Package ‘scales’

November 28, 2023

Title Scale Functions for Visualization

Version 1.3.0

Description Graphical scales map data to aesthetics, and provide methods for automatically determining breaks and labels for axes and legends.

License MIT + file LICENSE

URL https://scales.r-lib.org, https://github.com/r-lib/scales

BugReports https://github.com/r-lib/scales/issues

Depends R (>= 3.6)

Imports cli, farver (>= 2.0.3), glue, labeling, lifecycle, munsell (>= 0.5), R6, RColorBrewer, rlang (>= 1.0.0), viridisLite

Suggests bit64, covr, dichromat, ggplot2, hms (>= 0.5.0), stringi, testthat (>= 3.0.0)

Config/Needs/website tidyverse/tidytemplate

Config/testthat/edition 3

Encoding UTF-8

LazyLoad yes

RoxygenNote 7.2.3

NeedsCompilation yes

Author Hadley Wickham [aut],
Thomas Lin Pedersen [cre, aut]
(<https://orcid.org/0000-0002-5147-4711>),
Dana Seidel [aut],
Posit, PBC [cph, fnd]

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

Repository CRAN

Date/Publication 2023-11-28 09:10:06 UTC

R topics documented:

alpha *Modify colour transparency*

Description

Vectorised in both colour and alpha.

Usage

```
alpha(colour, alpha = NA)
```

Arguments

colour	colour
alpha	new alpha level in [0,1]. If alpha is NA, existing alpha values are preserved.

Examples

```
alpha("red", 0.1)
alpha(colours(), 0.5)
alpha("red", seq(0, 1, length.out = 10))
alpha(c("first" = "gold", "second" = "lightgray", "third" = "#cd7f32"), .5)
```

`breaks_extended` *Automatic breaks for numeric axes*

Description

Uses Wilkinson's extended breaks algorithm as implemented in the **labeling** package.

Usage

```
breaks_extended(n = 5, ...)
```

Arguments

`n` Desired number of breaks. You may get slightly more or fewer breaks that requested.

`...` other arguments passed on to `labeling::extended()`

References

Talbot, J., Lin, S., Hanrahan, P. (2010) An Extension of Wilkinson's Algorithm for Positioning Tick Labels on Axes, InfoVis 2010 `http://vis.stanford.edu/files/2010-TickLabels-InfoVis.pdf`.

Examples

```
demo_continuous(c(0, 10))
demo_continuous(c(0, 10), breaks = breaks_extended(3))
demo_continuous(c(0, 10), breaks = breaks_extended(10))
```

`breaks_log` *Breaks for log axes*

Description

This algorithm starts by looking for integer powers of base. If that doesn't provide enough breaks, it then looks for additional intermediate breaks which are integer multiples of integer powers of base. If that fails (which it can for very small ranges), we fall back to `extended_breaks()`

Usage

```
breaks_log(n = 5, base = 10)
```

Arguments

`n` desired number of breaks

`base` base of logarithm to use

Details

The algorithm starts by looking for a set of integer powers of base that cover the range of the data. If that does not generate at least n - 2 breaks, we look for an integer between 1 and base that splits the interval approximately in half. For example, in the case of base = 10, this integer is 3 because log10(3) = 0.477. This leaves 2 intervals: c(1, 3) and c(3, 10). If we still need more breaks, we look for another integer that splits the largest remaining interval (on the log-scale) approximately in half. For base = 10, this is 5 because log10(5) = 0.699.

The generic algorithm starts with a set of integers steps containing only 1 and a set of candidate integers containing all integers larger than 1 and smaller than base. Then for each remaining candidate integer x, the smallest interval (on the log-scale) in the vector sort(c(x, steps, base)) is calculated. The candidate x which yields the largest minimal interval is added to steps and removed from the candidate set. This is repeated until either a sufficient number of breaks, >= n-2, are returned or all candidates have been used.

Examples

```
demo_log10(c(1, 1e5))
demo_log10(c(1, 1e6))

# Request more breaks by setting n
demo_log10(c(1, 1e6), breaks = breaks_log(6))

# Some tricky ranges
demo_log10(c(2000, 9000))
demo_log10(c(2000, 14000))
demo_log10(c(2000, 85000), expand = c(0, 0))

# An even smaller range that requires falling back to linear breaks
demo_log10(c(1800, 2000))
```

breaks_pretty *Pretty breaks for date/times*

Description

Uses default R break algorithm as implemented in pretty(). This is primarily useful for date/times, as extended_breaks() should do a slightly better job for numeric scales.

Usage

```
breaks_pretty(n = 5, ...)
```

Arguments

n	Desired number of breaks. You may get slightly more or fewer breaks that requested.
...	other arguments passed on to pretty()

Examples

```
one_month <- as.POSIXct(c("2020-05-01", "2020-06-01"))
demo_datetime(one_month)
demo_datetime(one_month, breaks = breaks_pretty(2))
demo_datetime(one_month, breaks = breaks_pretty(4))

# Tightly spaced date breaks often need custom labels too
demo_datetime(one_month, breaks = breaks_pretty(12))
demo_datetime(one_month,
  breaks = breaks_pretty(12),
  labels = label_date_short()
)
```

`breaks_timespan` *Breaks for timespan data*

Description

As timespan units span a variety of bases (1000 below seconds, 60 for second and minutes, 24 for hours, and 7 for days), the range of the input data determines the base used for calculating breaks

Usage

```
breaks_timespan(unit = c("secs", "mins", "hours", "days", "weeks"), n = 5)
```

Arguments

`unit`	The unit used to interpret numeric data input
`n`	Desired number of breaks. You may get slightly more or fewer breaks that requested.

Examples

```
demo_timespan(seq(0, 100), breaks = breaks_timespan())
```

`breaks_width` *Equally spaced breaks*

Description

Useful for numeric, date, and date-time scales.

Usage

```
breaks_width(width, offset = 0)
```

Arguments

`width` Distance between each break. Either a number, or for date/times, a single string of the form `"{n} {unit}"`, e.g. "1 month", "5 days". Unit can be of one "sec", "min", "hour", "day", "week", "month", "year".

`offset` Use if you don't want breaks to start at zero, or on a conventional date or time boundary such as the 1st of January or midnight. Either a number, or for date/times, a single string of the form `"{n} {unit}"`, as for `width`.

`offset` can be a vector, which will accumulate in the order given. This is mostly useful for dates, where e.g. `c("3 months", "5 days")` will offset by three months and five days, which is useful for the UK tax year. Note that due to way that dates are rounded, there's no guarantee that `offset = c(x, y)` will give the same result as `offset = c(y, x)`.

Examples

```
demo_continuous(c(0, 100))
demo_continuous(c(0, 100), breaks = breaks_width(10))
demo_continuous(c(0, 100), breaks = breaks_width(20, -4))
demo_continuous(c(0, 100), breaks = breaks_width(20, 4))

# This is also useful for dates
one_month <- as.POSIXct(c("2020-05-01", "2020-06-01"))
demo_datetime(one_month)
demo_datetime(one_month, breaks = breaks_width("1 week"))
demo_datetime(one_month, breaks = breaks_width("5 days"))
# This is so useful that scale_x_datetime() has a shorthand:
demo_datetime(one_month, date_breaks = "5 days")

# hms times also work
one_hour <- hms::hms(hours = 0:1)
demo_time(one_hour)
demo_time(one_hour, breaks = breaks_width("15 min"))
demo_time(one_hour, breaks = breaks_width("600 sec"))

# Offets are useful for years that begin on dates other than the 1st of
# January, such as the UK financial year, which begins on the 1st of April.
three_years <- as.POSIXct(c("2020-01-01", "2021-01-01", "2022-01-01"))
demo_datetime(
  three_years,
  breaks = breaks_width("1 year", offset = "3 months")
)

# The offset can be a vector, to create offsets that have compound units,
# such as the UK fiscal (tax) year, which begins on the 6th of April.
demo_datetime(
  three_years,
  breaks = breaks_width("1 year", offset = c("3 months", "5 days"))
)
```

col2hcl *Modify standard R colour in hcl colour space.*

Description

Transforms rgb to hcl, sets non-missing arguments and then backtransforms to rgb.

Usage

```
col2hcl(colour, h = NULL, c = NULL, l = NULL, alpha = NULL)
```

Arguments

colour	character vector of colours to be modified
h	Hue, [0, 360]
c	Chroma, [0, 100]
l	Luminance, [0, 100]
alpha	Alpha, [0, 1].

Examples

```
reds <- rep("red", 6)
show_col(col2hcl(reds, h = seq(0, 180, length = 6)))
show_col(col2hcl(reds, c = seq(0, 80, length = 6)))
show_col(col2hcl(reds, l = seq(0, 100, length = 6)))
show_col(col2hcl(reds, alpha = seq(0, 1, length = 6)))
```

colour_ramp *Fast colour interpolation*

Description

Returns a function that maps the interval [0,1] to a set of colours. Interpolation is performed in the CIELAB colour space. Similar to colorRamp(space = 'Lab'), but hundreds of times faster, and provides results in "#RRGGBB" (or "#RRGGBBAA") character form instead of RGB colour matrices.

Usage

```
colour_ramp(colors, na.color = NA, alpha = TRUE)
```

Arguments

`colors` Colours to interpolate; must be a valid argument to `grDevices::col2rgb()`. This can be a character vector of `"#RRGGBB"` or `"#RRGGBBAA"`, colour names from `grDevices::colors()`, or a positive integer that indexes into `grDevices::palette()`.

`na.color` The colour to map to `NA` values (for example, `"#606060"` for dark grey, or `"#00000000"` for transparent) and values outside of [0,1]. Can itself by `NA`, which will simply cause an `NA` to be inserted into the output.

`alpha` Whether to include alpha transparency channels in interpolation. If `TRUE` then the alpha information is included in the interpolation. The returned colours will be provided in `"#RRGGBBAA"` format when needed, i.e., in cases where the colour is not fully opaque, so that the `"AA"` part is not equal to `"FF"`. Fully opaque colours will be returned in `"#RRGGBB"` format. If `FALSE`, the alpha information is discarded before interpolation and colours are always returned as `"#RRGGBB"`.

Value

A function that takes a numeric vector and returns a character vector of the same length with RGB or RGBA hex colours.

See Also

`colorRamp`

Examples

```
ramp <- colour_ramp(c("red", "green", "blue"))
show_col(ramp(seq(0, 1, length = 12)))
```

`col_numeric` *Colour mapping*

Description

Conveniently maps data values (numeric or factor/character) to colours according to a given palette, which can be provided in a variety of formats.

Usage

```
col_numeric(
  palette,
  domain,
  na.color = "#808080",
  alpha = FALSE,
  reverse = FALSE
)

col_bin(
  palette,
  domain,
  bins = 7,
  pretty = TRUE,
  na.color = "#808080",
  alpha = FALSE,
  reverse = FALSE,
  right = FALSE
)

col_quantile(
  palette,
  domain,
  n = 4,
  probs = seq(0, 1, length.out = n + 1),
  na.color = "#808080",
  alpha = FALSE,
  reverse = FALSE,
  right = FALSE
)

col_factor(
  palette,
  domain,
  levels = NULL,
  ordered = FALSE,
  na.color = "#808080",
  alpha = FALSE,
  reverse = FALSE
)
```

Arguments

palette — The colours or colour function that values will be mapped to

domain — The possible values that can be mapped.

For `col_numeric` and `col_bin`, this can be a simple numeric range (e.g. `c(0, 100)`); `col_quantile` needs representative numeric data; and `col_factor` needs categorical data.

If `NULL`, then whenever the resulting colour function is called, the `x` value will represent the domain. This implies that if the function is invoked multiple times, the encoding between values and colours may not be consistent; if consistency is needed, you must provide a non-`NULL` domain.

na.color — The colour to return for `NA` values. Note that `na.color = NA` is valid.

alpha — Whether alpha channels should be respected or ignored. If `TRUE` then colors without explicit alpha information will be treated as fully opaque.

reverse — Whether the colors (or color function) in `palette` should be used in reverse order. For example, if the default order of a palette goes from blue to green, then `reverse = TRUE` will result in the colors going from green to blue.

`bins` 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 the domain values are to be cut.

`pretty` Whether to use the function `pretty()` to generate the bins when the argument `bins` is a single number. When `pretty = TRUE`, the actual number of bins may not be the number of bins you specified. When `pretty = FALSE`, `seq()` is used to generate the bins and the breaks may not be "pretty".

`right` parameter supplied to `base::cut()`. See Details

`n` Number of equal-size quantiles desired. For more precise control, use the `probs` argument instead.

`probs` See `stats::quantile()`. If provided, the `n` argument is ignored.

`levels` An alternate way of specifying levels; if specified, domain is ignored

`ordered` If `TRUE` and `domain` needs to be coerced to a factor, treat it as already in the correct order

Details

`col_numeric` is a simple linear mapping from continuous numeric data to an interpolated palette.

`col_bin` also maps continuous numeric data, but performs binning based on value (see the `base::cut()` function). `col_bin` defaults for the `cut` function are `include.lowest = TRUE` and `right = FALSE`.

`col_quantile` similarly bins numeric data, but via the `stats::quantile()` function.

`col_factor` maps factors to colours. If the palette is discrete and has a different number of colours than the number of factors, interpolation is used.

The `palette` argument can be any of the following:

1. A character vector of RGB or named colours. Examples: `palette()`, `c("#000000", "#0000FF", "#FFFFFF")`, `topo.colors(10)`
2. The name of an RColorBrewer palette, e.g. `"BuPu"` or `"Greens"`.
3. The full name of a viridis palette: `"viridis"`, `"magma"`, `"inferno"`, or `"plasma"`.
4. A function that receives a single value between 0 and 1 and returns a colour. Examples: `colorRamp(c("#000000", "#FFFFFF"), interpolate="spline")`.

Value

A function that takes a single parameter `x`; when called with a vector of numbers (except for `col_factor`, which expects factors/characters), #RRGGBB colour strings are returned (unless `alpha = TRUE` in which case #RRGGBBAA may also be possible).

Examples

```
pal <- col_bin("Greens", domain = 0:100)
show_col(pal(sort(runif(10, 60, 100))))

# Exponential distribution, mapped continuously
show_col(col_numeric("Blues", domain = NULL)(sort(rexp(16))))
# Exponential distribution, mapped by interval
```

```
show_col(col_bin("Blues", domain = NULL, bins = 4)(sort(rexp(16))))
# Exponential distribution, mapped by quantile
show_col(col_quantile("Blues", domain = NULL)(sort(rexp(16))))

# Categorical data; by default, the values being coloured span the gamut...
show_col(col_factor("RdYlBu", domain = NULL)(LETTERS[1:5]))
# ...unless the data is a factor, without droplevels...
show_col(col_factor("RdYlBu", domain = NULL)(factor(LETTERS[1:5], levels = LETTERS)))
# ...or the domain is stated explicitly.
show_col(col_factor("RdYlBu", levels = LETTERS)(LETTERS[1:5]))
```

`cscale` *Continuous scale*

Description

Continuous scale

Usage

```
cscale(x, palette, na.value = NA_real_, trans = transform_identity())
```

Arguments

`x` vector of continuous values to scale

`palette` palette to use.

Built in palettes: `pal_area`, `pal_brewer`, `pal_dichromat`, `pal_div_gradient`, `pal_gradient_n`, `pal_grey`, `pal_hue`, `pal_identity`, `pal_linetype`, `pal_manual`, `pal_rescale`, `pal_seq_gradient`, `pal_shape`, `pal_viridis`

`na.value` value to use for missing values

`trans` transformation object describing the how to transform the raw data prior to scaling. Defaults to the identity transformation which leaves the data unchanged.

Built in transformations: `transform_asinh`, `transform_asn`, `transform_atanh`, `transform_boxcox`, `transform_compose`, `transform_date`, `transform_exp`, `transform_hms`, `transform_identity`, `transform_log`, `transform_log10`, `transform_log1p`, `transform_log2`, `transform_logit`, `transform_modulus`, `transform_probability`, `transform_probit`, `transform_pseudo_log`, `transform_reciprocal`, `transform_reverse`, `transform_sqrt`, `transform_time`, `transform_timespan`, `transform_yj`.

Examples

```
with(mtcars, plot(disp, mpg, cex = cscale(hp, pal_rescale())))
with(mtcars, plot(disp, mpg, cex = cscale(hp, pal_rescale(),
  trans = transform_sqrt()
)))
with(mtcars, plot(disp, mpg, cex = cscale(hp, pal_area())))
with(mtcars, plot(disp, mpg,
  pch = 20, cex = 5,
  col = cscale(hp, pal_seq_gradient("grey80", "black"))
))
```

dscale *Discrete scale*

Description

Discrete scale

Usage

```
dscale(x, palette, na.value = NA)
```

Arguments

x	vector of discrete values to scale
palette	aesthetic palette to use
na.value	aesthetic to use for missing values

Examples

```
with(mtcars, plot(disp, mpg,
  pch = 20, cex = 3,
  col = dscale(factor(cyl), pal_brewer())
))
```

expand_range *Expand a range with a multiplicative or additive constant*

Description

Expand a range with a multiplicative or additive constant

Usage

```
expand_range(range, mul = 0, add = 0, zero_width = 1)
```

Arguments

range	range of data, numeric vector of length 2
mul	multiplicative constant
add	additive constant
zero_width	distance to use if range has zero width

`label_bytes` *Label bytes (1 kB, 2 MB, etc)*

Description

Scale bytes into human friendly units. Can use either SI units (e.g. kB = 1000 bytes) or binary units (e.g. kiB = 1024 bytes). See Units of Information on Wikipedia for more details.

Usage

```
label_bytes(units = "auto_si", accuracy = 1, scale = 1, ...)
```

Arguments

`units` Unit to use. Should either one of:

- "kB", "MB", "GB", "TB", "PB", "EB", "ZB", and "YB" for SI units (base 1000).
- "kiB", "MiB", "GiB", "TiB", "PiB", "EiB", "ZiB", and "YiB" for binary units (base 1024).
- `auto_si` or `auto_binary` to automatically pick the most appropriate unit for each value.

`accuracy` A number to round to. Use (e.g.) `0.01` to show 2 decimal places of precision. If `NULL`, the default, uses a heuristic that should ensure breaks have the minimum number of digits needed to show the difference between adjacent values.

Applied to rescaled data.

`scale` A scaling factor: `x` will be multiplied by `scale` before formatting. This is useful if the underlying data is very small or very large.

`...` Arguments passed on to `number`

`prefix` Additional text to display before the number. The suffix is applied to absolute value before `style_positive` and `style_negative` are processed so that `prefix = "$"` will yield (e.g.) `-$1` and `($1)`.

`suffix` Additional text to display after the number.

`big.mark` Character used between every 3 digits to separate thousands.

`decimal.mark` The character to be used to indicate the numeric decimal point.

`style_positive` A string that determines the style of positive numbers:

- `"none"` (the default): no change, e.g. `1`.
- `"plus"`: preceded by `+`, e.g. `+1`.
- `"space"`: preceded by a Unicode "figure space", i.e., a space equally as wide as a number or `+`. Compared to `"none"`, adding a figure space can ensure numbers remain properly aligned when they are left- or right-justified.

`style_negative` A string that determines the style of negative numbers:

- `"hyphen"` (the default): preceded by a standard hypen `-`, e.g. `-1`.

- "minus", uses a proper Unicode minus symbol. This is a typographical nicety that ensures - aligns with the horizontal bar of the the horizontal bar of +.
- "parens", wrapped in parentheses, e.g. (1).

scale_cut Named numeric vector that allows you to rescale large (or small) numbers and add a prefix. Built-in helpers include:

- cut_short_scale(): [10^3, 10^6) = K, [10^6, 10^9) = M, [10^9, 10^12) = B, [10^12, Inf) = T.
- cut_long_scale(): [10^3, 10^6) = K, [10^6, 10^12) = M, [10^12, 10^18) = B, [10^18, Inf) = T.
- cut_si(unit): uses standard SI units.

If you supply a vector c(a = 100, b = 1000), absolute values in the range [0, 100) will not be rescaled, absolute values in the range [100, 1000) will be divided by 100 and given the suffix "a", and absolute values in the range [1000, Inf) will be divided by 1000 and given the suffix "b". If the division creates an irrational value (or one with many digits), the cut value below will be tried to see if it improves the look of the final label.

trim Logical, if FALSE, values are right-justified to a common width (see base::format()).

Value

A labeller function that takes a numeric vector of breaks and returns a character vector of labels.

See Also

Other labels for continuous scales: label_currency(), label_number_auto(), label_number_si(), label_ordinal(), label_parse(), label_percent(), label_pvalue(), label_scientific()

Other labels for log scales: label_log(), label_number_si(), label_scientific()

Examples

```
demo_continuous(c(1, 1e6))
demo_continuous(c(1, 1e6), labels = label_bytes())

# Auto units are particularly nice on log scales
demo_log10(c(1, 1e7), labels = label_bytes())

# You can also set the units
demo_continuous(c(1, 1e6), labels = label_bytes("kB"))

# You can also use binary units where a megabyte is defined as
# (1024) ^ 2 bytes rather than (1000) ^ 2. You'll need to override
# the default breaks to make this more informative.
demo_continuous(c(1, 1024^2),
  breaks = breaks_width(250 * 1024),
  labels = label_bytes("auto_binary")
)
```

`label_currency` *Label currencies ($100, €2.50, etc)*

Description

Format numbers as currency, rounding values to monetary or fractional monetary using unit a convenient heuristic.

Usage

```
label_currency(
  accuracy = NULL,
  scale = 1,
  prefix = "$",
  suffix = "",
  big.mark = ",",
  decimal.mark = ".",
  trim = TRUE,
  largest_with_fractional = 1e+05,
  ...
)
```

Arguments

`accuracy, largest_with_fractional`
: Number to round to. If `NULL`, the default, values will be rounded to the nearest integer, unless any of the values has non-zero fractional component (e.g. cents) and the largest value is less than `largest_with_fractional` which by default is 100,000.

`scale`
: A scaling factor: `x` will be multiplied by `scale` before formatting. This is useful if the underlying data is very small or very large.

`prefix, suffix`
: Symbols to display before and after value.

`big.mark`
: Character used between every 3 digits to separate thousands.

`decimal.mark`
: The character to be used to indicate the numeric decimal point.

`trim`
: Logical, if `FALSE`, values are right-justified to a common width (see `base::format()`).

`...`
: Arguments passed on to `number`

  `style_positive` A string that determines the style of positive numbers:

  - `"none"` (the default): no change, e.g. `1`.
  - `"plus"`: preceded by `+`, e.g. `+1`.
  - `"space"`: preceded by a Unicode "figure space", i.e., a space equally as wide as a number or `+`. Compared to `"none"`, adding a figure space can ensure numbers remain properly aligned when they are left- or right-justified.

  `style_negative` A string that determines the style of negative numbers:

- "hyphen" (the default): preceded by a standard hypen -, e.g. -1.
- "minus", uses a proper Unicode minus symbol. This is a typographical nicety that ensures - aligns with the horizontal bar of the the horizontal bar of +.
- "parens", wrapped in parentheses, e.g. (1).

scale_cut Named numeric vector that allows you to rescale large (or small) numbers and add a prefix. Built-in helpers include:

- cut_short_scale(): [10^3, 10^6) = K, [10^6, 10^9) = M, [10^9, 10^12) = B, [10^12, Inf) = T.
- cut_long_scale(): [10^3, 10^6) = K, [10^6, 10^12) = M, [10^12, 10^18) = B, [10^18, Inf) = T.
- cut_si(unit): uses standard SI units.

If you supply a vector c(a = 100, b = 1000), absolute values in the range [0, 100) will not be rescaled, absolute values in the range [100, 1000) will be divided by 100 and given the suffix "a", and absolute values in the range [1000, Inf) will be divided by 1000 and given the suffix "b". If the division creates an irrational value (or one with many digits), the cut value below will be tried to see if it improves the look of the final label.

Value

All label_() functions return a "labelling" function, i.e. a function that takes a vector x and returns a character vector of length(x) giving a label for each input value.

Labelling functions are designed to be used with the labels argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

See Also

Other labels for continuous scales: label_bytes(), label_number_auto(), label_number_si(), label_ordinal(), label_parse(), label_percent(), label_pvalue(), label_scientific()

Examples

```
demo_continuous(c(0, 1), labels = label_currency())
demo_continuous(c(1, 100), labels = label_currency())

# Customise currency display with prefix and suffix
demo_continuous(c(1, 100), labels = label_currency(prefix = "USD "))
yen <- label_currency(
  prefix = "¥",
  suffix = "",
  big.mark = ".",
  decimal.mark = ","
)
demo_continuous(c(1000, 1100), labels = yen)

# Use style_negative = "parens" for finance style display
demo_continuous(c(-100, 100), labels = label_currency(style_negative = "parens"))
```

```
# Use scale_cut to use K/M/B where appropriate
demo_log10(c(1, 1e16),
  breaks = log_breaks(7, 1e3),
  labels = label_currency(scale_cut = cut_short_scale())
)
# cut_short_scale() uses B = one thousand million
# cut_long_scale() uses B = one million million
demo_log10(c(1, 1e16),
  breaks = log_breaks(7, 1e3),
  labels = label_currency(scale_cut = cut_long_scale())
)

# You can also define your own breaks
gbp <- label_currency(
  prefix = "\u00a3",
  scale_cut = c(0, k = 1e3, m = 1e6, bn = 1e9, tn = 1e12)
)
demo_log10(c(1, 1e12), breaks = log_breaks(5, 1e3), labels = gbp)
```

label_date *Label date/times*

Description

`label_date()` and `label_time()` label date/times using date/time format strings. `label_date_short()` automatically constructs a short format string sufficient to uniquely identify labels. It's inspired by matplotlib's `ConciseDateFormatter`, but uses a slightly different approach: `ConciseDateFormatter` formats "firsts" (e.g. first day of month, first day of day) specially; `date_short()` formats changes (e.g. new month, new year) specially. `label_timespan()` is intended to show time passed and adds common time units suffix to the input (ns, us, ms, s, m, h, d, w).

Usage

```
label_date(format = "%Y-%m-%d", tz = "UTC", locale = NULL)

label_date_short(format = c("%Y", "%b", "%d", "%H:%M"), sep = "\n")

label_time(format = "%H:%M:%S", tz = "UTC", locale = NULL)

label_timespan(
  unit = c("secs", "mins", "hours", "days", "weeks"),
  space = FALSE,
  ...
)
```

Arguments

`format` For `date_format()` and `time_format()` a date/time format string using standard POSIX specification. See `strptime()` for details.

For `date_short()` a character vector of length 4 giving the format components to use for year, month, day, and hour respectively.

`tz` a time zone name, see `timezones()`. Defaults to UTC

`locale` Locale to use when for day and month names. The default uses the current locale. Setting this argument requires stringi, and you can see a complete list of supported locales with `stringi::stri_locale_list()`.

`sep` Separator to use when combining date formats into a single string.

`unit` The unit used to interpret numeric input

`space` Add a space before the time unit?

`...` Arguments passed on to `number`

`accuracy` A number to round to. Use (e.g.) `0.01` to show 2 decimal places of precision. If `NULL`, the default, uses a heuristic that should ensure breaks have the minimum number of digits needed to show the difference between adjacent values.
Applied to rescaled data.

`scale` A scaling factor: `x` will be multiplied by `scale` before formatting. This is useful if the underlying data is very small or very large.

`prefix` Additional text to display before the number. The suffix is applied to absolute value before `style_positive` and `style_negative` are processed so that `prefix = "$"` will yield (e.g.) `-$1` and `($1)`.

`suffix` Additional text to display after the number.

`big.mark` Character used between every 3 digits to separate thousands.

`decimal.mark` The character to be used to indicate the numeric decimal point.

`style_positive` A string that determines the style of positive numbers:

- `"none"` (the default): no change, e.g. `1`.
- `"plus"`: preceded by `+`, e.g. `+1`.
- `"space"`: preceded by a Unicode "figure space", i.e., a space equally as wide as a number or `+`. Compared to `"none"`, adding a figure space can ensure numbers remain properly aligned when they are left- or right-justified.

`style_negative` A string that determines the style of negative numbers:

- `"hyphen"` (the default): preceded by a standard hypen `-`, e.g. `-1`.
- `"minus"`, uses a proper Unicode minus symbol. This is a typographical nicety that ensures `−` aligns with the horizontal bar of the the horizontal bar of `+`.
- `"parens"`, wrapped in parentheses, e.g. `(1)`.

`trim` Logical, if `FALSE`, values are right-justified to a common width (see `base::format()`).

Value

All `label_()` functions return a "labelling" function, i.e. a function that takes a vector `x` and returns a character vector of `length(x)` giving a label for each input value.

Labelling functions are designed to be used with the `labels` argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

Examples

```
date_range <- function(start, days) {
  start <- as.POSIXct(start)
  c(start, start + days * 24 * 60 * 60)
}

two_months <- date_range("2020-05-01", 60)
demo_datetime(two_months)
demo_datetime(two_months, labels = date_format("%m/%d"))
demo_datetime(two_months, labels = date_format("%e %b", locale = "fr"))
demo_datetime(two_months, labels = date_format("%e %B", locale = "es"))
# ggplot2 provides a short-hand:
demo_datetime(two_months, date_labels = "%m/%d")

# An alternative labelling system is label_date_short()
demo_datetime(two_months, date_breaks = "7 days", labels = label_date_short())
# This is particularly effective for dense labels
one_year <- date_range("2020-05-01", 365)
demo_datetime(one_year, date_breaks = "month")
demo_datetime(one_year, date_breaks = "month", labels = label_date_short())
```

label_log *Label numbers in log format (10^3, 10^6, etc)*

Description

`label_log()` displays numbers as base^exponent, using superscript formatting.

Usage

```
label_log(base = 10, digits = 3)
```

Arguments

`base`	Base of logarithm to use
`digits`	Number of significant digits to show for the exponent. Argument is passed on to `base::format()`.

Value

All `label_()` functions return a "labelling" function, i.e. a function that takes a vector `x` and returns a character vector of `length(x)` giving a label for each input value.

Labelling functions are designed to be used with the `labels` argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

See Also

`breaks_log()` for the related breaks algorithm.

Other labels for log scales: `label_bytes()`, `label_number_si()`, `label_scientific()`

Examples

```
demo_log10(c(1, 1e5), labels = label_log())
demo_log10(c(1, 1e5), breaks = breaks_log(base = 2), labels = label_log(base = 2))
```

`label_number` *Label numbers in decimal format (e.g. 0.12, 1,234)*

Description

Use `label_number()` force decimal display of numbers (i.e. don't use scientific notation). `label_comma()` is a special case that inserts a comma every three digits.

Usage

```
label_number(
  accuracy = NULL,
  scale = 1,
  prefix = "",
  suffix = "",
  big.mark = " ",
  decimal.mark = ".",
  style_positive = c("none", "plus", "space"),
  style_negative = c("hyphen", "minus", "parens"),
  scale_cut = NULL,
  trim = TRUE,
  ...
)

label_comma(
  accuracy = NULL,
  scale = 1,
  prefix = "",
  suffix = "",
  big.mark = ",",
  decimal.mark = ".",
  trim = TRUE,
  digits,
  ...
)
```

Arguments

`accuracy` A number to round to. Use (e.g.) `0.01` to show 2 decimal places of precision. If `NULL`, the default, uses a heuristic that should ensure breaks have the minimum number of digits needed to show the difference between adjacent values.

Applied to rescaled data.

`scale` A scaling factor: `x` will be multiplied by `scale` before formatting. This is useful if the underlying data is very small or very large.

`prefix` Additional text to display before the number. The suffix is applied to absolute value before `style_positive` and `style_negative` are processed so that `prefix = "$"` will yield (e.g.) `-$1` and `($1)`.

`suffix` Additional text to display after the number.

`big.mark` Character used between every 3 digits to separate thousands.

`decimal.mark` The character to be used to indicate the numeric decimal point.

`style_positive` A string that determines the style of positive numbers:

- `"none"` (the default): no change, e.g. `1`.
- `"plus"`: preceded by `+`, e.g. `+1`.
- `"space"`: preceded by a Unicode "figure space", i.e., a space equally as wide as a number or `+`. Compared to `"none"`, adding a figure space can ensure numbers remain properly aligned when they are left- or right-justified.

`style_negative` A string that determines the style of negative numbers:

- `"hyphen"` (the default): preceded by a standard hypen `-`, e.g. `-1`.
- `"minus"`, uses a proper Unicode minus symbol. This is a typographical nicety that ensures `−` aligns with the horizontal bar of the the horizontal bar of `+`.
- `"parens"`, wrapped in parentheses, e.g. `(1)`.

`scale_cut` Named numeric vector that allows you to rescale large (or small) numbers and add a prefix. Built-in helpers include:

- `cut_short_scale()`: [10^3, 10^6) = K, [10^6, 10^9) = M, [10^9, 10^12) = B, [10^12, Inf) = T.
- `cut_long_scale()`: [10^3, 10^6) = K, [10^6, 10^12) = M, [10^12, 10^18) = B, [10^18, Inf) = T.
- `cut_si(unit)`: uses standard SI units.

If you supply a vector `c(a = 100, b = 1000)`, absolute values in the range `[0, 100)` will not be rescaled, absolute values in the range `[100, 1000)` will be divided by 100 and given the suffix "a", and absolute values in the range `[1000, Inf)` will be divided by 1000 and given the suffix "b". If the division creates an irrational value (or one with many digits), the cut value below will be tried to see if it improves the look of the final label.

`trim` Logical, if `FALSE`, values are right-justified to a common width (see `base::format()`).

`...` Other arguments passed on to `base::format()`.

`digits` **[Deprecated]** Use `accuracy` instead.

Value

All `label_()` functions return a "labelling" function, i.e. a function that takes a vector `x` and returns a character vector of `length(x)` giving a label for each input value.

Labelling functions are designed to be used with the `labels` argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

Examples

```
demo_continuous(c(-1e6, 1e6))
demo_continuous(c(-1e6, 1e6), labels = label_number())
demo_continuous(c(-1e6, 1e6), labels = label_comma())

# Use scale to rescale very small or large numbers to generate
# more readable labels
demo_continuous(c(0, 1e6), labels = label_number())
demo_continuous(c(0, 1e6), labels = label_number(scale = 1 / 1e3))
demo_continuous(c(0, 1e-6), labels = label_number())
demo_continuous(c(0, 1e-6), labels = label_number(scale = 1e6))

#' Use scale_cut to automatically add prefixes for large/small numbers
demo_log10(
  c(1, 1e9),
  breaks = log_breaks(10),
  labels = label_number(scale_cut = cut_short_scale())
)
demo_log10(
  c(1, 1e9),
  breaks = log_breaks(10),
  labels = label_number(scale_cut = cut_si("m"))
)
demo_log10(
  c(1e-9, 1),
  breaks = log_breaks(10),
  labels = label_number(scale_cut = cut_si("g"))
)
# use scale and scale_cut when data already uses SI prefix
# for example, if data was stored in kg
demo_log10(
  c(1e-9, 1),
  breaks = log_breaks(10),
  labels = label_number(scale_cut = cut_si("g"), scale = 1e3)
)

#' # Use style arguments to vary the appearance of positive and negative numbers
```

```
demo_continuous(c(-1e3, 1e3), labels = label_number(
  style_positive = "plus",
  style_negative = "minus"
))
demo_continuous(c(-1e3, 1e3), labels = label_number(style_negative = "parens"))

# You can use prefix and suffix for other types of display
demo_continuous(c(32, 212), labels = label_number(suffix = "\u00b0F"))
demo_continuous(c(0, 100), labels = label_number(suffix = "\u00b0C"))
```

label_number_auto *Label numbers, avoiding scientific notation where possible*

Description

Switches between number_format() and scientific_format() based on a set of heuristics designed to automatically generate useful labels across a wide range of inputs

Usage

```
label_number_auto()
```

See Also

Other labels for continuous scales: label_bytes(), label_currency(), label_number_si(), label_ordinal(), label_parse(), label_percent(), label_pvalue(), label_scientific()

Examples

```
# Very small and very large numbers get scientific notation
demo_continuous(c(0, 1e-6), labels = label_number_auto())
demo_continuous(c(0, 1e9), labels = label_number_auto())

# Other ranges get the numbers printed in full
demo_continuous(c(0, 1e-3), labels = label_number_auto())
demo_continuous(c(0, 1), labels = label_number_auto())
demo_continuous(c(0, 1e3), labels = label_number_auto())
demo_continuous(c(0, 1e6), labels = label_number_auto())

# Transformation is applied individually so you get as little
# scientific notation as possible
demo_log10(c(1, 1e7), labels = label_number_auto())
```

label_ordinal *Label ordinal numbers (1st, 2nd, 3rd, etc)*

Description

Round values to integers and then display as ordinal values (e.g. 1st, 2nd, 3rd). Built-in rules are provided for English, French, and Spanish.

Usage

```
label_ordinal(
  prefix = "",
  suffix = "",
  big.mark = " ",
  rules = ordinal_english(),
  ...
)

ordinal_english()

ordinal_french(gender = c("masculin", "feminin"), plural = FALSE)

ordinal_spanish()
```

Arguments

`prefix, suffix` Symbols to display before and after value.

`big.mark` Character used between every 3 digits to separate thousands.

`rules` Named list of regular expressions, matched in order. Name gives suffix, and value specifies which numbers to match.

`...` Arguments passed on to `number`

`accuracy` A number to round to. Use (e.g.) `0.01` to show 2 decimal places of precision. If `NULL`, the default, uses a heuristic that should ensure breaks have the minimum number of digits needed to show the difference between adjacent values.

Applied to rescaled data.

`scale` A scaling factor: `x` will be multiplied by `scale` before formatting. This is useful if the underlying data is very small or very large.

`decimal.mark` The character to be used to indicate the numeric decimal point.

`style_positive` A string that determines the style of positive numbers:

- `"none"` (the default): no change, e.g. `1`.
- `"plus"`: preceded by `+`, e.g. `+1`.
- `"space"`: preceded by a Unicode "figure space", i.e., a space equally as wide as a number or `+`. Compared to `"none"`, adding a figure space can ensure numbers remain properly aligned when they are left- or right-justified.

`style_negative` A string that determines the style of negative numbers:

- `"hyphen"` (the default): preceded by a standard hypen `-`, e.g. `-1`.
- `"minus"`, uses a proper Unicode minus symbol. This is a typographical nicety that ensures `-` aligns with the horizontal bar of the the horizontal bar of `+`.
- `"parens"`, wrapped in parentheses, e.g. `(1)`.

`scale_cut` Named numeric vector that allows you to rescale large (or small) numbers and add a prefix. Built-in helpers include:

- `cut_short_scale()`: [10^3, 10^6) = K, [10^6, 10^9) = M, [10^9, 10^12) = B, [10^12, Inf) = T.
- `cut_long_scale()`: [10^3, 10^6) = K, [10^6, 10^12) = M, [10^12, 10^18) = B, [10^18, Inf) = T.
- `cut_si(unit)`: uses standard SI units.

If you supply a vector `c(a = 100, b = 1000)`, absolute values in the range `[0, 100)` will not be rescaled, absolute values in the range `[100, 1000)` will be divided by 100 and given the suffix "a", and absolute values in the range `[1000, Inf)` will be divided by 1000 and given the suffix "b". If the division creates an irrational value (or one with many digits), the cut value below will be tried to see if it improves the look of the final label.

`trim` Logical, if `FALSE`, values are right-justified to a common width (see `base::format()`).

`gender` Masculin or feminin gender for French ordinal.

`plural` Plural or singular for French ordinal.

Value

All `label_()` functions return a "labelling" function, i.e. a function that takes a vector `x` and returns a character vector of `length(x)` giving a label for each input value.

Labelling functions are designed to be used with the `labels` argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

See Also

Other labels for continuous scales: `label_bytes()`, `label_currency()`, `label_number_auto()`, `label_number_si()`, `label_parse()`, `label_percent()`, `label_pvalue()`, `label_scientific()`

Examples

```
demo_continuous(c(1, 5))
demo_continuous(c(1, 5), labels = label_ordinal())
demo_continuous(c(1, 5), labels = label_ordinal(rules = ordinal_french()))

# The rules are just a set of regular expressions that are applied in turn
ordinal_french()
ordinal_english()

# Note that ordinal rounds values, so you may need to adjust the breaks too
demo_continuous(c(1, 10))
demo_continuous(c(1, 10), labels = label_ordinal())
demo_continuous(c(1, 10),
  labels = label_ordinal(),
  breaks = breaks_width(2)
)
```

---

label_parse *Label with mathematical annotations*

Description

label_parse() produces expression from strings by parsing them; label_math() constructs expressions by replacing the pronoun .x with each string.

Usage

```
label_parse()

label_math(expr = 10^.x, format = force)
```

Arguments

expr	expression to use
format	another format function to apply prior to mathematical transformation - this makes it easier to use floating point numbers in mathematical expressions.

Value

All label_() functions return a "labelling" function, i.e. a function that takes a vector x and returns a character vector of length(x) giving a label for each input value.

Labelling functions are designed to be used with the labels argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

See Also

plotmath for the details of mathematical formatting in R.

Other labels for continuous scales: label_bytes(), label_currency(), label_number_auto(), label_number_si(), label_ordinal(), label_percent(), label_pvalue(), label_scientific()

Other labels for discrete scales: label_wrap()

Examples

```
# Use label_parse() with discrete scales
greek <- c("alpha", "beta", "gamma")
demo_discrete(greek)
demo_discrete(greek, labels = label_parse())

# Use label_math() with continuous scales
demo_continuous(c(1, 5))
demo_continuous(c(1, 5), labels = label_math(alpha[.x]))
demo_continuous(c(1, 5), labels = label_math())
```

---

label_percent *Label percentages (2.5%, 50%, etc)*

Description

Label percentages (2.5%, 50%, etc)

Usage

```
label_percent(
  accuracy = NULL,
  scale = 100,
  prefix = "",
  suffix = "%",
  big.mark = " ",
  decimal.mark = ".",
  trim = TRUE,
  ...
)
```

Arguments

accuracy	A number to round to. Use (e.g.) 0.01 to show 2 decimal places of precision. If NULL, the default, uses a heuristic that should ensure breaks have the minimum number of digits needed to show the difference between adjacent values. Applied to rescaled data.
scale	A scaling factor: x will be multiplied by scale before formatting. This is useful if the underlying data is very small or very large.
prefix	Additional text to display before the number. The suffix is applied to absolute value before style_positive and style_negative are processed so that prefix = "$" will yield (e.g.) -$1 and ($1).
suffix	Additional text to display after the number.
big.mark	Character used between every 3 digits to separate thousands.
decimal.mark	The character to be used to indicate the numeric decimal point.

`trim` Logical, if `FALSE`, values are right-justified to a common width (see `base::format()`).

`...` Arguments passed on to `label_number`

`style_positive` A string that determines the style of positive numbers:

- `"none"` (the default): no change, e.g. `1`.
- `"plus"`: preceded by `+`, e.g. `+1`.
- `"space"`: preceded by a Unicode "figure space", i.e., a space equally as wide as a number or `+`. Compared to `"none"`, adding a figure space can ensure numbers remain properly aligned when they are left- or right-justified.

`style_negative` A string that determines the style of negative numbers:

- `"hyphen"` (the default): preceded by a standard hypen `-`, e.g. `-1`.
- `"minus"`, uses a proper Unicode minus symbol. This is a typographical nicety that ensures `-` aligns with the horizontal bar of the the horizontal bar of `+`.
- `"parens"`, wrapped in parentheses, e.g. `(1)`.

`scale_cut` Named numeric vector that allows you to rescale large (or small) numbers and add a prefix. Built-in helpers include:

- `cut_short_scale()`: [10^3, 10^6) = K, [10^6, 10^9) = M, [10^9, 10^12) = B, [10^12, Inf) = T.
- `cut_long_scale()`: [10^3, 10^6) = K, [10^6, 10^12) = M, [10^12, 10^18) = B, [10^18, Inf) = T.
- `cut_si(unit)`: uses standard SI units.

If you supply a vector `c(a = 100, b = 1000)`, absolute values in the range `[0, 100)` will not be rescaled, absolute values in the range `[100, 1000)` will be divided by 100 and given the suffix "a", and absolute values in the range `[1000, Inf)` will be divided by 1000 and given the suffix "b". If the division creates an irrational value (or one with many digits), the cut value below will be tried to see if it improves the look of the final label.

Value

All `label_()` functions return a "labelling" function, i.e. a function that takes a vector `x` and returns a character vector of `length(x)` giving a label for each input value.

Labelling functions are designed to be used with the `labels` argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

See Also

Other labels for continuous scales: `label_bytes()`, `label_currency()`, `label_number_auto()`, `label_number_si()`, `label_ordinal()`, `label_parse()`, `label_pvalue()`, `label_scientific()`

Examples

```
demo_continuous(c(0, 1))
demo_continuous(c(0, 1), labels = label_percent())
```

```
# Use prefix and suffix to create your own variants
french_percent <- label_percent(
  decimal.mark = ",",
  suffix = " %"
)
demo_continuous(c(0, .01), labels = french_percent)
```

---

label_pvalue *Label p-values (e.g. <0.001, 0.25, p >= 0.99)*

Description

Formatter for p-values, using "<" and ">" for p-values close to 0 and 1.

Usage

```
label_pvalue(
  accuracy = 0.001,
  decimal.mark = ".",
  prefix = NULL,
  add_p = FALSE
)
```

Arguments

accuracy	A number to round to. Use (e.g.) 0.01 to show 2 decimal places of precision. If NULL, the default, uses a heuristic that should ensure breaks have the minimum number of digits needed to show the difference between adjacent values. Applied to rescaled data.
decimal.mark	The character to be used to indicate the numeric decimal point.
prefix	A character vector of length 3 giving the prefixes to put in front of numbers. The default values are c("<", "", ">") if add_p is TRUE and c("p<", "p=", "p>") if FALSE.
add_p	Add "p=" before the value?

Value

All label_() functions return a "labelling" function, i.e. a function that takes a vector x and returns a character vector of length(x) giving a label for each input value.

Labelling functions are designed to be used with the labels argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

See Also

Other labels for continuous scales: label_bytes(), label_currency(), label_number_auto(), label_number_si(), label_ordinal(), label_parse(), label_percent(), label_scientific()

Examples

```
demo_continuous(c(0, 1))
demo_continuous(c(0, 1), labels = label_pvalue())
demo_continuous(c(0, 1), labels = label_pvalue(accuracy = 0.1))
demo_continuous(c(0, 1), labels = label_pvalue(add_p = TRUE))

# Or provide your own prefixes
prefix <- c("p < ", "p = ", "p > ")
demo_continuous(c(0, 1), labels = label_pvalue(prefix = prefix))
```

`label_scientific` *Label numbers with scientific notation (e.g. 1e05, 1.5e-02)*

Description

Label numbers with scientific notation (e.g. 1e05, 1.5e-02)

Usage

```
label_scientific(
  digits = 3,
  scale = 1,
  prefix = "",
  suffix = "",
  decimal.mark = ".",
  trim = TRUE,
  ...
)
```

Arguments

`digits` Number of digits to show before exponent.

`scale` A scaling factor: `x` will be multiplied by `scale` before formatting. This is useful if the underlying data is very small or very large.

`prefix, suffix` Symbols to display before and after value.

`decimal.mark` The character to be used to indicate the numeric decimal point.

`trim` Logical, if `FALSE`, values are right-justified to a common width (see `base::format()`).

`...` Other arguments passed on to `base::format()`.

Value

All `label_()` functions return a "labelling" function, i.e. a function that takes a vector `x` and returns a character vector of `length(x)` giving a label for each input value.

Labelling functions are designed to be used with the `labels` argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

See Also

Other labels for continuous scales: label_bytes(), label_currency(), label_number_auto(), label_number_si(), label_ordinal(), label_parse(), label_percent(), label_pvalue()

Other labels for log scales: label_bytes(), label_log(), label_number_si()

Examples

```
demo_continuous(c(1, 10))
demo_continuous(c(1, 10), labels = label_scientific())
demo_continuous(c(1, 10), labels = label_scientific(digits = 3))

demo_log10(c(1, 1e9))
```

label_wrap *Label strings by wrapping across multiple lines*

Description

Uses strwrap() to split long labels across multiple lines.

Usage

```
label_wrap(width)
```

Arguments

width	Number of characters per line.

Value

All label_() functions return a "labelling" function, i.e. a function that takes a vector x and returns a character vector of length(x) giving a label for each input value.

Labelling functions are designed to be used with the labels argument of ggplot2 scales. The examples demonstrate their use with x scales, but they work similarly for all scales, including those that generate legends rather than axes.

See Also

Other labels for discrete scales: label_parse()

Examples

```
x <- c(
  "this is a long label",
  "this is another long label",
  "this a label this is even longer"
)
demo_discrete(x)
demo_discrete(x, labels = label_wrap(10))
demo_discrete(x, labels = label_wrap(20))
```

`minor_breaks_width` *Minor breaks*

Description

Generate minor breaks between major breaks either spaced with a fixed width, or having a fixed number.

Usage

```
minor_breaks_width(width, offset)

minor_breaks_n(n)
```

Arguments

`width` Distance between each break. Either a number, or for date/times, a single string of the form `"{n} {unit}"`, e.g. "1 month", "5 days". Unit can be of one "sec", "min", "hour", "day", "week", "month", "year".

`offset` Use if you don't want breaks to start at zero, or on a conventional date or time boundary such as the 1st of January or midnight. Either a number, or for date/times, a single string of the form `"{n} {unit}"`, as for `width`.

`offset` can be a vector, which will accumulate in the order given. This is mostly useful for dates, where e.g. `c("3 months", "5 days")` will offset by three months and five days, which is useful for the UK tax year. Note that due to way that dates are rounded, there's no guarantee that `offset = c(x, y)` will give the same result as `offset = c(y, x)`.

`n` number of breaks

Examples

```
demo_log10(c(1, 1e6))
if (FALSE) {
  # Requires https://github.com/tidyverse/ggplot2/pull/3591
  demo_log10(c(1, 1e6), minor_breaks = minor_breaks_n(10))
}
```

muted *Mute standard colour*

Description

Mute standard colour

Usage

```
muted(colour, l = 30, c = 70)
```

Arguments

colour	character vector of colours to modify
l	new luminance
c	new chroma

Examples

```
muted("red")
muted("blue")
show_col(c("red", "blue", muted("red"), muted("blue")))
```

oob *Out of bounds handling*

Description

This set of functions modify data values outside a given range. The `oob_*()` functions are designed to be passed as the `oob` argument of ggplot2 continuous and binned scales, with `oob_discard` being an exception.

These functions affect out of bounds values in the following ways:

- `oob_censor()` replaces out of bounds values with `NA`s. This is the default `oob` argument for continuous scales.
- `oob_censor_any()` acts like `oob_censor()`, but also replaces infinite values with `NA`s.
- `oob_squish()` replaces out of bounds values with the nearest limit. This is the default `oob` argument for binned scales.
- `oob_squish_any()` acts like `oob_squish()`, but also replaces infinite values with the nearest limit.
- `oob_squish_infinite()` only replaces infinite values by the nearest limit.
- `oob_keep()` does not adjust out of bounds values. In position scales, behaves as zooming limits without data removal.
- `oob_discard()` removes out of bounds values from the input. Not suitable for ggplot2 scales.

Usage

```
oob_censor(x, range = c(0, 1), only.finite = TRUE)

oob_censor_any(x, range = c(0, 1))

oob_discard(x, range = c(0, 1))

oob_squish(x, range = c(0, 1), only.finite = TRUE)

oob_squish_any(x, range = c(0, 1))

oob_squish_infinite(x, range = c(0, 1))

oob_keep(x, range = c(0, 1))

censor(x, range = c(0, 1), only.finite = TRUE)

discard(x, range = c(0, 1))

squish(x, range = c(0, 1), only.finite = TRUE)

squish_infinite(x, range = c(0, 1))
```

Arguments

x	A numeric vector of values to modify.
range	A numeric vector of length two giving the minimum and maximum limit of the desired output range respectively.
only.finite	A logical of length one. When TRUE, only finite values are altered. When FALSE, also infinite values are altered.

Details

The oob_censor_any() and oob_squish_any() functions are the same as oob_censor() and oob_squish() with the only.finite argument set to FALSE.

Replacing position values with NAs, as oob_censor() does, will typically lead to removal of those datapoints in ggplot.

Setting ggplot coordinate limits is equivalent to using oob_keep() in position scales.

Value

Most oob_() functions return a vector of numerical values of the same length as the x argument, wherein out of bounds values have been modified. Only oob_discard() returns a vector of less than or of equal length to the x argument.

Old interface

censor(), squish(), squish_infinite() and discard() are no longer recommended; please use oob_censor(), oob_squish(), oob_squish_infinite() and oob_discard() instead.

Author(s)

oob_squish(): Homer Strong homer.strong@gmail.com

Examples

```
# Censoring replaces out of bounds values with NAs
oob_censor(c(-Inf, -1, 0.5, 1, 2, NA, Inf))
oob_censor_any(c(-Inf, -1, 0.5, 1, 2, NA, Inf))

# Squishing replaces out of bounds values with the nearest range limit
oob_squish(c(-Inf, -1, 0.5, 1, 2, NA, Inf))
oob_squish_any(c(-Inf, -1, 0.5, 1, 2, NA, Inf))
oob_squish_infinite(c(-Inf, -1, 0.5, 1, 2, NA, Inf))

# Keeping does not alter values
oob_keep(c(-Inf, -1, 0.5, 1, 2, NA, Inf))

# Discarding will remove out of bounds values
oob_discard(c(-Inf, -1, 0.5, 1, 2, NA, Inf))
```

pal_area *Area palettes (continuous)*

Description

Area palettes (continuous)

Usage

```
pal_area(range = c(1, 6))

area_pal(range = c(1, 6))

abs_area(max)
```

Arguments

range	Numeric vector of length two, giving range of possible sizes. Should be greater than 0.
max	A number representing the maximum size.

`pal_brewer` *Colour Brewer palette (discrete)*

Description

Colour Brewer palette (discrete)

Usage

```
pal_brewer(type = "seq", palette = 1, direction = 1)

brewer_pal(type = "seq", palette = 1, direction = 1)
```

Arguments

`type` One of "seq" (sequential), "div" (diverging) or "qual" (qualitative)

`palette` If a string, will use that named palette. If a number, will index into the list of palettes of appropriate `type`

`direction` Sets the order of colours in the scale. If 1, the default, colours are as output by `RColorBrewer::brewer.pal()`. If -1, the order of colours is reversed.

References

https://colorbrewer2.org

Examples

```
show_col(pal_brewer()(10))
show_col(pal_brewer("div")(5))
show_col(pal_brewer(palette = "Greens")(5))

# Can use with gradient_n to create a continuous gradient
cols <- pal_brewer("div")(5)
show_col(pal_gradient_n(cols)(seq(0, 1, length.out = 30)))
```

`pal_dichromat` *Dichromat (colour-blind) palette (discrete)*

Description

Dichromat (colour-blind) palette (discrete)

Usage

```
pal_dichromat(name)

dichromat_pal(name)
```

Arguments

`name` Name of colour palette. One of: `BrowntoBlue.10`, `BrowntoBlue.12`, `BluetoDarkOrange.12`, `BluetoDarkOrange.18`, `DarkRedtoBlue.12`, `DarkRedtoBlue.18`, `BluetoGreen.14`, `BluetoGray.8`, `BluetoOrangeRed.14`, `BluetoOrange.10`, `BluetoOrange.12`, `BluetoOrange.8`, `LightBluetoDarkBlue.10`, `LightBluetoDarkBlue.7`, `Categorical.12`, `GreentoMagenta.16`, `SteppedSequential.5`

Examples

```
if (requireNamespace("dichromat", quietly = TRUE)) {
  show_col(pal_dichromat("BluetoOrange.10")(10))
  show_col(pal_dichromat("BluetoOrange.10")(5))

  # Can use with gradient_n to create a continous gradient
  cols <- pal_dichromat("DarkRedtoBlue.12")(12)
  show_col(pal_gradient_n(cols)(seq(0, 1, length.out = 30)))
}
```

`pal_div_gradient` *Diverging colour gradient (continuous).*

Description

Diverging colour gradient (continuous).

Usage

```
pal_div_gradient(
  low = mnsl("10B 4/6"),
  mid = mnsl("N 8/0"),
  high = mnsl("10R 4/6"),
  space = "Lab"
)

div_gradient_pal(
  low = mnsl("10B 4/6"),
  mid = mnsl("N 8/0"),
  high = mnsl("10R 4/6"),
  space = "Lab"
)
```

Arguments

`low` colour for low end of gradient.

`mid` colour for mid point

`high` colour for high end of gradient.

`space` colour space in which to calculate gradient. Must be "Lab" - other values are deprecated.

Examples

```
x <- seq(-1, 1, length.out = 100)
r <- sqrt(outer(x^2, x^2, "+"))
image(r, col = pal_div_gradient()(seq(0, 1, length.out = 12)))
image(r, col = pal_div_gradient()(seq(0, 1, length.out = 30)))
image(r, col = pal_div_gradient()(seq(0, 1, length.out = 100)))

library(munsell)
pal <- pal_div_gradient(low = mnsl(complement("10R 4/6"), fix = TRUE))
image(r, col = pal(seq(0, 1, length.out = 100)))
```

---

pal_gradient_n *Arbitrary colour gradient palette (continuous)*

Description

Arbitrary colour gradient palette (continuous)

Usage

```
pal_gradient_n(colours, values = NULL, space = "Lab")

gradient_n_pal(colours, values = NULL, space = "Lab")
```

Arguments

colours	vector of colours
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.

---

pal_grey *Grey scale palette (discrete)*

Description

Grey scale palette (discrete)

Usage

```
pal_grey(start = 0.2, end = 0.8)

grey_pal(start = 0.2, end = 0.8)
```

Arguments

`start`	grey value at low end of palette
`end`	grey value at high end of palette

See Also

`pal_seq_gradient()` for continuous version

Examples

```
show_col(pal_grey()(25))
show_col(pal_grey(0, 1)(25))
```

`pal_hue` *Hue palette (discrete)*

Description

Hue palette (discrete)

Usage

```
pal_hue(h = c(0, 360) + 15, c = 100, l = 65, h.start = 0, direction = 1)

hue_pal(h = c(0, 360) + 15, c = 100, l = 65, h.start = 0, direction = 1)
```

Arguments

`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

Examples

```
show_col(pal_hue()(4))
show_col(pal_hue()(9))
show_col(pal_hue(l = 90)(9))
show_col(pal_hue(l = 30)(9))

show_col(pal_hue()(9))
show_col(pal_hue(direction = -1)(9))
show_col(pal_hue(h.start = 30)(9))
show_col(pal_hue(h.start = 90)(9))
show_col(pal_hue()(9))
show_col(pal_hue(h = c(0, 90))(9))
show_col(pal_hue(h = c(90, 180))(9))
show_col(pal_hue(h = c(180, 270))(9))
show_col(pal_hue(h = c(270, 360))(9))
```

`pal_identity` *Identity palette*

Description

Leaves values unchanged - useful when the data is already scaled.

Usage

```
pal_identity()

identity_pal()
```

`pal_linetype` *Line type palette (discrete)*

Description

Based on a set supplied by Richard Pearson, University of Manchester

Usage

```
pal_linetype()

linetype_pal()
```

`pal_manual` *Manual palette (discrete)*

Description

Manual palette (discrete)

Usage

```
pal_manual(values)

manual_pal(values)
```

Arguments

`values`	vector of values to be used as a palette.

pal_rescale *Rescale palette (continuous)*

Description

Just rescales the input to the specific output range. Useful for alpha, size, and continuous position.

Usage

```
pal_rescale(range = c(0.1, 1))

rescale_pal(range = c(0.1, 1))
```

Arguments

range	Numeric vector of length two, giving range of possible values. Should be between 0 and 1.

pal_seq_gradient *Sequential colour gradient palette (continuous)*

Description

Sequential colour gradient palette (continuous)

Usage

```
pal_seq_gradient(low = mnsl("10B 4/6"), high = mnsl("10R 4/6"), space = "Lab")

seq_gradient_pal(low = mnsl("10B 4/6"), high = mnsl("10R 4/6"), space = "Lab")
```

Arguments

low	colour for low end of gradient.
high	colour for high end of gradient.
space	colour space in which to calculate gradient. Must be "Lab" - other values are deprecated.

Examples

```
x <- seq(0, 1, length.out = 25)
show_col(pal_seq_gradient()(x))
show_col(pal_seq_gradient("white", "black")(x))

library(munsell)
show_col(pal_seq_gradient("white", mnsl("10R 4/6"))(x))
```

`pal_shape` *Shape palette (discrete)*

Description

Shape palette (discrete)

Usage

```
pal_shape(solid = TRUE)

shape_pal(solid = TRUE)
```

Arguments

`solid`	should shapes be solid or not?

`pal_viridis` *Viridis palette*

Description

Viridis palette

Usage

```
pal_viridis(alpha = 1, begin = 0, end = 1, direction = 1, option = "D")

viridis_pal(alpha = 1, begin = 0, end = 1, direction = 1, option = "D")
```

Arguments

`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"`)

References

https://bids.github.io/colormap/

Examples

```
show_col(pal_viridis()(10))
show_col(pal_viridis(direction = -1)(6))
show_col(pal_viridis(begin = 0.2, end = 0.8)(4))
show_col(pal_viridis(option = "plasma")(6))
```

Range *Mutable ranges*

Description

Mutable ranges have a two methods (`train` and `reset`), and make it possible to build up complete ranges with multiple passes.

rescale *Rescale continuous vector to have specified minimum and maximum*

Description

Rescale continuous vector to have specified minimum and maximum

Usage

```
rescale(x, to, from, ...)

## S3 method for class 'numeric'
rescale(x, to = c(0, 1), from = range(x, na.rm = TRUE, finite = TRUE), ...)

## S3 method for class 'dist'
rescale(x, to = c(0, 1), from = range(x, na.rm = TRUE, finite = TRUE), ...)

## S3 method for class 'logical'
rescale(x, to = c(0, 1), from = range(x, na.rm = TRUE, finite = TRUE), ...)

## S3 method for class 'POSIXt'
rescale(x, to = c(0, 1), from = range(x, na.rm = TRUE, finite = TRUE), ...)

## S3 method for class 'Date'
rescale(x, to = c(0, 1), from = range(x, na.rm = TRUE, finite = TRUE), ...)

## S3 method for class 'integer64'
```

```
rescale(x, to = c(0, 1), from = range(x, na.rm = TRUE), ...)

## S3 method for class 'difftime'
rescale(x, to = c(0, 1), from = range(x, na.rm = TRUE, finite = TRUE), ...)

## S3 method for class 'AsIs'
rescale(x, to, from, ...)
```

Arguments

`x`	continuous vector of values to manipulate.
`to`	output range (numeric vector of length two)
`from`	input range (vector of length two). If not given, is calculated from the range of `x`
`...`	other arguments passed on to methods

Details

Objects of class `<AsIs>` are returned unaltered.

Examples

```
rescale(1:100)
rescale(runif(50))
rescale(1)
```

`rescale_max` *Rescale numeric vector to have specified maximum*

Description

Rescale numeric vector to have specified maximum

Usage

```
rescale_max(x, to = c(0, 1), from = range(x, na.rm = TRUE))
```

Arguments

`x`	numeric vector of values to manipulate.
`to`	output range (numeric vector of length two)
`from`	input range (numeric vector of length two). If not given, is calculated from the range of `x`

Examples

```
rescale_max(1:100)
rescale_max(runif(50))
rescale_max(1)
```

rescale_mid *Rescale vector to have specified minimum, midpoint, and maximum*

Description

Rescale vector to have specified minimum, midpoint, and maximum

Usage

```
rescale_mid(x, to, from, mid, ...)

## S3 method for class 'numeric'
rescale_mid(x, to = c(0, 1), from = range(x, na.rm = TRUE), mid = 0, ...)

## S3 method for class 'logical'
rescale_mid(x, to = c(0, 1), from = range(x, na.rm = TRUE), mid = 0, ...)

## S3 method for class 'dist'
rescale_mid(x, to = c(0, 1), from = range(x, na.rm = TRUE), mid = 0, ...)

## S3 method for class 'POSIXt'
rescale_mid(x, to = c(0, 1), from = range(x, na.rm = TRUE), mid, ...)

## S3 method for class 'Date'
rescale_mid(x, to = c(0, 1), from = range(x, na.rm = TRUE), mid, ...)

## S3 method for class 'integer64'
rescale_mid(x, to = c(0, 1), from = range(x, na.rm = TRUE), mid = 0, ...)

## S3 method for class 'AsIs'
rescale_mid(x, to, from, ...)
```

Arguments

`x`	vector of values to manipulate.
`to`	output range (numeric vector of length two)
`from`	input range (vector of length two). If not given, is calculated from the range of `x`
`mid`	mid-point of input range
`...`	other arguments passed on to methods

Details

Objects of class `<AsIs>` are returned unaltered.

Examples

```
rescale_mid(1:100, mid = 50.5)
rescale_mid(runif(50), mid = 0.5)
rescale_mid(1)
```

`rescale_none` *Don't perform rescaling*

Description

Don't perform rescaling

Usage

```
rescale_none(x, ...)
```

Arguments

`x`	numeric vector of values to manipulate.
`...`	all other arguments ignored

Examples

```
rescale_none(1:100)
```

`train_continuous` *Train (update) a continuous scale*

Description

Strips attributes and always returns a numeric vector

Usage

```
train_continuous(new, existing = NULL)
```

Arguments

`new`	New data to add to scale
`existing`	Optional existing scale to update

train_discrete *Train (update) a discrete scale*

Description

Train (update) a discrete scale

Usage

```
train_discrete(new, existing = NULL, drop = FALSE, na.rm = FALSE, fct = NA)
```

Arguments

`new`	New data to add to scale
`existing`	Optional existing scale to update
`drop`	`TRUE`, will drop factor levels not associated with data
`na.rm`	If `TRUE`, will remove missing values
`fct`	Treat `existing` as if it came from a factor (ie. don't sort the range)

transform_asinh *Inverse Hyperbolic Sine transformation*

Description

Inverse Hyperbolic Sine transformation

Usage

```
transform_asinh()

asinh_trans()
```

Examples

```
plot(transform_asinh(), xlim = c(-1e2, 1e2))
```

transform_asn *Arc-sin square root transformation*

Description

This is the variance stabilising transformation for the binomial distribution.

Usage

```
transform_asn()

asn_trans()
```

Examples

```
plot(transform_asn(), xlim = c(0, 1))
```

transform_atanh *Arc-tangent transformation*

Description

Arc-tangent transformation

Usage

```
transform_atanh()

atanh_trans()
```

Examples

```
plot(transform_atanh(), xlim = c(-1, 1))
```

`transform_boxcox` *Box-Cox & modulus transformations*

Description

The Box-Cox transformation is a flexible transformation, often used to transform data towards normality. The modulus transformation generalises Box-Cox to also work with negative values.

Usage

```
transform_boxcox(p, offset = 0)

boxcox_trans(p, offset = 0)

transform_modulus(p, offset = 1)

modulus_trans(p, offset = 1)
```

Arguments

`p` Transformation exponent, λ.

`offset` Constant offset. 0 for Box-Cox type 1, otherwise any non-negative constant (Box-Cox type 2). `transform_modulus()` sets the default to 1.

Details

The Box-Cox power transformation (type 1) requires strictly positive values and takes the following form for `y > 0`:

$$y^{(\lambda)} = \frac{y^\lambda - 1}{\lambda}$$

When `y = 0`, the natural log transform is used.

The modulus transformation implements a generalisation of the Box-Cox transformation that works for data with both positive and negative values. The equation takes the following forms, when `y != 0` :

$$y^{(\lambda)} = sign(y) * \frac{(|y| + 1)^\lambda - 1}{\lambda}$$

and when `y = 0`:

$$y^{(\lambda)} = sign(y) * \ln(|y| + 1)$$

References

Box, G. E., & Cox, D. R. (1964). An analysis of transformations. Journal of the Royal Statistical Society. Series B (Methodological), 211-252. https://www.jstor.org/stable/2984418

John, J. A., & Draper, N. R. (1980). An alternative family of transformations. Applied Statistics, 190-197. https://www.jstor.org/stable/2986305

See Also

transform_yj()

Examples

```
plot(transform_boxcox(-1), xlim = c(0, 10))
plot(transform_boxcox(0), xlim = c(0, 10))
plot(transform_boxcox(1), xlim = c(0, 10))
plot(transform_boxcox(2), xlim = c(0, 10))

plot(transform_modulus(-1), xlim = c(-10, 10))
plot(transform_modulus(0), xlim = c(-10, 10))
plot(transform_modulus(1), xlim = c(-10, 10))
plot(transform_modulus(2), xlim = c(-10, 10))
```

transform_compose *Compose two or more transformations together*

Description

This transformer provides a general mechanism for composing two or more transformers together. The most important use case is to combine reverse with other transformations.

Usage

```
transform_compose(...)

compose_trans(...)
```

Arguments

...	One or more transformers, either specified with string or as individual transformer objects.

Examples

```
demo_continuous(10^c(-2:4), trans = "log10", labels = label_log())
demo_continuous(10^c(-2:4), trans = c("log10", "reverse"), labels = label_log())
```

transform_date *Transformation for dates (class Date)*

Description

Transformation for dates (class Date)

Usage

```
transform_date()

date_trans()
```

Examples

```
years <- seq(as.Date("1910/1/1"), as.Date("1999/1/1"), "years")
t <- transform_date()
t$transform(years)
t$inverse(t$transform(years))
t$format(t$breaks(range(years)))
```

transform_exp *Exponential transformation (inverse of log transformation)*

Description

Exponential transformation (inverse of log transformation)

Usage

```
transform_exp(base = exp(1))

exp_trans(base = exp(1))
```

Arguments

base	Base of logarithm

Examples

```
plot(transform_exp(0.5), xlim = c(-2, 2))
plot(transform_exp(1), xlim = c(-2, 2))
plot(transform_exp(2), xlim = c(-2, 2))
plot(transform_exp(), xlim = c(-2, 2))
```

`transform_identity` *Identity transformation (do nothing)*

Description

Identity transformation (do nothing)

Usage

```
transform_identity()

identity_trans()
```

Examples

```
plot(transform_identity(), xlim = c(-1, 1))
```

`transform_log` *Log transformations*

Description

- `transform_log()`: `log(x)`
- `log1p()`: `log(x + 1)`
- `transform_pseudo_log()`: smoothly transition to linear scale around 0.

Usage

```
transform_log(base = exp(1))

transform_log10()

transform_log2()

transform_log1p()

log_trans(base = exp(1))

log10_trans()

log2_trans()

log1p_trans()

transform_pseudo_log(sigma = 1, base = exp(1))

pseudo_log_trans(sigma = 1, base = exp(1))
```

Arguments

base	base of logarithm
sigma	Scaling factor for the linear part of pseudo-log transformation.

Examples

```
plot(transform_log2(), xlim = c(0, 5))
plot(transform_log(), xlim = c(0, 5))
plot(transform_log10(), xlim = c(0, 5))

plot(transform_log(), xlim = c(0, 2))
plot(transform_log1p(), xlim = c(-1, 1))

# The pseudo-log is defined for all real numbers
plot(transform_pseudo_log(), xlim = c(-5, 5))
lines(transform_log(), xlim = c(0, 5), col = ”red”)

# For large positives numbers it's very close to log
plot(transform_pseudo_log(), xlim = c(1, 20))
lines(transform_log(), xlim = c(1, 20), col = ”red”)
```

transform_probability *Probability transformation*

Description

Probability transformation

Usage

```
transform_probability(distribution, ...)

transform_logit()

transform_probit()

probability_trans(distribution, ...)

logit_trans()

probit_trans()
```

Arguments

distribution	probability distribution. Should be standard R abbreviation so that "p" + distribution is a valid cumulative distribution function, "q" + distribution is a valid quantile function, and "d" + distribution is a valid probability density function.
...	other arguments passed on to distribution and quantile functions

Examples

```
plot(transform_logit(), xlim = c(0, 1))
plot(transform_probit(), xlim = c(0, 1))
```

transform_reciprocal *Reciprocal transformation*

Description

Reciprocal transformation

Usage

```
transform_reciprocal()

reciprocal_trans()
```

Examples

```
plot(transform_reciprocal(), xlim = c(0, 1))
```

transform_reverse *Reverse transformation*

Description

reversing transformation works by multiplying the input with -1. This means that reverse transformation cannot easily be composed with transformations that require positive input unless the reversing is done as a final step.

Usage

```
transform_reverse()

reverse_trans()
```

Examples

```
plot(transform_reverse(), xlim = c(-1, 1))
```

transform_sqrt *Square-root transformation*

Description

This is the variance stabilising transformation for the Poisson distribution.

Usage

```
transform_sqrt()

sqrt_trans()
```

Examples

```
plot(transform_sqrt(), xlim = c(0, 5))
```

transform_time *Transformation for date-times (class POSIXt)*

Description

Transformation for date-times (class POSIXt)

Usage

```
transform_time(tz = NULL)

time_trans(tz = NULL)
```

Arguments

tz	Optionally supply the time zone. If NULL, the default, the time zone will be extracted from first input with a non-null tz.

Examples

```
hours <- seq(ISOdate(2000, 3, 20, tz = ""), by = "hour", length.out = 10)
t <- transform_time()
t$transform(hours)
t$inverse(t$transform(hours))
t$format(t$breaks(range(hours)))
```

transform_timespan *Transformation for times (class hms)*

Description

transform_timespan() provides transformations for data encoding time passed along with breaks and label formatting showing standard unit of time fitting the range of the data. transform_hms() provides the same but using standard hms idioms and formatting.

Usage

```
transform_timespan(unit = c("secs", "mins", "hours", "days", "weeks"))

timespan_trans(unit = c("secs", "mins", "hours", "days", "weeks"))

transform_hms()

hms_trans()
```

Arguments

unit	The unit used to interpret numeric input

Examples

```
# transform_timespan allows you to specify the time unit numeric data is
# interpreted in
trans_min <- transform_timespan("mins")
demo_timespan(seq(0, 100), trans = trans_min)
# Input already in difftime format is interpreted correctly
demo_timespan(as.difftime(seq(0, 100), units = "secs"), trans = trans_min)

if (require("hms")) {
  # transform_hms always assumes seconds
  hms <- round(runif(10) * 86400)
  t <- transform_hms()
  t$transform(hms)
  t$inverse(t$transform(hms))
  t$breaks(hms)
  # The break labels also follow the hms format
  demo_timespan(hms, trans = t)
}
```

transform_yj *Yeo-Johnson transformation*

Description

The Yeo-Johnson transformation is a flexible transformation that is similar to Box-Cox, transform_boxcox(), but does not require input values to be greater than zero.

Usage

```
transform_yj(p)

yj_trans(p)
```

Arguments

p	Transformation exponent, λ.

Details

The transformation takes one of four forms depending on the values of y and λ.

- $y \geq 0$ and $\lambda \neq 0 : y^{(\lambda)} = \frac{(y+1)^\lambda - 1}{\lambda}$
- $y \geq 0$ and $\lambda = 0$: $y^{(\lambda)} = \ln(y+1)$
- $y < 0$ and $\lambda \neq 2$: $y^{(\lambda)} = -\frac{(-y+1)^{(2-\lambda)} - 1}{2-\lambda}$
- $y < 0$ and $\lambda = 2$: $y^{(\lambda)} = -\ln(-y+1)$

References

Yeo, I., & Johnson, R. (2000). A New Family of Power Transformations to Improve Normality or Symmetry. Biometrika, 87(4), 954-959. https://www.jstor.org/stable/2673623

Examples

```
plot(transform_yj(-1), xlim = c(-10, 10))
plot(transform_yj(0), xlim = c(-10, 10))
plot(transform_yj(1), xlim = c(-10, 10))
plot(transform_yj(2), xlim = c(-10, 10))
```

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

Description

The machine epsilon is the difference between 1.0 and the next number that can be represented by the machine. By default, this function uses epsilon * 1000 as the tolerance. First it scales the values so that they have a mean of 1, and then it checks if the difference between them is larger than the tolerance.

Usage

```
zero_range(x, tol = 1000 * .Machine$double.eps)
```

Arguments

`x`	numeric range: vector of length 2
`tol`	A value specifying the tolerance.

Value

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

Examples

```
eps <- .Machine$double.eps
zero_range(c(1, 1 + eps))
zero_range(c(1, 1 + 99 * eps))
zero_range(c(1, 1 + 1001 * eps))
zero_range(c(1, 1 + 2 * eps), tol = eps)

# Scaling up or down all the values has no effect since the values
# are rescaled to 1 before checking against tol
zero_range(100000 * c(1, 1 + eps))
zero_range(100000 * c(1, 1 + 1001 * eps))
zero_range(.00001 * c(1, 1 + eps))
zero_range(.00001 * c(1, 1 + 1001 * eps))

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

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

Index