Using the scales package

01 - Tweaking axis breaks and labels

AUTHOR
Qiushi Yan

AFFILIATION
Communication University of China

PUBLISHED
Nov. 26, 2019

CITATION
Yan, 2019

TABLE OF CONTENTS

Basics
Axis breaks
- `breaks_width()`: equally spaced breaks
- `breaks_pretty()`: pretty breaks
- `breaks_extended()`: Wilkinson's extended breaks algorithm for numerical axes
- `breaks_log()`: breaks for log axes

Axis labels
- label numbers
  - decimal format
  - scientific format
  - ordinal numbers (1st, 2nd, 3rd, etc.)
  - SI unit prefix
  - percent format
  - label currencies
  - mathematical annotations
  - label p-values
  - label bytes
- label date / times
- label strings

One of the most slighted parts of making a **ggplot2** [1] visualization is scaling, and its inverse, guiding. This is the case partly because in ggplot2 scales and guides are automatically generated, and generated pretty well. Perhaps frequentyly we work with `scale_color_` and `scale_fill_` to change palettes used, yet aside from that, we have few experience tweaking scales, adjusting breaks and labels, modifying axes and legends or so. The **scales** [2] provides a internal scaling infrastructure used by ggplot2, and a set of consistent tools to override the default breaks, labels, transformations and palettes.

The **scales** package can be installed from cran via:

```
install.packages("scales")
```

or from GitHub if you want the development version:

```
devtools::install_github("r-lib/scales")
```

```
library(scales)
library(ggplot2)
```

If you are just tweaking a few plots, running `library(scales)`, is not recommended because when you type (e.g.) `scales::label_` autocomplete will provide you with a list of labelling functions to job your memory.

Note: This sereis of blogs are based on **scales** 1.1.0.9000.

Basics

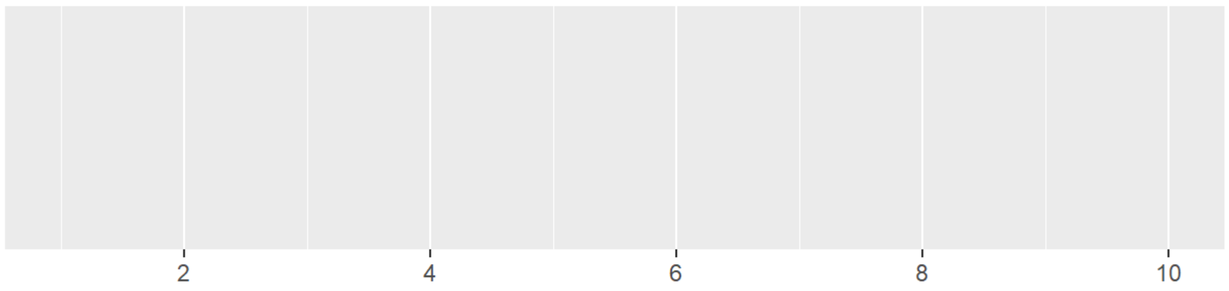
There are 4 helper functions in **scales** used to demonstrate **ggplot2** style scales for specific types of data:

- `demo_continuous()` and `demo_log10()` for numerical axes
- `demo_discrete()` for discrete axes
- `demo_datetime` for data / time axes

These functions share common API deisgn, with the first argument specifying the limits of the scale, and `breaks`, `labels` arguments overriding its default apperance.

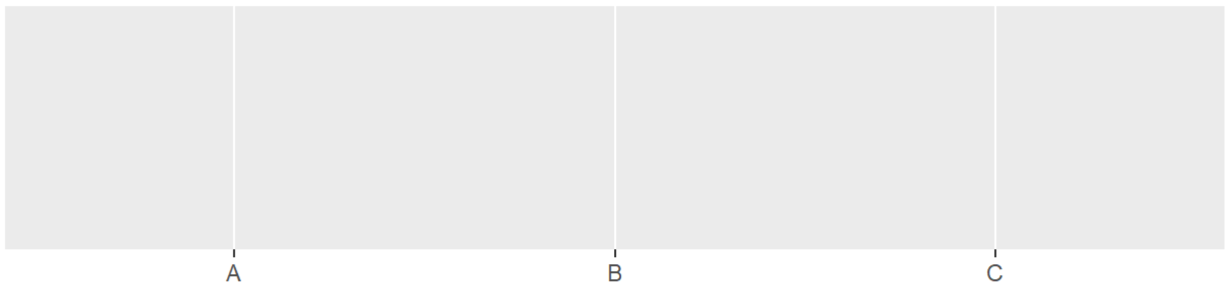
```
demo_continuous(c(1, 10), breaks = breaks_width(2))
```

```
#> scale_x_continuous(breaks = breaks_width(2))
```



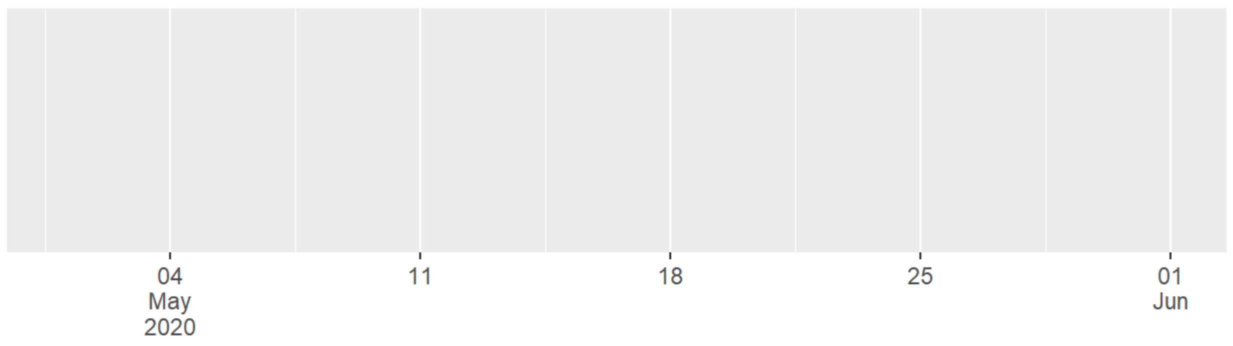
```
demo_discrete(c("A", "B", "C"))
```

```
#> scale_x_discrete()
```



```
one_month <- as.POSIXct(c("2020-05-01", "2020-06-01"))
demo_datetime(one_month, labels = label_date_short())
```

```
#> scale_x_datetime(labels = label_date_short())
```



Axis breaks

`breaks_width()`: equally spaced breaks

`breaks_width()` is commoly supplied to the `breaks` arguent in scale function for equally spaced breaks, useful for numeric, date, and date-time scales.

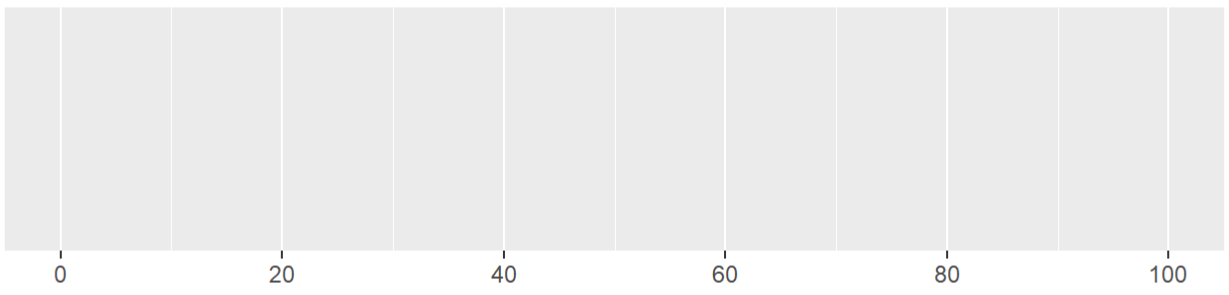
```
breaks_width(width, offset = 0)
```

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

An simple example :

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

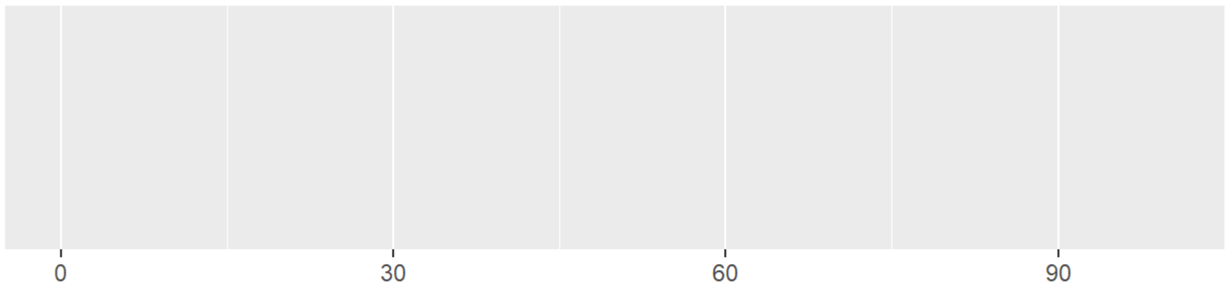
```
#> scale_x_continuous(breaks = breaks_width(20))
```



The break width doesn't have to be a divisor of the scale span, in those cases limits of the scale will be automatically extented or cut:

```
demo_continuous(c(0, 100), breaks = breaks_width(30))
```

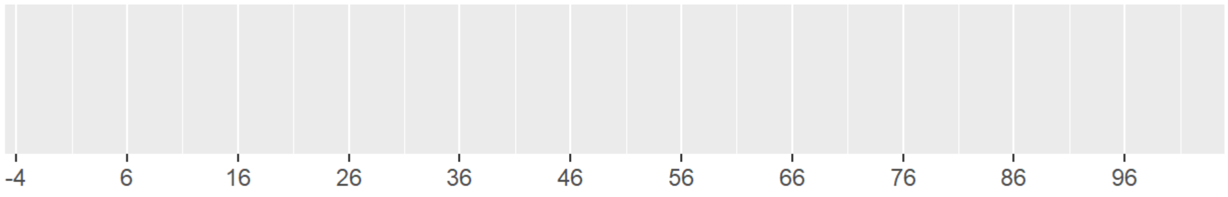
```
#> scale_x_continuous(breaks = breaks_width(30))
```



The `offset` argument specifies an new starting point with an "offset" away from the original one:

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

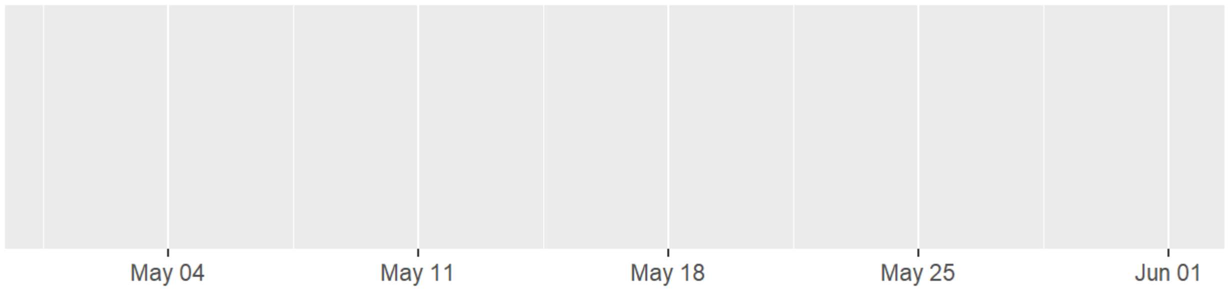
```
#> scale_x_continuous(breaks = breaks_width(10, -4))
```



`breaks_width()` also works on dates and time, now `width` could be a single string of the form “n unit”, e.g. “1 month”, “5 days”, or one of “sec”, “min”, “hour”, “day”, “week”, “month”, “year”.

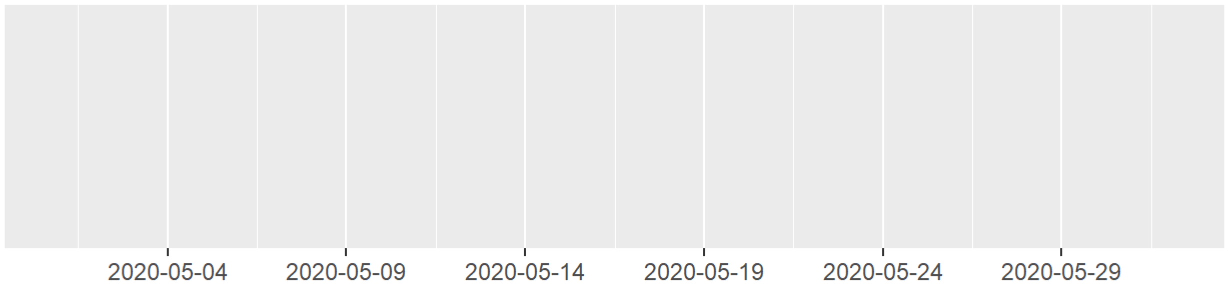
```
one_month <- as.POSIXct(c("2020-05-01", "2020-06-01"))
demo_datetime(one_month)
```

```
#> scale_x_datetime()
```



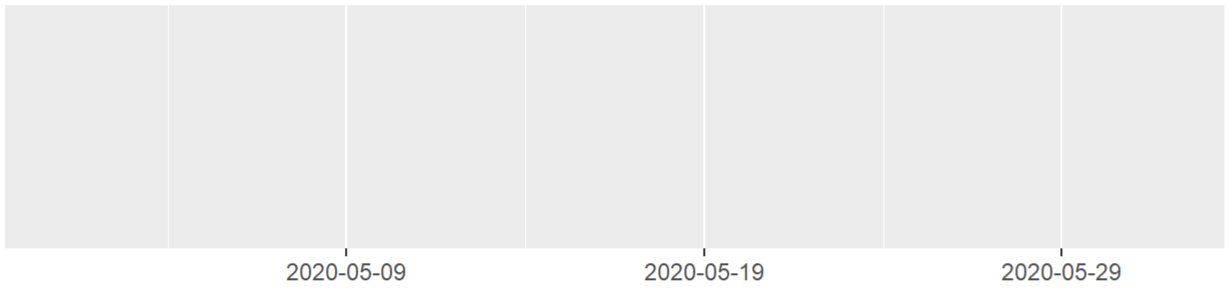
```
# better specifying labels as well
demo_datetime(one_month, breaks = breaks_width("5 days"))
```

```
#> scale_x_datetime(breaks = breaks_width("5 days"))
```



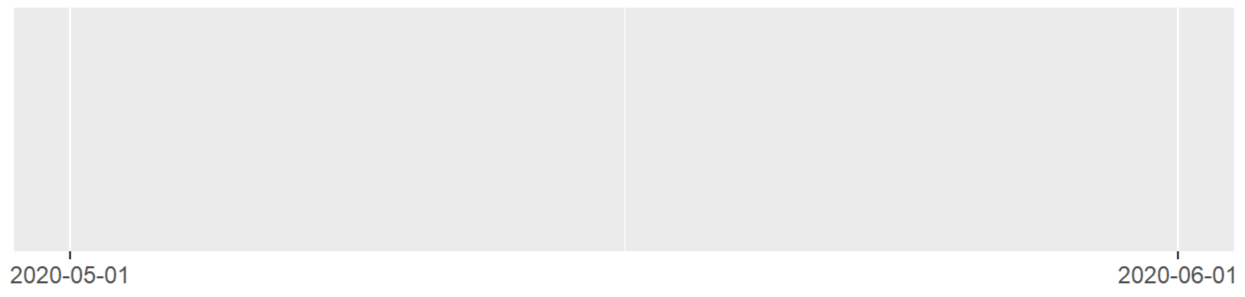
```r
demo_datetime(one_month, breaks = breaks_width("10 days"))
```

```
#> scale_x_datetime(breaks = breaks_width("10 days"))
```



```r
demo_datetime(one_month, breaks = breaks_width("month"))
```

```
#> scale_x_datetime(breaks = breaks_width("month"))
```



`breaks_pretty()`: pretty breaks

In base R, `pretty()` compute breaks based on a specific sequence, i.e:

```
# automatically choosing # of breaks
pretty(1:30)
```

```
#> [1]  0  5 10 15 20 25 30
```

```
# n giving the desired number of intervals,  result may be more or fewer
pretty(1:30, n = 3)
```

```
#> [1]  0 10 20 30
```

`pretty()` could also be used to compute breakpoints for date / time object, since they can be coerced to numeric data:

```
pretty(one_month, n = 6)
```

```
#> [1] "2020-04-27 CST" "2020-05-04 CST" "2020-05-11 CST"
#> [4] "2020-05-18 CST" "2020-05-25 CST" "2020-06-01 CST"
```

```
as.numeric(one_month)
```

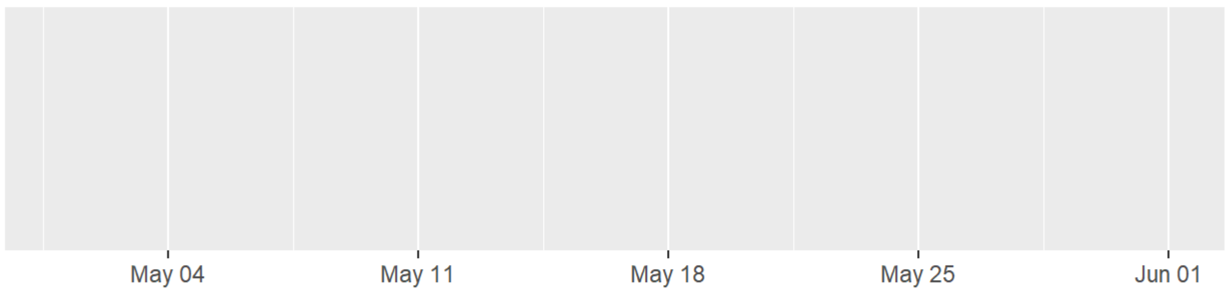
```
#> [1] 1588262400 1590940800
```

Other breakpoints algorithm can be found in the **labeling** package [3] .

`breaks_pretty()` uses default R break algorithm as implemented in `pretty()`, this is primarily used for datetime axes in ggplot2 ecosystem, and `breaks_extended` should do a slightly better job for numerical scales:

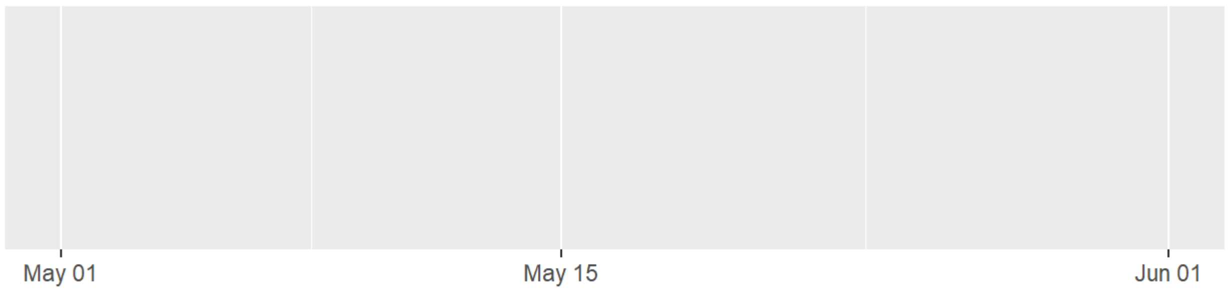
```
demo_datetime(one_month)
```

```
#> scale_x_datetime()
```



```
demo_datetime(one_month, breaks = breaks_pretty(n = 4))
```

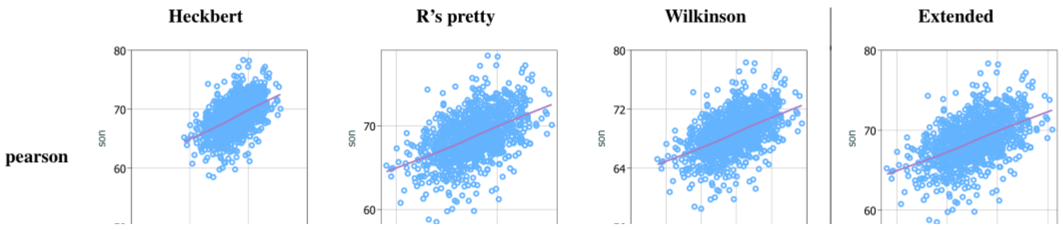
```
#> scale_x_datetime(breaks = breaks_pretty(n = 4))
```



breaks_extended(): Wilkinson’s extended breaks algorithm for numerical axes

`breaks_extended()` uses Wilkinson’s extended breaks algorithm as implemented in the **labeling** package. `extended()`, its corresponding function in base R, is an enhanced version of Wilkinson’s optimization-based axis labeling approach `wilkinson()`. It performs better than a variety of labeling algorithm on random labeling and breaking tasks, including `pretty()`.

For more details, please see [4] .



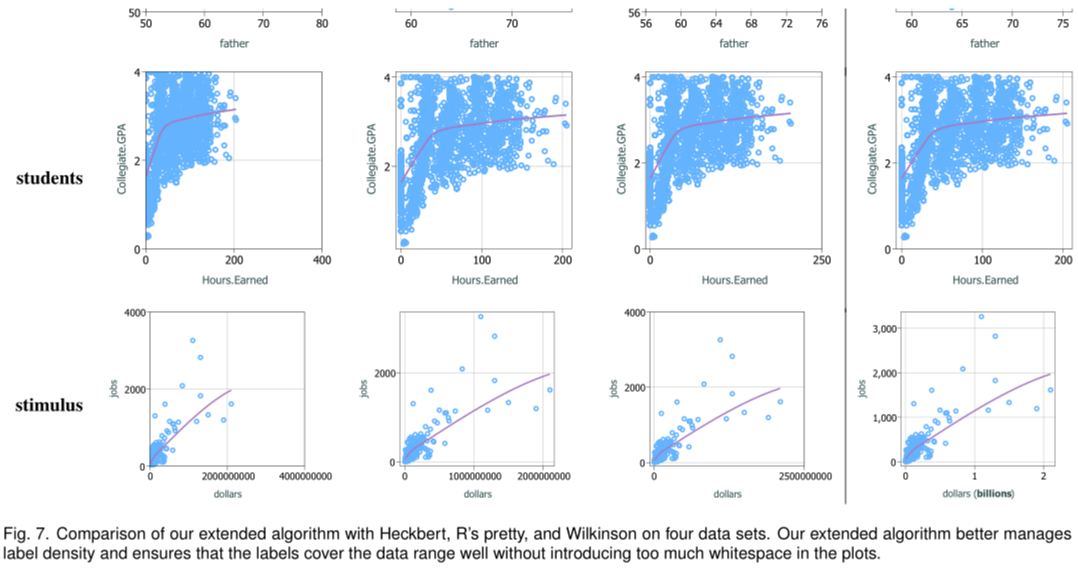


Figure 1: A algorithm comparison plot presented in the paper mentioned above

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

n

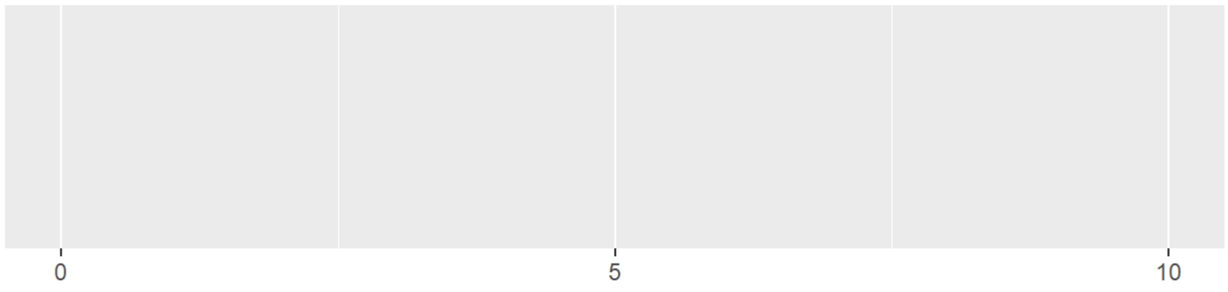
Desired number of breaks. You may get slightly more or fewer breaks that requested.

...

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

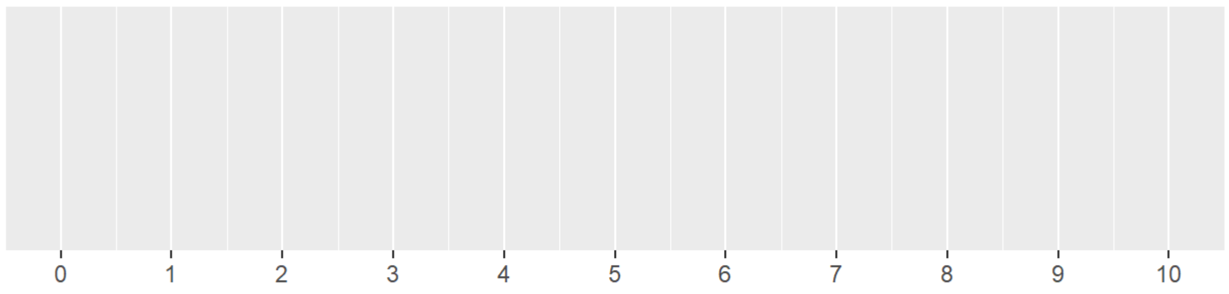
```
demo_continuous(c(0, 10), breaks = breaks_extended(3))
```

```
#> scale_x_continuous(breaks = breaks_extended(3))
```



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

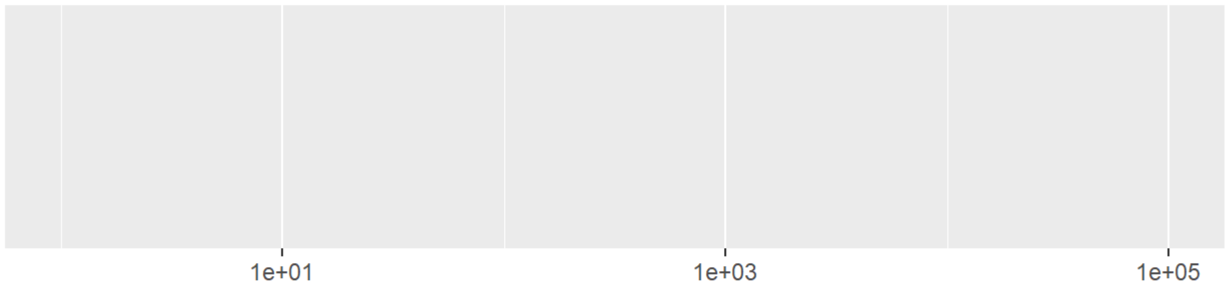
```
#> scale_x_continuous(breaks = breaks_extended(10))
```



`breaks_log()`: breaks for log axes

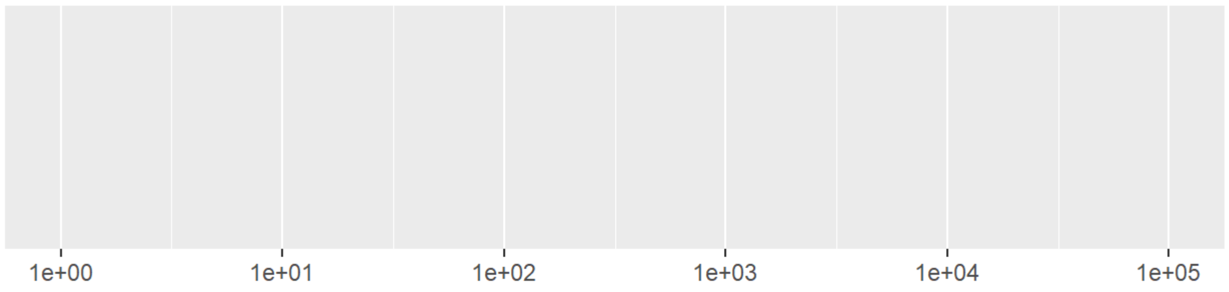
```
demo_log10(c(1, 1e5))
```

```
#> scale_x_log10()
```



```
# Request more breaks by setting n
demo_log10(c(1, 1e5), breaks = breaks_log(n = 6))
```

```
#> scale_x_log10(breaks = breaks_log(n = 6))
```



Axis labels

label numbers

decimal format

Use `label_number()` and its variants to force decimal display of numbers, that is, the antithesis of using scientific notation(e.g., 2×10^6 in decimal format would be $2,000,000$). `label_comma()` is a special case that inserts a comma every three digits.

```
label_number(accuracy = NULL, scale = 1,
             prefix = "", suffix = "",
             big.mark = " ", decimal.mark = ".")

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

comma(x, accuracy = NULL, scale = 1,
      prefix = "", suffix = "",
      big.mark = ",", decimal.mark = ".")
```

`comma()` should be replaced with label_comma()

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.

scale

A scaling factor: x will be multiplied by scale before formating. 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.

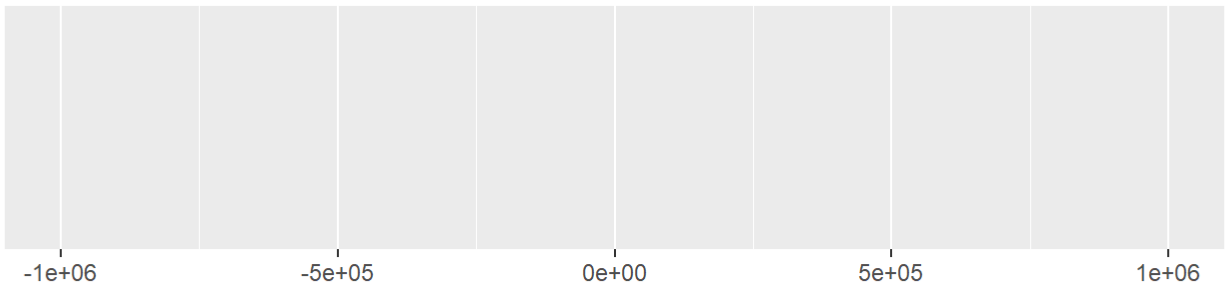
decimal.mark
The character to be used to indicate the numeric decimal point.

`label_numebr` is maily used for large number and `label_comma()` for smaller one, but they are exchangeable.

some examples:

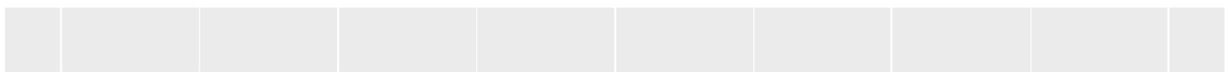
```
demo_continuous(c(-1e6, 1e6))
```

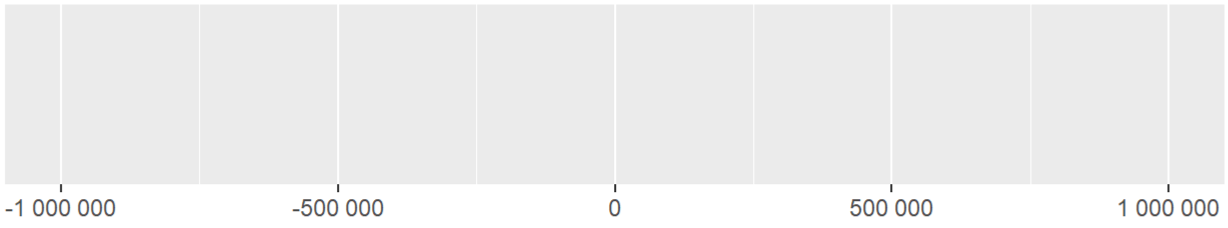
```
#> scale_x_continuous()
```



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

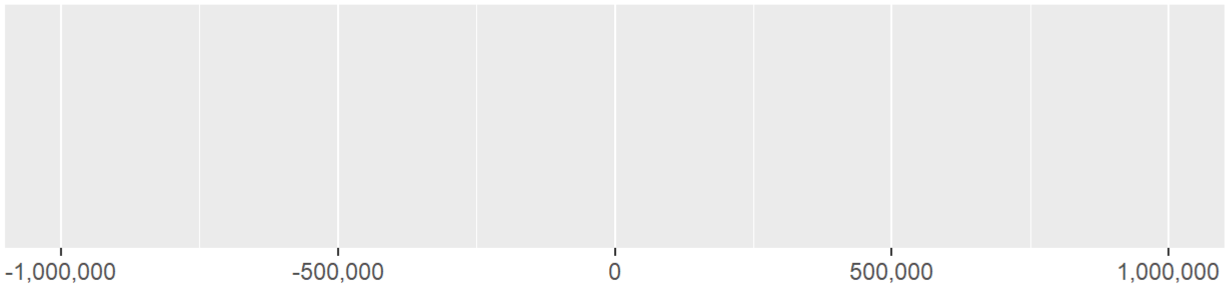
```
#> scale_x_continuous(labels = label_number())
```





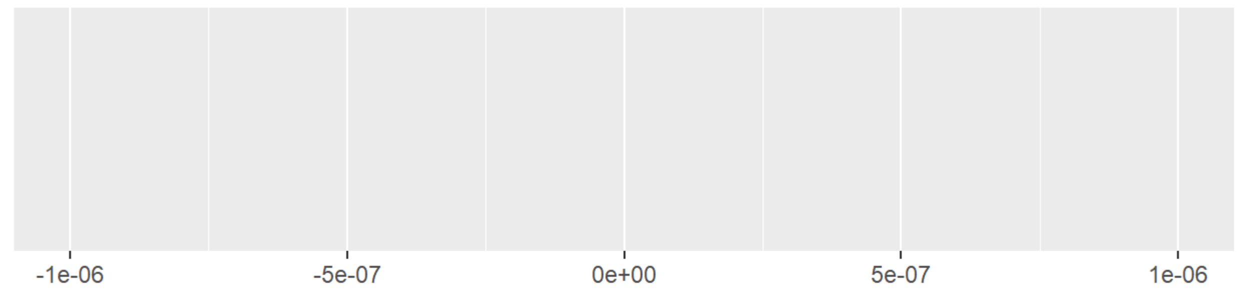
```
demo_continuous(c(-1e6, 1e6), labels = label_comma())
```

```
#> scale_x_continuous(labels = label_comma())
```



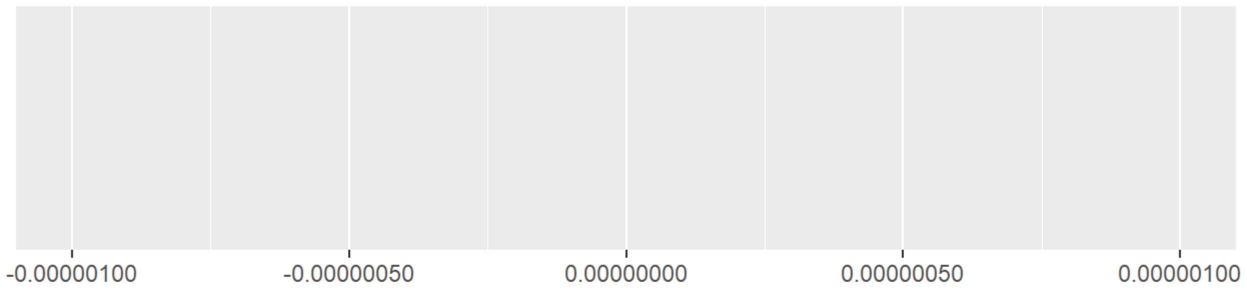
```
# smaller data
demo_continuous(c(-1e-6, 1e-6))
```

```
#> scale_x_continuous()
```



```
demo_continuous(c(-1e-6, 1e-6), labels = label_number())
```

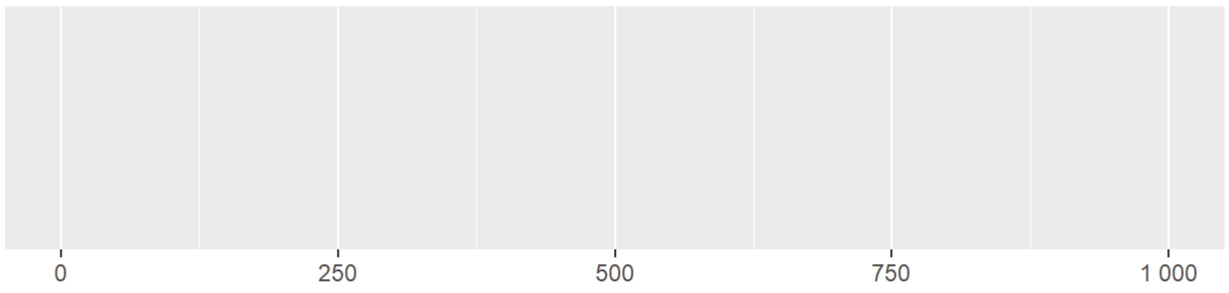
```
#> scale_x_continuous(labels = label_number())
```



Use **scale** to rescale very small or large numbers to generate more readable labels:

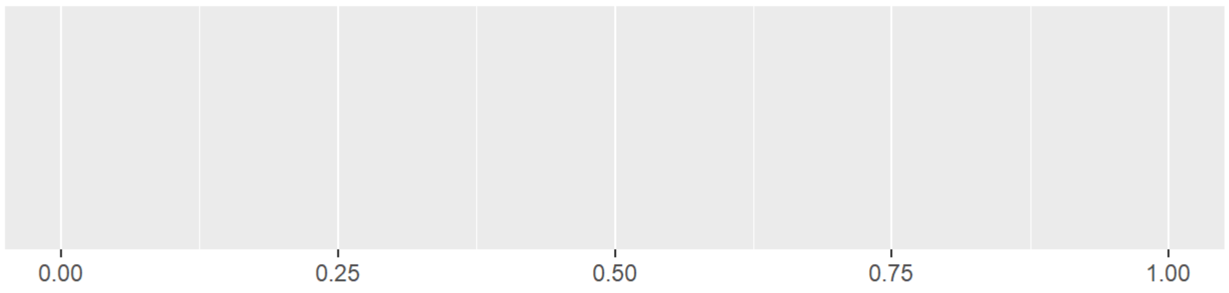
```
demo_continuous(c(0, 1e6), labels = label_number(scale = 1 / 1e3))
```

```
#> scale_x_continuous(labels = label_number(scale = 1/1000))
```



```
demo_continuous(c(0, 1e-6), labels = label_number(scale = 1e6))
```

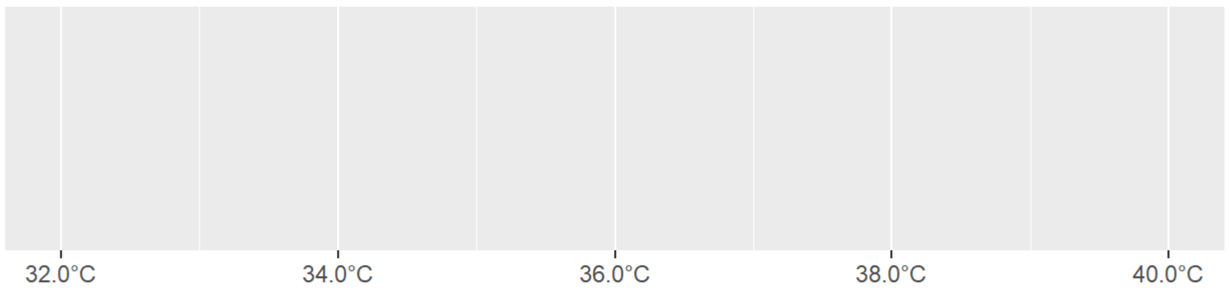
```
#> scale_x_continuous(labels = label_number(scale = 1e+06))
```



Use `prefix` and `suffix` for other types of display:

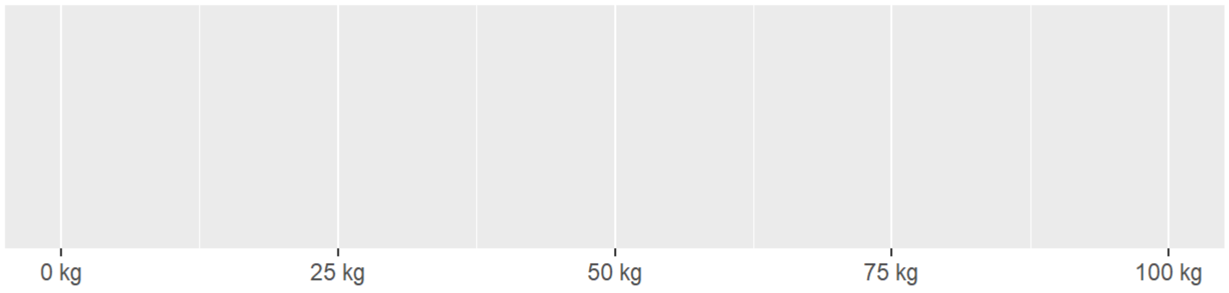
```
demo_continuous(c(32, 40), label = label_number(suffix = "\u00b0C"))
```

```
#> scale_x_continuous(label = label_number(suffix = "°C"))
```



```
demo_continuous(c(0, 100), label = label_number(suffix = " kg"))
```

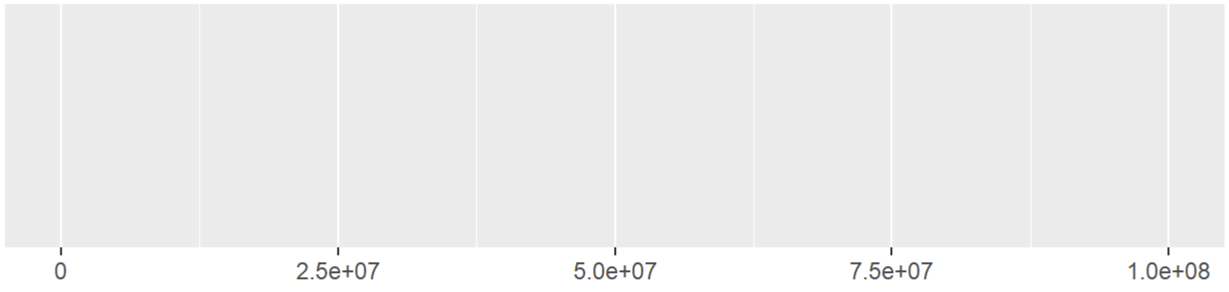
```
#> scale_x_continuous(label = label_number(suffix = " kg"))
```



There is a `label_number_auto()` function that are designed to automatically generated scientific or decimal format labels:

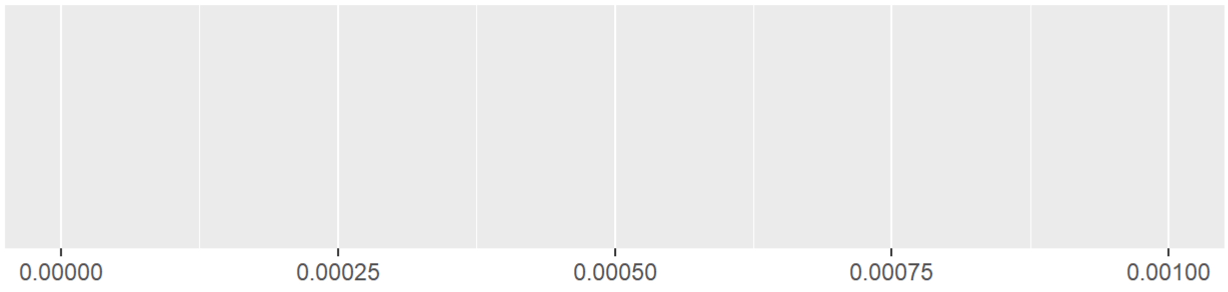
```
# scientific notation
demo_continuous(c(0, 1e8), labels = label_number_auto())
```

```
#> scale_x_continuous(labels = label_number_auto())
```



```
# decimal foramt
demo continuous(c(0. 1e-3). labels = label number auto())
```

```
#> scale_x_continuous(labels = label_number_auto())
```



scientific format

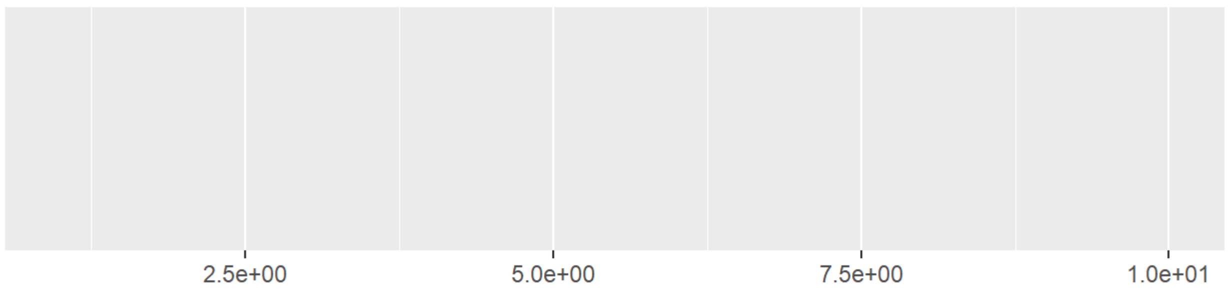
`label_scientific()` forces numbers to be labelled with scientific notation;

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

digits
Number of digits to show before exponent.

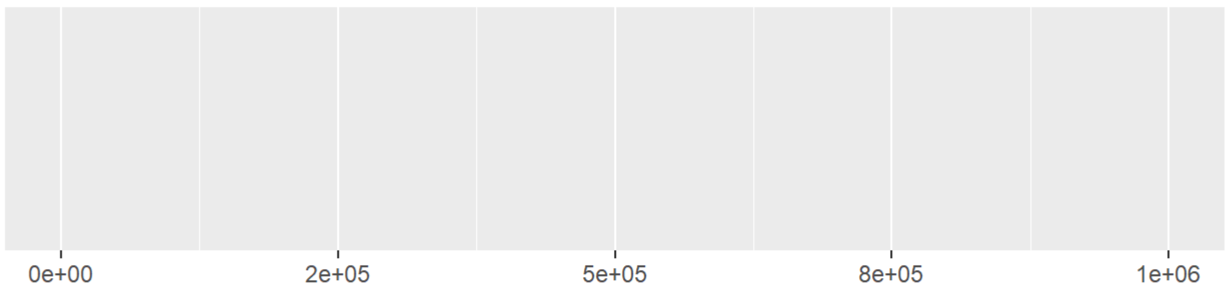
```
demo_continuous(c(1, 10), labels = label_scientific())
```

```
#> scale_x_continuous(labels = label_scientific())
```



```
demo_continuous(c(0, 1e6), labels = label_scientific(digits = 1))
```

```
#> scale_x_continuous(labels = label_scientific(digits = 1))
```



ordinal numbers (1st, 2nd, 3rd, etc.)

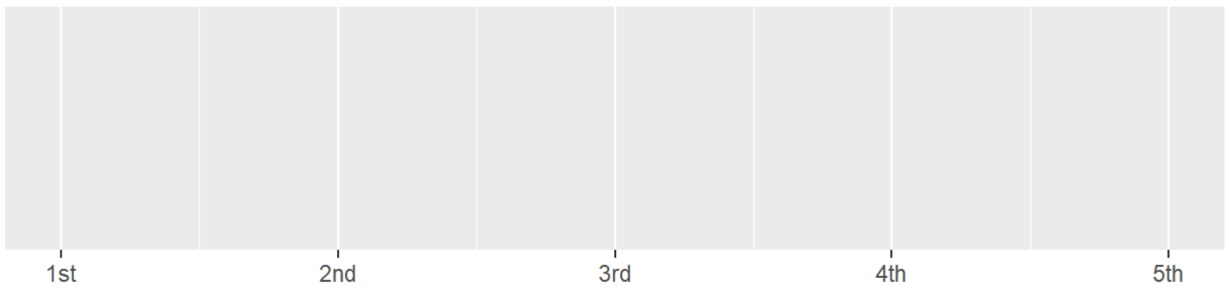
Round values to integers and then display as ordinal values (e.g. 1st, 2nd, 3rd). Built-in rules are provided for English, French, and Spanish.

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

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

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

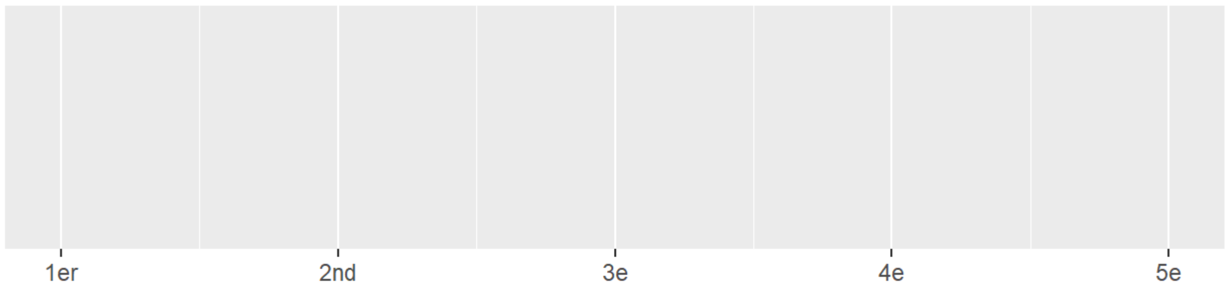
```
#> scale_x_continuous(labels = label_ordinal())
```



Other languages:

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

```
#> scale_x_continuous(labels = label_ordinal(rules = ordinal_french()))
```



SI unit prefix

SI units are any of the units adopted for international use under the Système International d'Unités, now employed for all scientific and most technical purposes. There are seven fundamental units: the metre, kilogram, second, ampere, kelvin, candela, and mole; and two supplementary units: the radian and the steradian.

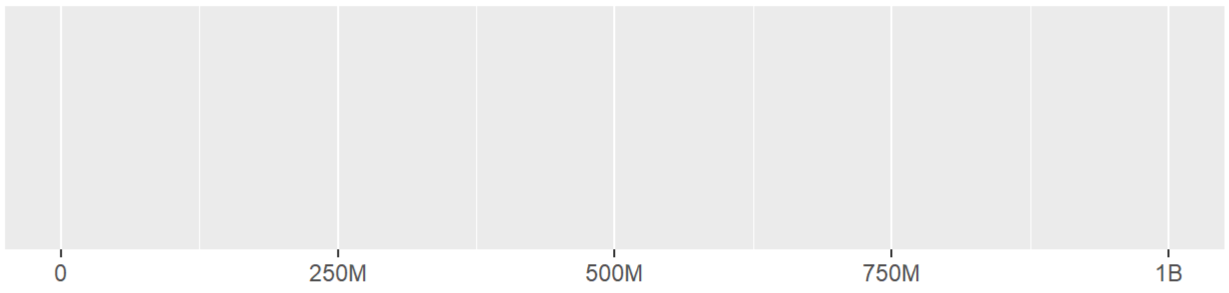
`label_number_si()` automatically scales and labels with the best SI prefix, "K" for values ≥ 10e3, "M" for ≥ 10e6, "B" for ≥ 10e9, and "T" for ≥ 10e12.

```
label_number_si(accuracy = 1, unit = NULL)
```

unit: unit used in the original data, optional

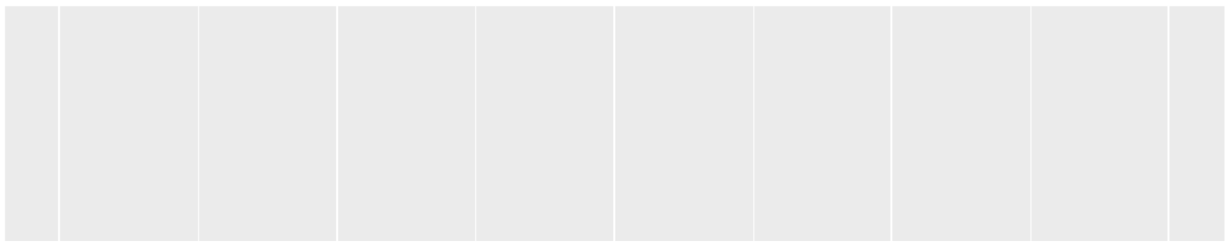
```
# default si units
demo_continuous(c(1, 1e9), label = label_number_si())
```

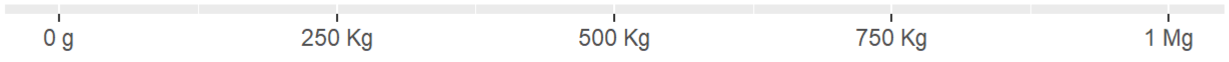
```
#> scale_x_continuous(label = label_number_si())
```



```
# the original data are measuring weight, in g
demo_continuous(c(1e3, 1e6), label = label_number_si(unit = "g"))
```

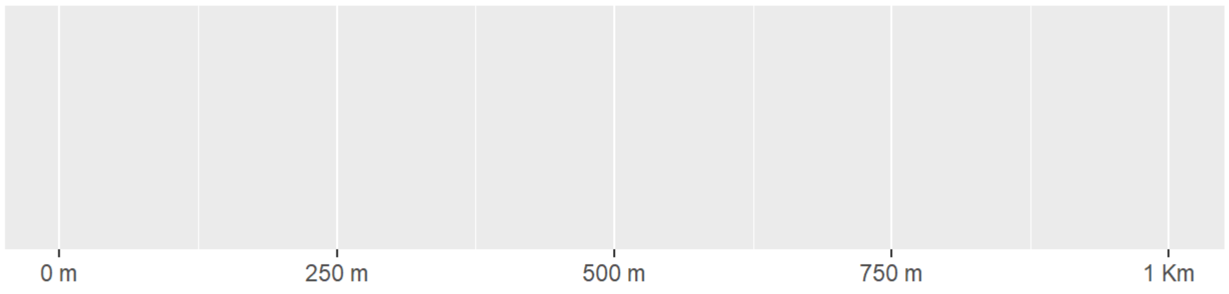
```
#> scale_x_continuous(label = label_number_si(unit = "g"))
```





```
# the original data are measuring length, in m
demo_continuous(c(1, 1000), label = label_number_si(unit = "m"))
```

```
#> scale_x_continuous(label = label_number_si(unit = "m"))
```



percent format

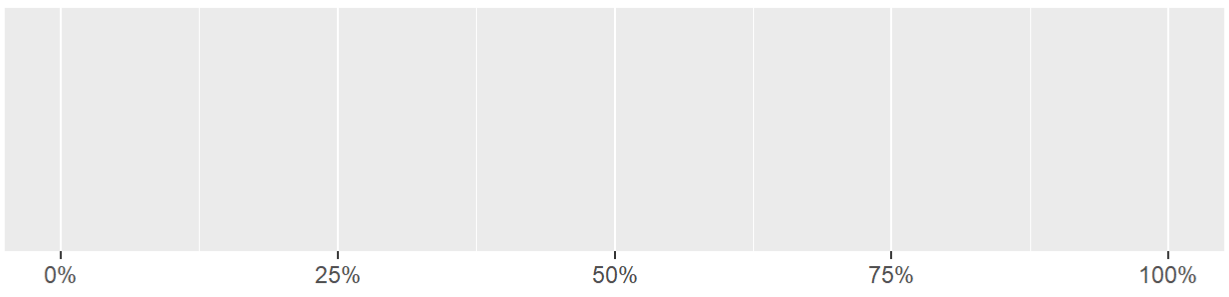
`label_percent()` is used to generate percentage-format labels(e.g., 2.5%, 50%, etc.)

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

`percent()` and `percent_format()` are retired; please use label_percent() instead.

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

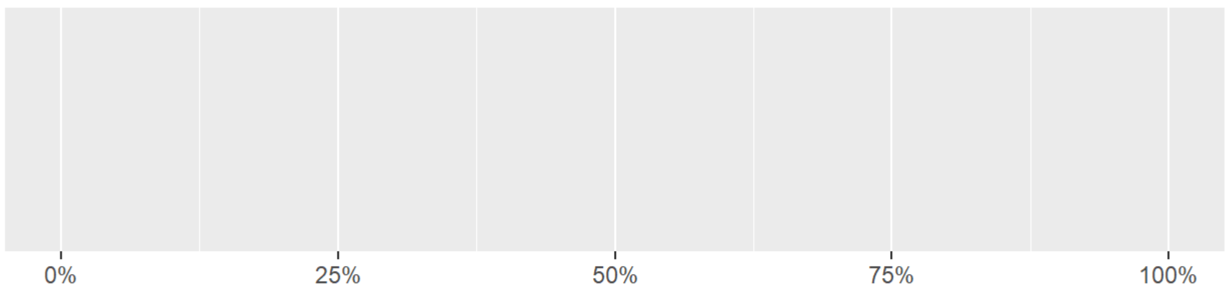
```
#> scale_x_continuous(labels = label_percent())
```



When applying `label_percent()`, every numebr are first multiplied by 100 and then assigned a “%” suffix, it’s sometimes useful to adjust `scale` to change this behaviour:

```
demo_continuous(c(0, 100), labels = label_percent(scale = 1))
```

```
#> scale_x_continuous(labels = label_percent(scale = 1))
```



label currencies

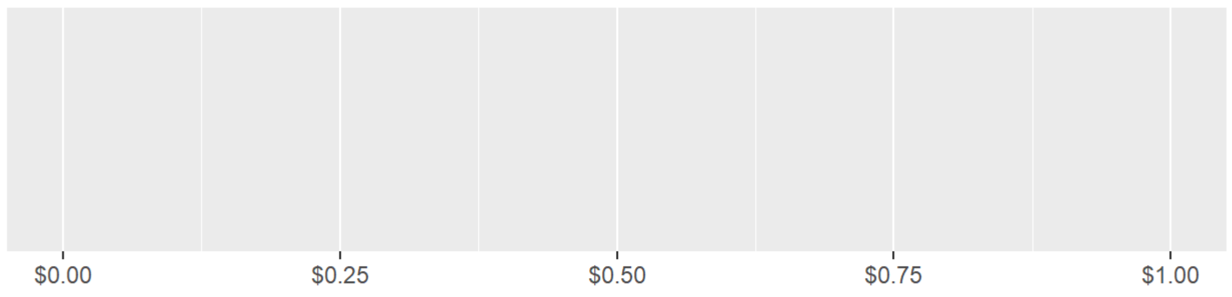
`label_dollar()` format numbers as currency, rounding values to dollars or cents using a convenient heuristic.

```
label_dollar(accuracy = NULL, scale = 1, prefix = "$", suffix = "",
  big.mark = ",", decimal.mark = ".", trim = TRUE,
  largest_with_cents = 1e+05, negative_parens = FALSE, ...)
```

largest_with_cents
values has non-zero fractional component (e.g. cents) and the largest value is less than largest_with_cents which by default is 100,000.

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

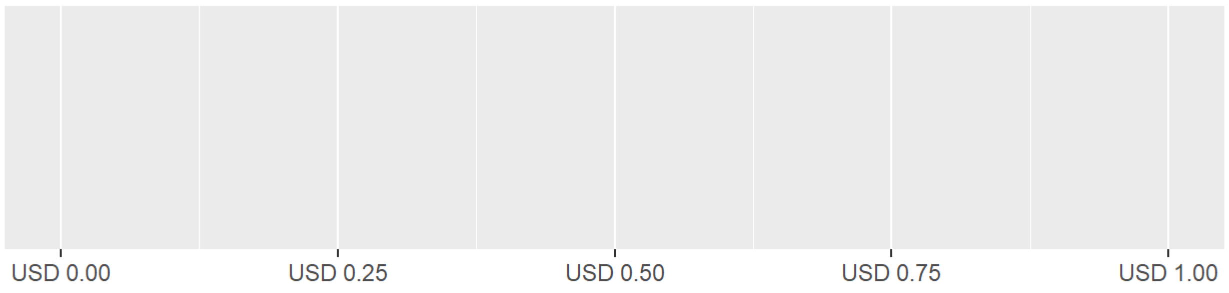
```
#> scale_x_continuous(labels = label_dollar())
```



Change `prefix`:

```
demo_continuous(c(0, 1), labels = label_dollar(prefix = "USD "))
```

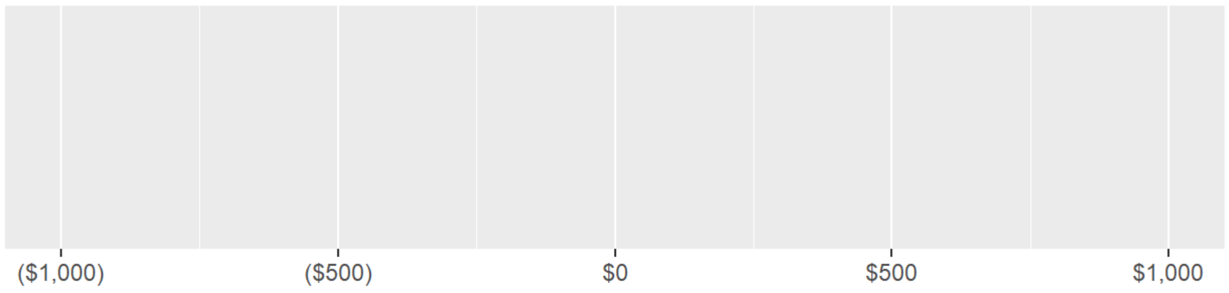
```
#> scale_x_continuous(labels = label_dollar(prefix = "USD "))
```



Use `negative_parens = TRUE` for finance style display:

```r
demo_continuous(c(-1000, 1000), labels = label_dollar(negative_parens = T))
```

```
#> scale_x_continuous(labels = label_dollar(negative_parens = T))
```



mathematical annotations

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

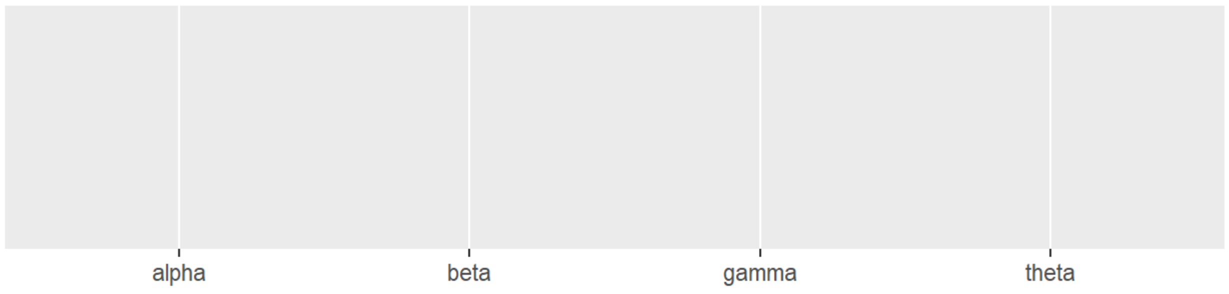
```
label_parse()

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

Use `label_parse()` with discrete scales:

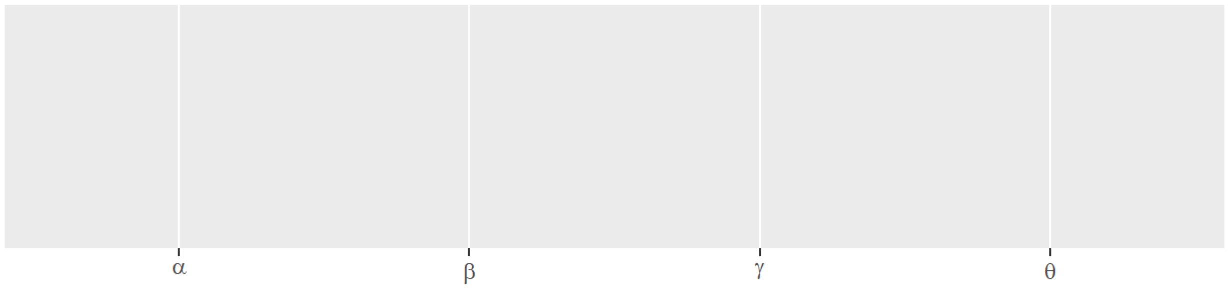
```
demo_discrete(c("alpha", "beta", "gamma", "theta"))
```

```
#> scale_x_discrete()
```



```
demo_discrete(c("alpha", "beta", "gamma", "theta"), labels = label_parse())
```

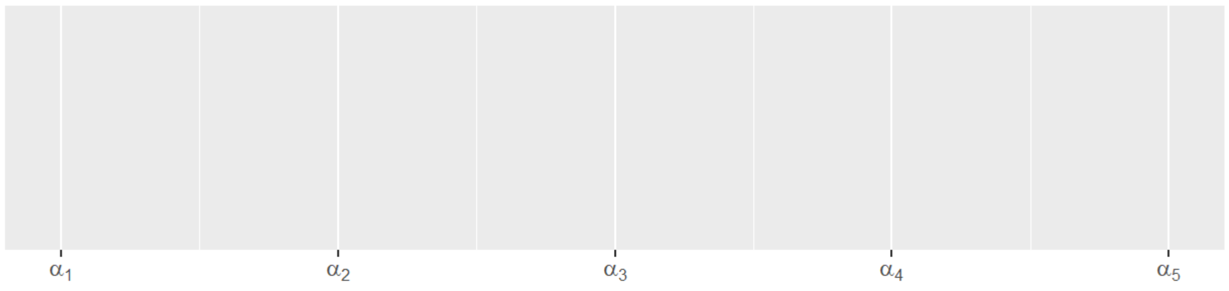
```
#> scale_x_discrete(labels = label_parse())
```



Use `label_math()` with continuous scales:

```
demo_continuous(c(1, 5), labels = label_math(alpha[.x]))
```

```
#> scale_x_continuous(labels = label_math(alpha[.x]))
```



label p-values

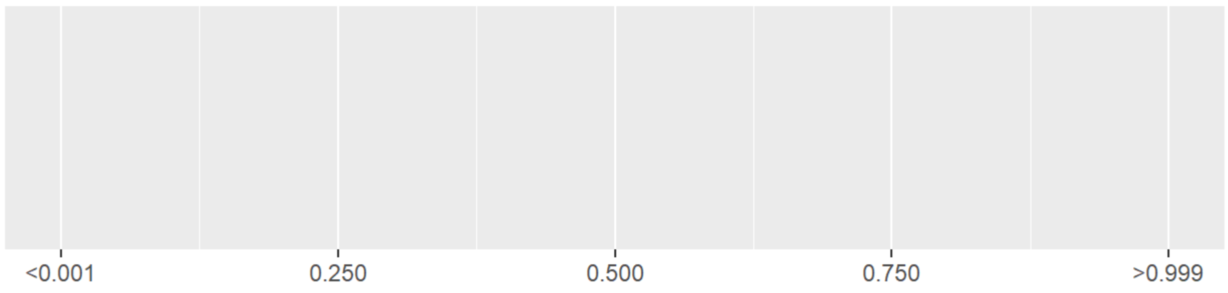
`label_pvalue()` is a convenient formmater for p-values, using “**<**” and “**>**” for p-values close to 0 and 1.

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

add_p
Add “p=” before the value?

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

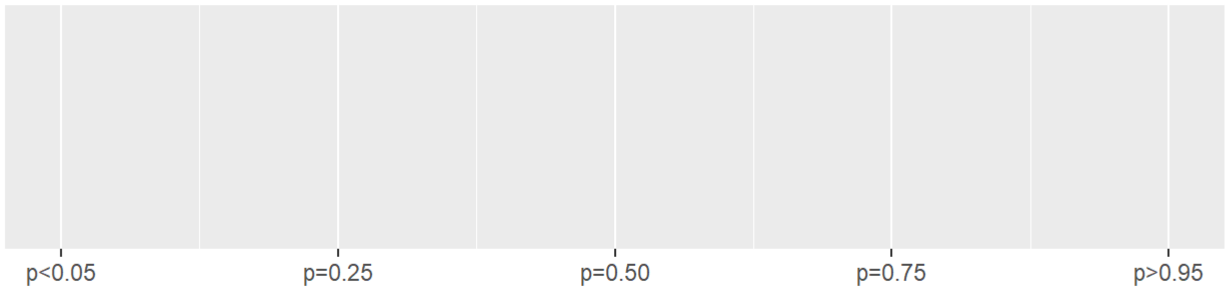
```
#> scale_x_continuous(labels = label_pvalue())
```



`accuracy` can be used as significant level:

```
demo_continuous(c(0, 1), labels = label_pvalue(accuracy = 0.05, add_p = TRUE))
```

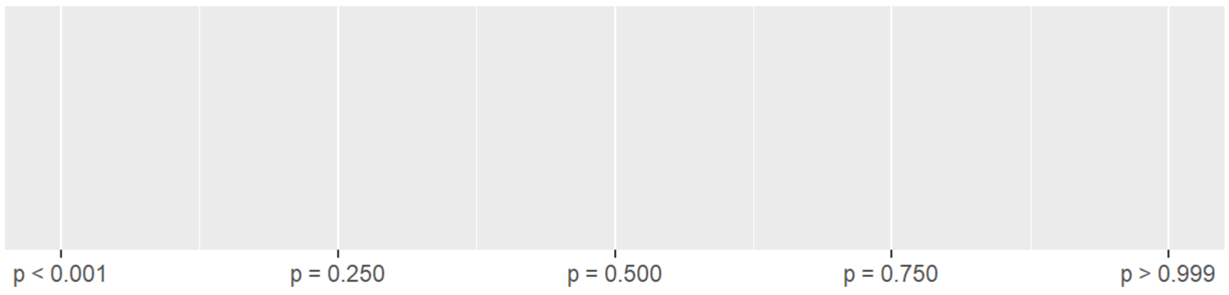
```
#> scale_x_continuous(labels = label_pvalue(accuracy = 0.05, add_p = TRUE))
```



Or provide your own prefixes:

```
prefix <- c("p < ", "p = ", "p > ")
demo_continuous(c(0, 1), labels = label_pvalue(prefix = prefix))
```

```
#> scale_x_continuous(labels = label_pvalue(prefix = prefix))
```



label bytes

`label_bytes` scale bytes into human friendly units. Can use either SI units (e.g. kB = 1000 bytes) or binary units (e.g. kiB = 1024 bytes).

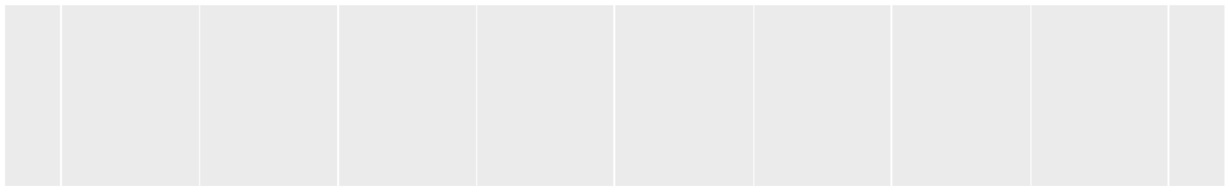
```
label_bytes(units = "auto_si", accuracy = 1)
```

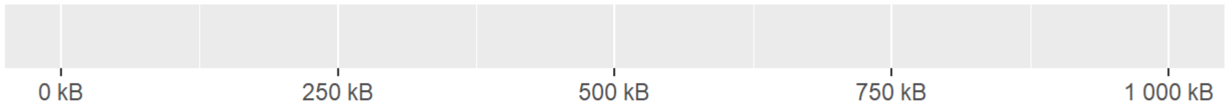
units
Unit to use. Should either one of:
- "kB", "MB", "GB", "TB", "PB", "EB", "ZB", and "YB" for SI units (base 1000). - "kiB", "MiB", "GiB", "TiB", "PiB", "EiB", "ZiB", and "YiB" for binary units (base 1024).

Note: here the `units` argument are **unit to use**, not **the original unit** (in the case of `label_number_si()`), the original unit here are always bytes.
`auto_si` or `auto_binary` to automatically pick the most approrpiate unit for each value.

```
demo_continuous(c(1, 1e6), label = label_bytes("kB"))
```

```
#> scale_x_continuous(label = label_bytes("kB"))
```





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.

label date / times

`label_date()` and `label_time()` label date/times using date/time format strings. `label_date_short()` automatically constructs a short format string suffiicient to uniquely identify labels.

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

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

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

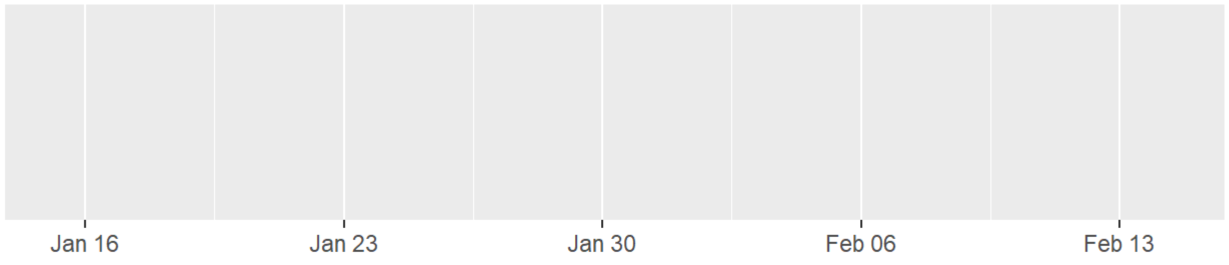
format a date/time format string using standard POSIX specification. See `strptime()` for details.

```
date_range <- function(start, days) {

  library(lubridate)
  start <- ymd(start)
  c(as.POSIXct(start), as.POSIXct(start + days(days)))
}

library(scales)
demo_datetime(date_range("20170115", 30))
```

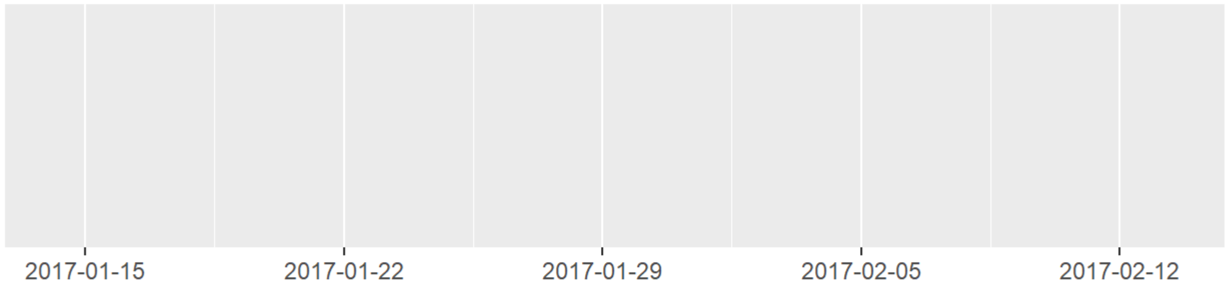
```
#> scale_x_datetime()
```



`demo_datetime()` works with objects of class `POSIXct` only

```
demo_datetime(date_range("20170115", 30), labels = label_date())
```

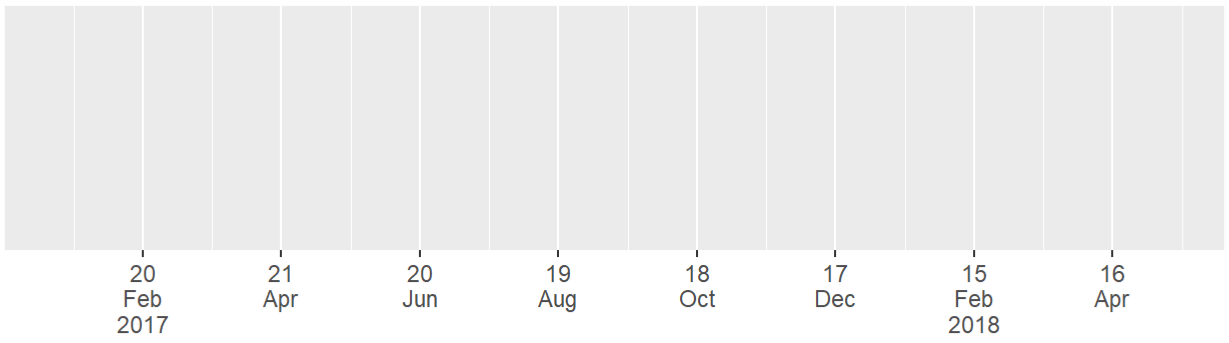
```
#> scale_x_datetime(labels = label_date())
```



Use `label_date_short()`, not here we combine what we have learned in `breaks_width()`

```
demo_datetime(date_range("20170115", 480), labels = label_date_short(),
              breaks = breaks_width("60 days"))
```

```
#> scale_x_datetime(labels = label_date_short(), breaks = breaks_width("60 days"))
```



When scaling dates and times, more often than not we have to specify `labels` and `breaks`, so **ggplot2** provides 2 short-hand arguments `date_breaks()` and `date_labels()`

i.e.

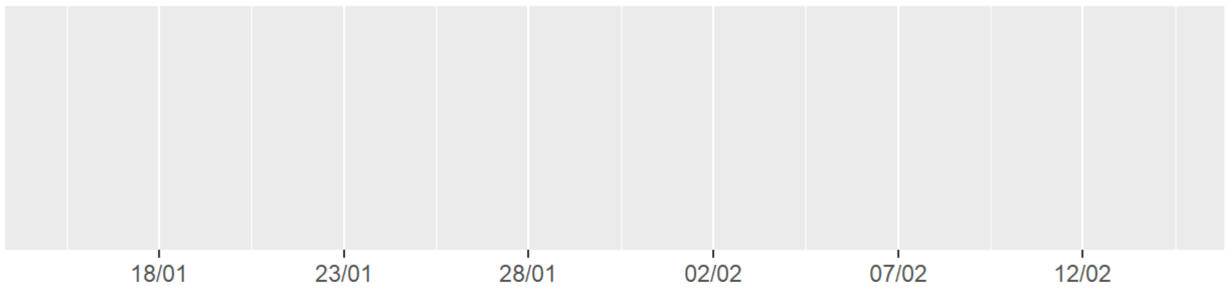
`date_breaks = "2 weeks"` **equivalent to** `breaks = breaks_width("2 weeks")`

`date_labels = "%m/%d/%y"` **equivalent to** `labels = label_date(format = "%m/%d/%y")`

if both are specified, `date_labels` and `date_breaks` override the other two.

```
demo_datetime(date_range("20170115", 30), date_labels = "%d/%m",
              date_breaks = "5 days")
```

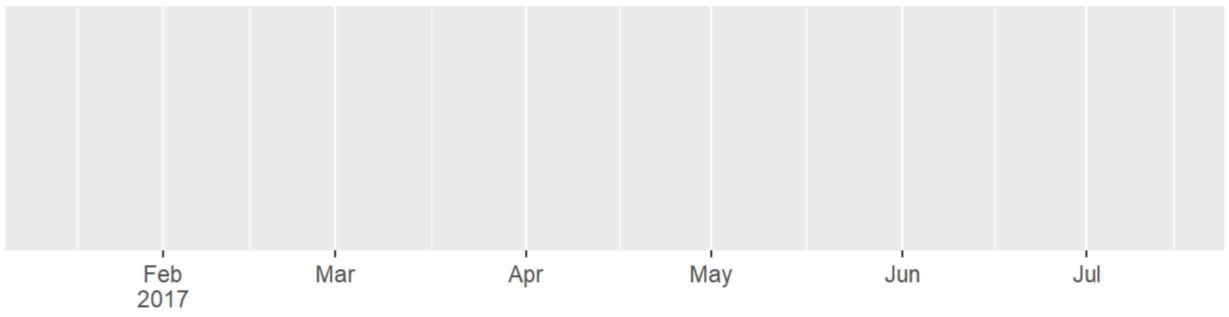
```
#> scale_x_datetime(date_labels = "%d/%m", date_breaks = "5 days")
```



mix 2 types of argument:

```
demo_datetime(date_range("20170115", 180),
              date_breaks = "month", labels = label_date_short())
```

```
#> scale_x_datetime(date_breaks = "month", labels = label_date_short())
```



label strings

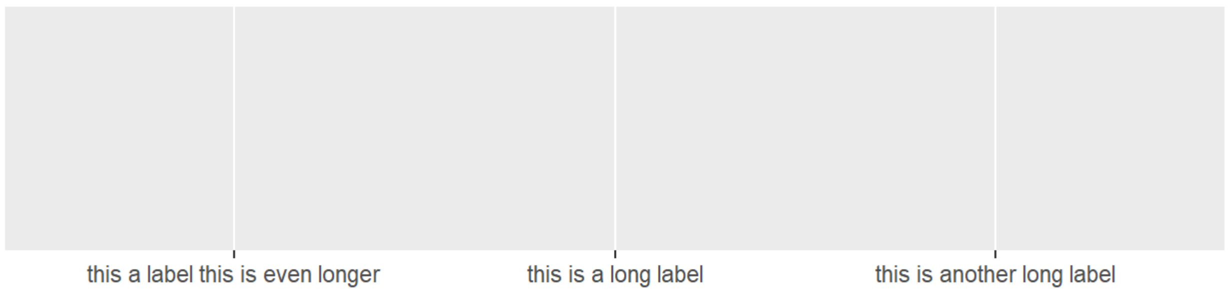
Use `label_wrap()` to wrap long strings:

```
label_wrap(width)
```

width: Number of characters per line

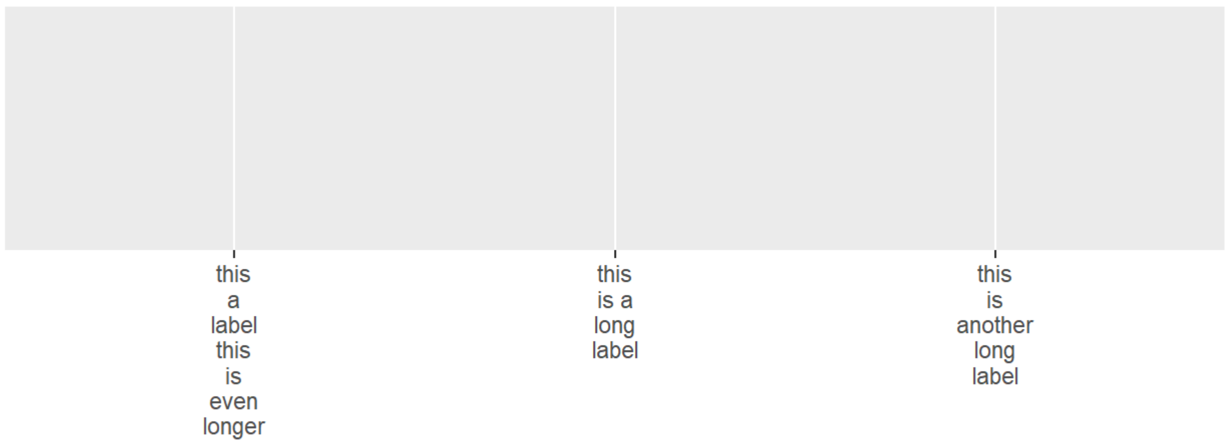
```
x <- c(
  "this is a long label",
  "this is another long label",
  "this a label this is even longer"
)
demo_discrete(x)
```

```
#> scale_x_discrete()
```



```
demo_discrete(x, labels = label_wrap(width = 5))
```

```
#> scale_x_discrete(labels = label_wrap(width = 5))
```



References

1. **ggplot2: Create Elegant Data Visualisations Using the Grammar of Graphics** [link]
   Wickham, H., Chang, W., Henry, L., Pedersen, T.L., Takahashi, K., Wilke, C., Woo, K. and Yutani, H., 2019.
2. **scales: Scale Functions for Visualization** [link]
   Wickham, H. and Seidel, D., 2019.
3. **labeling: Axis Labeling** [link]
   Talbot, J., 2014.
4. **An extension of Wilkinsonâ€™s algorithm for positioning tick labels on axes**
   Talbot, J., Lin, S. and Hanrahan, P., 2010. IEEE Transactions on visualization and computer graphics, Vol 16(6), pp. 1036--1043. IEEE.

Reuse

Citation

For attribution, please cite this work as

```
Yan (2019, Nov. 27). R Visualization Tips: Using the scales package. Retrieved from
https://bookdown.org/Maxine/ggplot2-maps/posts/2019-11-27-using-scales-package-to-modify-ggplot2-
scale/
```

BibTeX citation

```
@misc{yan2019using,
  author = {Yan, Qiushi},
  title = {R Visualization Tips: Using the scales package},
  url = {https://bookdown.org/Maxine/ggplot2-maps/posts/2019-11-27-using-scales-package-to-modify-
ggplot2-scale/},
  year = {2019}
}
```