position — A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The position argument accepts the following:

- The result of calling a position function, such as position_jitter(). This method allows for passing extra arguments to the position.
- A string naming the position adjustment. To give the position as a string, strip the function name of the position_ prefix. For example, to use position_jitter(), give the position as "jitter".
- For more information and other ways to specify the position, see the layer position documentation.

... — Other arguments passed on to layer()'s params argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the position argument, or aesthetics that are required can *not* be passed through .... Unknown arguments that are not part of the 4 categories below are ignored.

- Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, colour = "red" or linewidth = 3. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the params. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data.
- When constructing a layer using a stat_*() function, the ... argument can be used to pass on parameters to the geom part of the layer. An example of this is stat_density(geom = "area", outline.type = "both"). The geom's documentation lists which parameters it can accept.
- Inversely, when constructing a layer using a geom_*() function, the ... argument can be used to pass on parameters to the stat part of the layer. An example of this is geom_area(stat = "density", adjust = 0.5). The stat's documentation lists which parameters it can accept.
- The key_glyph argument of layer() may also be passed on through .... This can be one of the functions described as key glyphs, to change the display of the layer in the legend.

binwidth — The width of the bins. Can be specified as a numeric value or as a function that takes x after scale transformation as input and returns a single numeric value. When specifying a function along with a grouping structure, the function will be called once per group. The default is to use the number of bins in bins, covering the range of the data. You should always override this value, exploring multiple widths to find the best to illustrate the stories in your data.

	The bin width of a date variable is the number of days in each time; the bin width of a time variable is the number of seconds.
`bins`	Number of bins. Overridden by `binwidth`. Defaults to 30.
`breaks`	Alternatively, you can supply a numeric vector giving the bin boundaries. Overrides `binwidth`, `bins`, `center`, and `boundary`. Can also be a function that takes group-wise values as input and returns bin boundaries.
`drop`	drop if the output of `fun` is `NA`.
`fun`	function for summary.
`fun.args`	A list of extra arguments to pass to `fun`
`closed`	One of `"right"` or `"left"` indicating whether right or left edges of bins are included in the bin.
`center`, `boundary`	bin position specifiers. Only one, `center` or `boundary`, may be specified for a single plot. `center` specifies the center of one of the bins. `boundary` specifies the boundary between two bins. Note that if either is above or below the range of the data, things will be shifted by the appropriate integer multiple of `binwidth`. For example, to center on integers use `binwidth = 1` and `center = 0`, even if `0` is outside the range of the data. Alternatively, this same alignment can be specified with `binwidth = 1` and `boundary = 0.5`, even if `0.5` is outside the range of the data.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.

Aesthetics

- `x`: horizontal position
- `y`: vertical position
- `z`: value passed to the summary function

Computed variables

These are calculated by the 'stat' part of layers and can be accessed with delayed evaluation.

- `after_stat(x)`, `after_stat(y)`
  Location.
- `after_stat(value)`
  Value of summary statistic.

Dropped variables

z After binning, the z values of individual data points are no longer available.

Controlling binning parameters for the x and y directions

The arguments `bins`, `binwidth`, `breaks`, `center`, and `boundary` can be set separately for the x and y directions. When given as a scalar, one value applies to both directions. When given as a vector of length two, the first is applied to the x direction and the second to the y direction. Alternatively, these can be a named list containing `x` and `y` elements, for example `list(x = 10, y = 20)`.

See Also

`stat_summary_hex()` for hexagonal summarization. `stat_bin_2d()` for the binning options.

Examples

```
d <- ggplot(diamonds, aes(carat, depth, z = price))
d + stat_summary_2d()

# Specifying function
d + stat_summary_2d(fun = \(x) sum(x^2))
d + stat_summary_2d(fun = ~ 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()
d + stat_summary_hex(fun = ~ sum(.x^2))
}
```

`stat_summary_bin` *Summarise y values at unique/binned x*

Description

`stat_summary()` operates on unique `x` or `y`; `stat_summary_bin()` operates on binned `x` or `y`. They are more flexible versions of `stat_bin()`: instead of just counting, they can compute any aggregate.

Usage

```
stat_summary_bin(
  mapping = NULL,
  data = NULL,
  geom = "pointrange",
  position = "identity",
  ...,
  fun.data = NULL,
  fun = NULL,
  fun.max = NULL,
  fun.min = NULL,
  fun.args = list(),
  bins = 30,
  binwidth = NULL,
  breaks = NULL,
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE,
  fun.y = deprecated(),
  fun.ymin = deprecated(),
  fun.ymax = deprecated()
)

stat_summary(
  mapping = NULL,
  data = NULL,
  geom = "pointrange",
  position = "identity",
  ...,
  fun.data = NULL,
  fun = NULL,
  fun.max = NULL,
  fun.min = NULL,
  fun.args = list(),
  na.rm = FALSE,
  orientation = NA,
  show.legend = NA,
  inherit.aes = TRUE,
  fun.y = deprecated(),
  fun.ymin = deprecated(),
  fun.ymax = deprecated()
)
```

Arguments

`mapping`	Set of aesthetic mappings created by `aes()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function`

	can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following: • A `Geom` ggproto subclass, for example `GeomPoint`. • A string naming the geom. To give the geom as a string, strip the function name of the `geom_` prefix. For example, to use `geom_point()`, give the geom as `"point"`. • For more information and other ways to specify the geom, see the layer geom documentation.
`position`	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The `position` argument accepts the following: • The result of calling a position function, such as `position_jitter()`. This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the `position_` prefix. For example, to use `position_jitter()`, give the position as `"jitter"`. • For more information and other ways to specify the position, see the layer position documentation.
`...`	Other arguments passed on to `layer()`'s `params` argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the `position` argument, or aesthetics that are required can *not* be passed through `...`. Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, `colour = "red"` or `linewidth = 3`. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the `params`. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a `stat_*()` function, the `...` argument can be used to pass on parameters to the `geom` part of the layer. An example of this is `stat_density(geom = "area", outline.type = "both")`. The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a `geom_*()` function, the `...` argument can be used to pass on parameters to the `stat` part of the layer. An example of this is `geom_area(stat = "density", adjust = 0.5)`. The stat's documentation lists which parameters it can accept. • The `key_glyph` argument of `layer()` may also be passed on through `...`. This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
`fun.data`	A function that is given the complete data and should return a data frame with variables `ymin`, `y`, and `ymax`.

`fun.min`, `fun`, `fun.max`	Alternatively, supply three individual functions that are each passed a vector of values and should return a single number.
`fun.args`	Optional additional arguments passed on to the functions.
`bins`	Number of bins. Overridden by `binwidth`. Defaults to 30.
`binwidth`	The width of the bins. Can be specified as a numeric value or as a function that takes x after scale transformation as input and returns a single numeric value. When specifying a function along with a grouping structure, the function will be called once per group. The default is to use the number of bins in `bins`, covering the range of the data. You should always override this value, exploring multiple widths to find the best to illustrate the stories in your data. The bin width of a date variable is the number of days in each time; the bin width of a time variable is the number of seconds.
`breaks`	Alternatively, you can supply a numeric vector giving the bin boundaries. Overrides `binwidth` and `bins`.
`na.rm`	If `FALSE`, the default, missing values are removed with a warning. If `TRUE`, missing values are silently removed.
`orientation`	The orientation of the layer. The default (`NA`) automatically determines the orientation from the aesthetic mapping. In the rare event that this fails it can be given explicitly by setting `orientation` to either `"x"` or `"y"`. See the *Orientation* section for more detail.
`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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use `TRUE`. If `NA`, all levels are shown in legend, but unobserved levels are omitted.
`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. `annotation_borders()`.
`fun.ymin`, `fun.y`, `fun.ymax`	**[Deprecated]** Use the versions specified above instead.

Orientation

This geom treats each axis differently and, thus, can thus have two orientations. Often the orientation is easy to deduce from a combination of the given mappings and the types of positional scales in use. Thus, ggplot2 will by default try to guess which orientation the layer should have. Under rare circumstances, the orientation is ambiguous and guessing may fail. In that case the orientation can be specified directly using the `orientation` parameter, which can be either `"x"` or `"y"`. The value gives the axis that the geom should run along, `"x"` being the default orientation you would expect for the geom.

Summary functions

You can either supply summary functions individually (`fun`, `fun.max`, `fun.min`), 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.min min summary function (should take numeric vector and return single number)

fun main summary function (should take numeric vector and return single number)

fun.max max 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. min and max), use `fun.data`.

`fun.data` will receive data as if it was oriented along the x-axis and should return a data.frame that corresponds to that orientation. The layer will take care of flipping the input and output if it is oriented along the y-axis.

If no aggregation functions are supplied, will default to `mean_se()`.

Aesthetics

`stat_summary()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `x`
- `y`
- `group` → inferred

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

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", linewidth = 2, size = 3)

# Orientation follows the discrete axis
ggplot(mtcars, aes(mpg, factor(cyl))) +
  geom_point() +
  stat_summary(fun.data = "mean_cl_boot", colour = "red", linewidth = 2, size = 3)

# You can supply individual functions to summarise the value at
# each x:
d + stat_summary(fun = "median", colour = "red", size = 2, geom = "point")
d + stat_summary(fun = "mean", colour = "red", size = 2, geom = "point")
d + aes(colour = factor(vs)) + stat_summary(fun = mean, geom="line")

d + stat_summary(fun = mean, fun.min = min, fun.max = max, colour = "red")
```

```
d <- ggplot(diamonds, aes(cut))
d + geom_bar()
d + stat_summary(aes(y = price), fun = "mean", geom = "bar")

# Orientation of stat_summary_bin is ambiguous and must be specified directly
ggplot(diamonds, aes(carat, price)) +
  stat_summary_bin(fun = "mean", geom = "bar", orientation = 'y')


# Don't use ylim to zoom into a summary plot - this throws the
# data away
p <- ggplot(mtcars, aes(cyl, mpg)) +
  stat_summary(fun = "mean", geom = "point")
p
p + ylim(15, 30)
# Instead use coord_cartesian
p + coord_cartesian(ylim = c(15, 30))

# 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, ...)
}
d <- ggplot(mtcars, aes(cyl, mpg)) + geom_point()
# 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", fun.args = list(mult = 1), mapping = aes(group = cyl))
d + stat_sum_df("median_hilow", mapping = aes(group = 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_transform(y="log10")
}
```

stat_unique *Remove duplicates*

Description

Remove duplicates

Usage

```
stat_unique(
  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()`. If specified and `inherit.aes = TRUE` (the default), it is combined with the default mapping at the top level of the plot. You must supply `mapping` if there is no plot mapping.
`data`	The data to be displayed in this layer. There are three options: If `NULL`, the default, the data is inherited from the plot data as specified in the call to `ggplot()`. A `data.frame`, or other object, will override the plot data. All objects will be fortified to produce a data frame. See `fortify()` for which variables will be created. A `function` will be called with a single argument, the plot data. The return value must be a `data.frame`, and will be used as the layer data. A `function` can be created from a `formula` (e.g. `~ head(.x, 10)`).
`geom`	The geometric object to use to display the data for this layer. When using a `stat_*()` function to construct a layer, the `geom` argument can be used to override the default coupling between stats and geoms. The `geom` argument accepts the following:

- A Geom ggproto subclass, for example GeomPoint.
- A string naming the geom. To give the geom as a string, strip the function name of the geom_ prefix. For example, to use geom_point(), give the geom as "point".
- For more information and other ways to specify the geom, see the layer geom documentation.

position	A position adjustment to use on the data for this layer. This can be used in various ways, including to prevent overplotting and improving the display. The position argument accepts the following: • The result of calling a position function, such as position_jitter(). This method allows for passing extra arguments to the position. • A string naming the position adjustment. To give the position as a string, strip the function name of the position_ prefix. For example, to use position_jitter(), give the position as "jitter". • For more information and other ways to specify the position, see the layer position documentation.
...	Other arguments passed on to layer()'s params argument. These arguments broadly fall into one of 4 categories below. Notably, further arguments to the position argument, or aesthetics that are required can *not* be passed through .... Unknown arguments that are not part of the 4 categories below are ignored. • Static aesthetics that are not mapped to a scale, but are at a fixed value and apply to the layer as a whole. For example, colour = "red" or linewidth = 3. The geom's documentation has an **Aesthetics** section that lists the available options. The 'required' aesthetics cannot be passed on to the params. Please note that while passing unmapped aesthetics as vectors is technically possible, the order and required length is not guaranteed to be parallel to the input data. • When constructing a layer using a stat_*() function, the ... argument can be used to pass on parameters to the geom part of the layer. An example of this is stat_density(geom = "area", outline.type = "both"). The geom's documentation lists which parameters it can accept. • Inversely, when constructing a layer using a geom_*() function, the ... argument can be used to pass on parameters to the stat part of the layer. An example of this is geom_area(stat = "density", adjust = 0.5). The stat's documentation lists which parameters it can accept. • The key_glyph argument of layer() may also be passed on through .... This can be one of the functions described as key glyphs, to change the display of the layer in the legend.
na.rm	If FALSE, the default, missing values are removed with a warning. If TRUE, missing values are silently removed.
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. It can also be a named logical vector to finely select the aesthetics to display. To include legend keys for all levels, even when no data exists, use TRUE. If NA, all levels are shown in legend, but unobserved levels are omitted.

`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. `annotation_borders()`.

Aesthetics

`stat_unique()` understands the following aesthetics. Required aesthetics are displayed in bold and defaults are displayed for optional aesthetics:

- `group` → inferred

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

Examples

```
ggplot(mtcars, aes(vs, am)) +
  geom_point(alpha = 0.1)
ggplot(mtcars, aes(vs, am)) +
  geom_point(alpha = 0.1, stat = "unique")
```

`subtheme` *Shortcuts for theme settings*

Description

This collection of functions serves as a shortcut for `theme()` with shorter argument names. Besides the shorter arguments, it also helps in keeping theme declarations more organised.

Usage

```
theme_sub_axis(title, text, ticks, ticks.length, line, minor.ticks.length)

theme_sub_axis_x(title, text, ticks, ticks.length, line, minor.ticks.length)

theme_sub_axis_y(title, text, ticks, ticks.length, line, minor.ticks.length)

theme_sub_axis_bottom(
  title,
  text,
  ticks,
  ticks.length,
  line,
  minor.ticks,
  minor.ticks.length
)
```

```
theme_sub_axis_top(
  title,
  text,
  ticks,
  ticks.length,
  line,
  minor.ticks,
  minor.ticks.length
)

theme_sub_axis_left(
  title,
  text,
  ticks,
  ticks.length,
  line,
  minor.ticks,
  minor.ticks.length
)

theme_sub_axis_right(
  title,
  text,
  ticks,
  ticks.length,
  line,
  minor.ticks,
  minor.ticks.length
)

theme_sub_legend(
  text,
  text.position,
  title,
  title.position,
  background,
  frame,
  ticks,
  ticks.length,
  axis.line,
  spacing,
  spacing.x,
  spacing.y,
  margin,
  key,
  key.size,
  key.height,
  key.width,
  key.spacing,
  key.spacing.x,
  key.spacing.y,
  key.justification,
  byrow,
  position,
  direction,
  location,
  position.inside,
  justification,
  justification.top,
  justification.bottom,
  justification.left,
  justification.right,
  justification.inside,
  box,
  box.just,
  box.margin,
  box.background,
  box.spacing
)

theme_sub_panel(
  background,
  border,
  widths,
  heights,
  spacing,
  spacing.x,
  spacing.y,
  grid,
  grid.major,
  grid.minor,
  grid.major.x,
  grid.major.y,
  grid.minor.x,
  grid.minor.y,
  ontop
)

theme_sub_plot(
  background,
  title,
  title.position,
  subtitle,
  caption,
  caption.position,
  tag,
  tag.position,
  tag.location,
  margin
)

theme_sub_strip(
  background,
  background.x,
  background.y,
  clip,
  placement,
  text,
  text.x,
  text.x.bottom,
  text.x.top,
  text.y,
  text.y.left,
  text.y.right,
  switch.pad.grid,
  switch.pad.wrap
)
```

Arguments

`axis.line, background, background.x, background.y, border, box, box.background, box.just, box.margin, box.spacing, byrow, caption, caption.position, clip, direction, frame, grid, grid.major, grid.major.x, grid.major.y, grid.minor, grid.minor.x, grid.minor.y, heights, justification, justification.bottom, justification.inside, justification.left, justification.right, justification.top, key, key.height, key.justification, key.size, key.spacing, key.spacing.x, key.spacing.y, key.width, line, location, margin, minor.ticks, minor.ticks.length, ontop, placement, position, position.inside, spacing, spacing.x, spacing.y, subtitle, switch.pad.grid, switch.pad.wrap, tag, tag.location, tag.position, text, text.position, text.x, text.x.bottom, text.x.top, text.y, text.y.left, text.y.right, ticks, ticks.length, title, title.position, widths`	Arguments that are renamed and passed on to `theme()`.

Value

A `theme`-class object that can be added to a plot.

Functions

- `theme_sub_axis()`: Theme specification for all axes.
- `theme_sub_axis_x()`: Theme specification for both x axes.
- `theme_sub_axis_y()`: Theme specification for both y axes.

- `theme_sub_axis_bottom()`: Theme specification for the bottom x axis.
- `theme_sub_axis_top()`: Theme specification for the top x axis.
- `theme_sub_axis_left()`: Theme specification for the left y axis.
- `theme_sub_axis_right()`: Theme specification for the right y axis.
- `theme_sub_legend()`: Theme specification for the legend.
- `theme_sub_panel()`: Theme specification for the panels.
- `theme_sub_plot()`: Theme specification for the whole plot.
- `theme_sub_strip()`: Theme specification for facet strips.

Examples

```
# A standard plot
p <- ggplot(mtcars, aes(disp, mpg, colour = drat)) +
  geom_point()

red_text <- element_text(colour = "red")
red_line <- element_line(colour = "red")

# The theme settings below:
p + theme(
  axis.title.x.bottom = red_text,
  axis.text.x.bottom  = red_text,
  axis.line.x.bottom  = red_line,
  axis.ticks.x.bottom = red_line
)

# Are equivalent to these less verbose theme settings
p + theme_sub_axis_bottom(
  title = red_text,
  text  = red_text,
  line  = red_line,
  ticks = red_line
)
```

`theme` *Modify components of a theme*

Description

Themes are a powerful way to customize the non-data components of your plots: i.e. titles, labels, fonts, background, gridlines, and legends. Themes can be used to give plots a consistent customized look. Modify a single plot's theme using `theme()`; see `theme_update()` if you want modify the active theme, to affect all subsequent plots. Use the themes available in complete themes if you would like to use a complete theme such as `theme_bw()`, `theme_minimal()`, and more. Theme elements are documented together according to inheritance, read more about theme inheritance below.

Usage

```
theme(
  ...,
  line,
  rect,
  text,
  title,
  point,
  polygon,
  geom,
  spacing,
  margins,
  aspect.ratio,
  axis.title,
  axis.title.x,
  axis.title.x.top,
  axis.title.x.bottom,
  axis.title.y,
  axis.title.y.left,
  axis.title.y.right,
  axis.text,
  axis.text.x,
  axis.text.x.top,
  axis.text.x.bottom,
  axis.text.y,
  axis.text.y.left,
  axis.text.y.right,
  axis.text.theta,
  axis.text.r,
  axis.ticks,
  axis.ticks.x,
  axis.ticks.x.top,
  axis.ticks.x.bottom,
  axis.ticks.y,
  axis.ticks.y.left,
  axis.ticks.y.right,
  axis.ticks.theta,
  axis.ticks.r,
  axis.minor.ticks.x.top,
  axis.minor.ticks.x.bottom,
  axis.minor.ticks.y.left,
  axis.minor.ticks.y.right,
  axis.minor.ticks.theta,
  axis.minor.ticks.r,
  axis.ticks.length,
  axis.ticks.length.x,
  axis.ticks.length.x.top,
  axis.ticks.length.x.bottom,
  axis.ticks.length.y,
  axis.ticks.length.y.left,
  axis.ticks.length.y.right,
  axis.ticks.length.theta,
  axis.ticks.length.r,
  axis.minor.ticks.length,
  axis.minor.ticks.length.x,
  axis.minor.ticks.length.x.top,
  axis.minor.ticks.length.x.bottom,
  axis.minor.ticks.length.y,
  axis.minor.ticks.length.y.left,
  axis.minor.ticks.length.y.right,
  axis.minor.ticks.length.theta,
  axis.minor.ticks.length.r,
  axis.line,
  axis.line.x,
  axis.line.x.top,
  axis.line.x.bottom,
  axis.line.y,
  axis.line.y.left,
  axis.line.y.right,
  axis.line.theta,
  axis.line.r,
  legend.background,
  legend.margin,
  legend.spacing,
  legend.spacing.x,
  legend.spacing.y,
  legend.key,
  legend.key.size,
  legend.key.height,
  legend.key.width,
  legend.key.spacing,
  legend.key.spacing.x,
  legend.key.spacing.y,
  legend.key.justification,
  legend.frame,
  legend.ticks,
  legend.ticks.length,
  legend.axis.line,
  legend.text,
  legend.text.position,
  legend.title,
  legend.title.position,
  legend.position,
  legend.position.inside,
  legend.direction,
  legend.byrow,
  legend.justification,
  legend.justification.top,
  legend.justification.bottom,
  legend.justification.left,
  legend.justification.right,
  legend.justification.inside,
  legend.location,
  legend.box,
  legend.box.just,
  legend.box.margin,
  legend.box.background,
  legend.box.spacing,
  panel.background,
  panel.border,
  panel.spacing,
  panel.spacing.x,
  panel.spacing.y,
  panel.grid,
  panel.grid.major,
  panel.grid.minor,
  panel.grid.major.x,
  panel.grid.major.y,
  panel.grid.minor.x,
  panel.grid.minor.y,
  panel.ontop,
  panel.widths,
  panel.heights,
  plot.background,
  plot.title,
  plot.title.position,
  plot.subtitle,
  plot.caption,
  plot.caption.position,
  plot.tag,
  plot.tag.position,
  plot.tag.location,
  plot.margin,
  strip.background,
  strip.background.x,
  strip.background.y,
  strip.clip,
  strip.placement,
  strip.text,
  strip.text.x,
  strip.text.x.bottom,
  strip.text.x.top,
  strip.text.y,
  strip.text.y.left,
  strip.text.y.right,
  strip.switch.pad.grid,
  strip.switch.pad.wrap,
  complete = FALSE,
  validate = TRUE
)
```

Arguments

`...`	additional element specifications not part of base ggplot2. In general, these should also be defined in the `element tree` argument. Splicing a list is also supported.
`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`)
`point`	all point elements (`element_point()`)
`polygon`	all polygon elements (`element_polygon()`)
`geom`	defaults for geoms (`element_geom()`)
`spacing`	all spacings (`unit()`)
`margins`	all margins (`margin()`)
`aspect.ratio`	aspect ratio of the panel
`axis.title`, `axis.title.x`, `axis.title.y`, `axis.title.x.top`, `axis.title.x.bottom`, `axis.title.y.left`, `axis.title.y.right`	labels of axes (`element_text()`). Specify all axes' labels (`axis.title`), labels by plane (using `axis.title.x` or `axis.title.y`), or individually for each axis (using `axis.title.x.bottom`, `axis.title.x.top`, `axis.title.y.left`, `axis.title.y.right`). `axis.title.*.*` inherits from `axis.title.*` which inherits from `axis.title`, which in turn inherits from `text`
`axis.text`, `axis.text.x`, `axis.text.y`, `axis.text.x.top`, `axis.text.x.bottom`, `axis.text.y.left`, `axis.text.y.right`, `axis.text.theta`, `axis.text.r`	tick labels along axes (`element_text()`). Specify all axis tick labels (`axis.text`), tick labels by plane (using `axis.text.x` or `axis.text.y`), or individually for each axis (using `axis.text.x.bottom`, `axis.text.x.top`, `axis.text.y.left`, `axis.text.y.right`). `axis.text.*.*` inherits from `axis.text.*` which inherits from `axis.text`, which in turn inherits from `text`
`axis.ticks`, `axis.ticks.x`, `axis.ticks.x.top`, `axis.ticks.x.bottom`, `axis.ticks.y`, `axis.ticks.y.left`, `axis.ticks.y.right`, `axis.ticks.theta`, `axis.ticks.r`	tick marks along axes (`element_line()`). Specify all tick marks (`axis.ticks`), ticks by plane (using `axis.ticks.x` or `axis.ticks.y`), or individually for each axis (using `axis.ticks.x.bottom`, `axis.ticks.x.top`, `axis.ticks.y.left`, `axis.ticks.y.right`). `axis.ticks.*.*` inherits from `axis.ticks.*` which inherits from `axis.ticks`, which in turn inherits from `line`

`axis.minor.ticks.x.top`, `axis.minor.ticks.x.bottom`, `axis.minor.ticks.y.left`, `axis.minor.ticks.y.right`, `axis.minor.ticks.theta`, `axis.minor.ticks.r`	minor tick marks along axes (`element_line()`). `axis.minor.ticks.*.*` inherit from the corresponding major ticks `axis.ticks.*.*`.
`axis.ticks.length`, `axis.ticks.length.x`, `axis.ticks.length.x.top`, `axis.ticks.length.x.bottom`, `axis.ticks.length.y`, `axis.ticks.length.y.left`, `axis.ticks.length.y.right`, `axis.ticks.length.theta`, `axis.ticks.length.r`	length of tick marks (`unit`). `axis.ticks.length` inherits from `spacing`.
`axis.minor.ticks.length`, `axis.minor.ticks.length.x`, `axis.minor.ticks.length.x.top`, `axis.minor.ticks.length.x.bottom`, `axis.minor.ticks.length.y`, `axis.minor.ticks.length.y.left`, `axis.minor.ticks.length.y.right`, `axis.minor.ticks.length.theta`, `axis.minor.ticks.length.r`	length of minor tick marks (`unit`), or relative to `axis.ticks.length` when provided with `rel()`.
`axis.line`, `axis.line.x`, `axis.line.x.top`, `axis.line.x.bottom`, `axis.line.y`, `axis.line.y.left`, `axis.line.y.right`, `axis.line.theta`, `axis.line.r`	lines along axes (`element_line()`). Specify lines along all axes (`axis.line`), lines for each plane (using `axis.line.x` or `axis.line.y`), or individually for each axis (using `axis.line.x.bottom`, `axis.line.x.top`, `axis.line.y.left`, `axis.line.y.right`). `axis.line.*.*` inherits from `axis.line.*` which inherits from `axis.line`, which in turn inherits from `line`
`legend.background`	background of legend (`element_rect()`; inherits from `rect`)
`legend.margin`	the margin around each legend (`margin()`); inherits from `margins`.
`legend.spacing`, `legend.spacing.x`, `legend.spacing.y`	the spacing between legends (`unit`). `legend.spacing.x` & `legend.spacing.y` inherit from `legend.spacing` or can be specified separately. `legend.spacing` inherits from `spacing`.
`legend.key`	background underneath legend keys (`element_rect()`; inherits from `rect`)
`legend.key.size`, `legend.key.height`, `legend.key.width`	size of legend keys (`unit`); key background height & width inherit from `legend.key.size` or can be specified separately. In turn `legend.key.size` inherits from `spacing`.
`legend.key.spacing`, `legend.key.spacing.x`, `legend.key.spacing.y`	spacing between legend keys given as a `unit`. Spacing in the horizontal (x) and vertical (y) direction inherit from `legend.key.spacing` or can be specified separately. `legend.key.spacing` inherits from `spacing`.
`legend.key.justification`	Justification for positioning legend keys when more space is available than needed for display. The default, `NULL`, stretches keys into the available space. Can be a location like `"center"` or `"top"`, or a two-element numeric vector.
`legend.frame`	frame drawn around the bar (`element_rect()`).
`legend.ticks`	tick marks shown along bars or axes (`element_line()`)

`legend.ticks.length`
: length of tick marks in legend (`unit()`); inherits from `legend.key.size`.

`legend.axis.line`
: lines along axes in legends (`element_line()`)

`legend.text`
: legend item labels (`element_text()`; inherits from `text`)

`legend.text.position`
: placement of legend text relative to legend keys or bars ("top", "right", "bottom" or "left"). The legend text placement might be incompatible with the legend's direction for some guides.

`legend.title`
: title of legend (`element_text()`; inherits from `title`)

`legend.title.position`
: placement of legend title relative to the main legend ("top", "right", "bottom" or "left").

`legend.position`
: the default position of legends ("none", "left", "right", "bottom", "top", "inside")

`legend.position.inside`
: A numeric vector of length two setting the placement of legends that have the `"inside"` position.

`legend.direction`
: layout of items in legends ("horizontal" or "vertical")

`legend.byrow`
: whether the legend-matrix is filled by columns (`FALSE`, the default) or by rows (`TRUE`).

`legend.justification`
: anchor point for positioning legend inside plot ("center" or two-element numeric vector) or the justification according to the plot area when positioned outside the plot

`legend.justification.top`, `legend.justification.bottom`, `legend.justification.left`, `legend.justification.right`, `legend.justification.inside`
: Same as `legend.justification` but specified per `legend.position` option.

`legend.location`
: Relative placement of legends outside the plot as a string. Can be `"panel"` (default) to align legends to the panels or `"plot"` to align legends to the plot as a whole.

`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", "center" or "centre")

`legend.box.margin`
: margins around the full legend area, as specified using `margin()`; inherits from `margins`.

`legend.box.background`
: background of legend area (`element_rect()`; inherits from `rect`)

`legend.box.spacing`
: The spacing between the plotting area and the legend box (`unit`); inherits from `spacing`.

`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 be used with `fill = NA` (`element_rect()`; inherits from `rect`)

`panel.spacing`, `panel.spacing.x`, `panel.spacing.y`
: spacing between facet panels (`unit`). `panel.spacing.x` & `panel.spacing.y` inherit from `panel.spacing` or can be specified separately. `panel.spacing` inherits from `spacing`.

`panel.grid`, `panel.grid.major`, `panel.grid.minor`, `panel.grid.major.x`, `panel.grid.major.y`, `panel.grid.minor.x`, `panel.grid.minor.y`
: grid lines (`element_line()`). Specify major grid lines, or minor grid lines separately (using `panel.grid.major` or `panel.grid.minor`) or individually for each axis (using `panel.grid.major.x`, `panel.grid.minor.x`, `panel.grid.major.y`, `panel.grid.minor.y`). Y axis grid lines are horizontal and x axis grid lines are vertical. `panel.grid.*.*` inherits from `panel.grid.*` which inherits from `panel.grid`, which in turn inherits from `line`

`panel.ontop`
: option to place the panel (background, gridlines) over the data layers (`logical`). Usually used with a transparent or blank `panel.background`.

`panel.widths`, `panel.heights`
: Sizes for panels (`units`). Can be a single unit to set the total size for the panel area, or a unit vector to set the size of individual panels.

`plot.background`
: background of the entire plot (`element_rect()`; inherits from `rect`)

`plot.title`
: plot title (text appearance) (`element_text()`; inherits from `title`) left-aligned by default

`plot.title.position`, `plot.caption.position`
: Alignment of the plot title/subtitle and caption. The setting for `plot.title.position` applies to both the title and the subtitle. A value of "panel" (the default) means that titles and/or caption are aligned to the plot panels. A value of "plot" means that titles and/or caption are aligned to the entire plot (minus any space for margins and plot tag).

`plot.subtitle`
: plot subtitle (text appearance) (`element_text()`; inherits from `title`) left-aligned by default

`plot.caption`
: caption below the plot (text appearance) (`element_text()`; inherits from `title`) right-aligned by default

`plot.tag`
: upper-left label to identify a plot (text appearance) (`element_text()`; inherits from `title`) left-aligned by default

`plot.tag.position`
: The position of the tag as a string ("topleft", "top", "topright", "left", "right", "bottomleft", "bottom", "bottomright") or a coordinate. If a coordinate, can be a numeric vector of length 2 to set the x,y-coordinate relative to the whole plot. The coordinate option is unavailable for `plot.tag.location = "margin"`.

`plot.tag.location`	The placement of the tag as a string, one of `"panel"`, `"plot"` or `"margin"`. Respectively, these will place the tag inside the panel space, anywhere in the plot as a whole, or in the margin around the panel space.
`plot.margin`	margin around entire plot (`unit` with the sizes of the top, right, bottom, and left margins); inherits from `margin`.
`strip.background`, `strip.background.x`, `strip.background.y`	background of facet labels (`element_rect()`; inherits from `rect`). Horizontal facet background (`strip.background.x`) & vertical facet background (`strip.background.y`) inherit from `strip.background` or can be specified separately
`strip.clip`	should strip background edges and strip labels be clipped to the extend of the strip background? Options are `"on"` to clip, `"off"` to disable clipping or `"inherit"` (default) to take the clipping setting from the parent viewport.
`strip.placement`	placement of strip with respect to axes, either "inside" or "outside". Only important when axes and strips are on the same side of the plot.
`strip.text`, `strip.text.x`, `strip.text.y`, `strip.text.x.top`, `strip.text.x.bottom`, `strip.text.y.left`, `strip.text.y.right`	facet labels (`element_text()`; inherits from `text`). Horizontal facet labels (`strip.text.x`) & vertical facet labels (`strip.text.y`) inherit from `strip.text` or can be specified separately. Facet strips have dedicated position-dependent theme elements (`strip.text.x.top`, `strip.text.x.bottom`, `strip.text.y.left`, `strip.text.y.right`) that inherit from `strip.text.x` and `strip.text.y`, respectively. As a consequence, some theme stylings need to be applied to the position-dependent elements rather than to the parent elements
`strip.switch.pad.grid`, `strip.switch.pad.wrap`	space between strips and axes when strips are switched (`unit`); inherits from `spacing`.
`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. Also, when setting `complete = TRUE` all elements will be set to inherit from blank elements.
`validate`	`TRUE` to run `check_element()`, `FALSE` to bypass checks.

Theme inheritance

Theme elements inherit properties from other theme elements hierarchically. For example, `axis.title.x.bottom` inherits from `axis.title.x` which 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`. This means that you can modify the appearance of multiple elements by setting a single high-level component.

Learn more about setting these aesthetics in `vignette("ggplot2-specs")`.

See Also

`add_gg()` and %+replace%, `element_blank()`, `element_line()`, `element_rect()`, and `element_text()` for details of the specific theme elements.

The modifying theme components and theme elements sections of the online ggplot2 book.

Examples

```
p1 <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  labs(title = "Fuel economy declines as weight increases")
p1

# Plot ---------------------------------------------------------------------
p1 + theme(plot.title = element_text(size = rel(2)))
p1 + theme(plot.background = element_rect(fill = "green"))

# Panels --------------------------------------------------------------------

p1 + theme(panel.background = element_rect(fill = "white", colour = "grey50"))
p1 + theme(panel.border = element_rect(linetype = "dashed"))
p1 + theme(panel.grid.major = element_line(colour = "black"))
p1 + theme(
  panel.grid.major.y = element_blank(),
  panel.grid.minor.y = element_blank()
)

# Put gridlines on top of data
p1 + theme(
  panel.background = element_rect(fill = NA),
  panel.grid.major = element_line(colour = "grey50"),
  panel.ontop = TRUE
)

# Axes ----------------------------------------------------------------------
# Change styles of axes texts and lines
p1 + theme(axis.line = element_line(linewidth = 3, colour = "grey80"))
p1 + theme(axis.text = element_text(colour = "blue"))
p1 + theme(axis.ticks = element_line(linewidth = 2))

# Change the appearance of the y-axis title
p1 + theme(axis.title.y = element_text(size = rel(1.5), angle = 90))

# Make ticks point outwards on y-axis and inwards on x-axis
p1 + theme(
  axis.ticks.length.y = unit(.25, "cm"),
  axis.ticks.length.x = unit(-.25, "cm"),
  axis.text.x = element_text(margin = margin(t = .3, unit = "cm"))
)


# Legend --------------------------------------------------------------------
p2 <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point(aes(colour = factor(cyl), shape = factor(vs))) +
  labs(
    x = "Weight (1000 lbs)",
    y = "Fuel economy (mpg)",
    colour = "Cylinders",
    shape = "Transmission"
  )
p2

# Position
p2 + theme(legend.position = "none")
p2 + theme(legend.justification = "top")
p2 + theme(legend.position = "bottom")

# Or place legends inside the plot using relative coordinates between 0 and 1
# legend.justification sets the corner that the position refers to
p2 + theme(
  legend.position = "inside",
  legend.position.inside = c(.95, .95),
  legend.justification = c("right", "top"),
  legend.box.just = "right",
  legend.margin = margin_auto(6)
)

# The legend.box properties work similarly for the space around
# all the legends
p2 + theme(
  legend.box.background = element_rect(),
  legend.box.margin = margin_auto(6)
)

# You can also control the display of the keys
# and the justification related to the plot area can be set
p2 + theme(legend.key = element_rect(fill = "white", colour = "black"))
p2 + theme(legend.text = element_text(size = 8, colour = "red"))
p2 + theme(legend.title = element_text(face = "bold"))

# Strips --------------------------------------------------------------------

p3 <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  facet_wrap(~ cyl)
p3

p3 + theme(strip.background = element_rect(colour = "black", fill = "white"))
p3 + theme(strip.text.x = element_text(colour = "white", face = "bold"))
# More direct strip.text.x here for top
# as in the facet_wrap the default strip.position is "top"
p3 + theme(strip.text.x.top = element_text(colour = "white", face = "bold"))
p3 + theme(panel.spacing = unit(1, "lines"))

# Colours -------------------------------------------------------------------

p4 <- ggplot(mtcars, aes(wt, mpg)) +
  geom_point() +
  annotate(label = "Text Annotation", x = 5, y = 30, geom = "text")

# You can use the 'ink' setting to set colour defaults for all layers
p4 + theme(geom = element_geom(ink = "dodgerblue"))
```

```
# Alternate colours are derived from the 'accent' and 'paper' settings
p4 + geom_smooth(method = "lm", formula = y ~ x) +
  theme(geom = element_geom(accent = "tomato", paper = "orchid"))
# You can also set default palettes in the theme
p4 + aes(colour = drat) +
  theme(palette.colour.continuous = c("white", "pink", "hotpink"))
```

txhousing *Housing sales in TX*

Description

Information about the housing market in Texas provided by the TAMU real estate center, https://trerc.tamu.edu/.

Usage

```
txhousing
```

Format

A data frame with 8602 observations and 9 variables:

city Name of multiple listing service (MLS) area

year,month,date Date

sales Number of sales

volume Total value of sales

median Median sale price

listings Total active listings

inventory "Months inventory": amount of time it would take to sell all current listings at current pace of sales.

vars *Quote faceting variables*

Description

Just like `aes()`, `vars()` is a quoting function that takes inputs to be evaluated in the context of a dataset. These inputs can be:

- variable names
- complex expressions

In both cases, the results (the vectors that the variable represents or the results of the expressions) are used to form faceting groups.

Usage

```
vars(...)
```

Arguments

`...`	<`data-masking`> Variables or expressions automatically quoted. These are evaluated in the context of the data to form faceting groups. Can be named (the names are passed to a labeller).

See Also

`aes()`, `facet_wrap()`, `facet_grid()`

Examples

```
p <- ggplot(mtcars, aes(wt, disp)) + geom_point()
p + facet_wrap(vars(vs, am))

# vars() makes it easy to pass variables from wrapper functions:
wrap_by <- function(...) {
  facet_wrap(vars(...), labeller = label_both)
}
p + wrap_by(vs)
p + wrap_by(vs, am)

# You can also supply expressions to vars(). In this case it's often a
# good idea to supply a name as well:
p + wrap_by(drat = cut_number(drat, 3))

# Let's create another function for cutting and wrapping a
# variable. This time it will take a named argument instead of dots,
# so we'll have to use the "enquote and unquote" pattern:
wrap_cut <- function(var, n = 3) {
  # Let's enquote the named argument `var` to make it auto-quoting:
  var <- enquo(var)

  # `as_label()` will create a nice default name:
  nm <- as_label(var)

  # Now let's unquote everything at the right place. Note that we also
  # unquote `n` just in case the data frame has a column named
  # `n`. The latter would have precedence over our local variable
  # because the data is always masking the environment.
  wrap_by(!!nm := cut_number(!!var, !!n))
}

# Thanks to tidy eval idioms we now have another useful wrapper:
p + wrap_cut(drat)
```

Index