# Package 'timeplyr'

December 12, 2023

**Title** Fast Tidy Tools for Date and Date-Time Manipulation

**Version** 0.5.0

**Description** A set of fast tidy functions for wrangling, completing and summarising date and date-time data. It combines 'tidyverse' syntax with the efficiency of 'data.table' and speed of 'collapse'.

**License** GPL (>= 2)

**BugReports** https://github.com/NicChr/timeplyr/issues

**Depends** R (>= 3.5.0)

**Imports** collapse (>= 2.0.0), cppdoubles, data.table (>= 1.14.8), dplyr (>= 1.1.0), ggplot2 (>= 3.4.0), lubridate (>= 1.9.0), pillar(>= 1.7.0), rlang (>= 1.0.0), stringr (>= 1.4.0), tidyselect (>= 1.2.0), timechange (>= 0.2.0), vctrs (>= 0.6.0)

**Suggests** bench, knitr, nycflights13, outbreaks, rmarkdown, testthat (>= 3.0.0), tidyr, zoo

**LinkingTo** cpp11

**Config/testthat/edition** 3

**Encoding** UTF-8

**RoxygenNote** 7.2.3

**NeedsCompilation** yes

**Author** Nick Christofides [aut, cre] (<https://orcid.org/0000-0002-9743-7342>)

**Maintainer** Nick Christofides <nick.christofides.r@gmail.com>

**Repository** CRAN

**Date/Publication** 2023-12-12 12:20:02 UTC

## R topics documented:

---

`timeplyr-package` *timeplyr: Fast Tidy Tools for Date and Date-Time Manipulation*

---

## Description

A framework for handling raw date & datetime data
using tidy best-practices from the tidyverse, the efficiency of data.table, and the speed of collapse.

You can learn more about the tidyverse, data.table and collapse using the links below

tidyverse

data.table

collapse

## Author(s)

**Maintainer**: Nick Christofides <nick.christofides.r@gmail.com> (ORCID)

## See Also

Useful links:

- Report bugs at https://github.com/NicChr/timeplyr/issues

## .time_units *Time units*

### Description

Time units

### Usage

```
.time_units

.period_units

.duration_units

.extra_time_units
```

### Format

An object of class `character` of length 21.

An object of class `character` of length 7.

An object of class `character` of length 11.

An object of class `character` of length 10.

## age_years *Accurate and efficient age calculation*

### Description

Correct calculation of ages in years using lubridate periods. Leap year calculations work as well.

### Usage

```
age_years(start, end = if (is_date(start)) Sys.Date() else Sys.time())

age_months(start, end = if (is_date(start)) Sys.Date() else Sys.time())
```

### Arguments

| | |
|---|---|
| `start` | Start date/datetime, typically date of birth. |
| `end` | End date/datetime. Default is current date/datetime. |

### Value

Integer vector of age in years or months.

## arithmetic_mean *Unweighted & weighted arithmetic, geometric and harmonic mean*

### Description

Convenience functions for fast unweighted and weighted mean calculations.

### Usage

```
arithmetic_mean(x, weights = NULL, na.rm = TRUE, ...)

geometric_mean(x, weights = NULL, na.rm = TRUE, ...)

harmonic_mean(x, weights = NULL, na.rm = TRUE, ...)
```

### Arguments

| | |
|---|---|
| x | numeric Vector. |
| weights | numeric Vector of weights.<br>Default is NULL which performs an unweighted mean. |
| na.rm | logical Value (Default is TRUE). |
| ... | Further arguments passed to collapse::fmean. |

### Value

numeric(min(length(x), 1)).

## asc *Helpers to sort variables in ascending or descending order*

### Description

An alternative to dplyr::desc() which is much faster for character vectors and factors.

### Usage

```
asc(x)

desc(x)
```

### Arguments

| | |
|---|---|
| x | Vector. |

## Value

A numeric vector that can be ordered in ascending or descending order.
Useful in `dplyr::arrange()` or `farrange()`.

## Examples

```
library(dplyr)
library(timeplyr)

starwars %>%
  fdistinct(mass) %>%
  farrange(desc(mass))
```

---

`calendar` *Create a table of common time units from a date or datetime sequence.*

---

## Description

Create a table of common time units from a date or datetime sequence.

## Usage

```
calendar(
  x,
  label = TRUE,
  week_start = getOption("lubridate.week.start", 1),
  fiscal_start = getOption("lubridate.fiscal.start", 1),
  name = "time"
)

add_calendar(
  data,
  time = NULL,
  label = TRUE,
  week_start = getOption("lubridate.week.start", 1),
  fiscal_start = getOption("lubridate.fiscal.start", 1)
)
```

## Arguments

| | |
|---|---|
| `x` | date or datetime vector. |
| `label` | Logical. Should labelled (ordered factor) versions of week day and month be returned? Default is `TRUE`. |
| `week_start` | day on which week starts following ISO conventions - 1 means Monday, 7 means Sunday (default). When `label = TRUE`, this will be the first level of the returned factor. You can set `lubridate.week.start` option to control this parameter globally. |

| | |
|---|---|
| fiscal_start | Numeric indicating the starting month of a fiscal year. |
| name | Name of date/datetime column. |
| data | A data frame. |
| time | Time variable. |

## Value

An object of class tibble.

## Examples

```
library(timeplyr)
library(lubridate)

# Create a calendar for the current year
from <- floor_date(today(), unit = "year")
to <- ceiling_date(today(), unit = "year", change_on_boundary = TRUE) - days(1)

my_seq <- time_seq(from, to, time_by = "day")
calendar(my_seq)
```

---

# cpp_which *Efficient alternative to* which()

---

## Description

Exactly the same as which() but more memory efficient.

## Usage

```
cpp_which(x, invert = FALSE)
```

## Arguments

| | |
|---|---|
| x | A logical vector. |
| invert | If TRUE, indices of values that are not TRUE are returned (including NA). If FALSE (the default), only TRUE indices are returned. |

## Details

This implementation is similar in speed to which() but usually more memory efficient.

## Value

An unnamed integer vector.

## Examples

```
library(timeplyr)
library(bench)

x <- sample(c(TRUE, FALSE), 1e05, TRUE)
x[sample.int(1e05, round(1e05/3))] <- NA

mark(cpp_which(TRUE), which(TRUE))
mark(cpp_which(FALSE), which(FALSE))
mark(cpp_which(logical()), which(logical()))
mark(cpp_which(x), which(x), iterations = 20)
mark(base = which(is.na(match(x, TRUE))),
     collapse = collapse::whichNA(collapse::fmatch(x, TRUE, overid = 2L)),
     timeplyr = cpp_which(x, invert = TRUE),
     iterations = 20)
```

---

# crossed_join *A* `do.call()` *and* `data.table::CJ()` *method*

---

## Description

This function operates like `do.call(CJ, ...)` and accepts a list or data.frame as an argument. It has less overhead for small joins, especially when `unique = FALSE` and `as_dt = FALSE`. `NA`s are by default sorted last.

## Usage

```
crossed_join(
  X,
  sort = FALSE,
  unique = TRUE,
  as_dt = TRUE,
  strings_as_factors = FALSE,
  log_limit = 8
)
```

## Arguments

| | |
|---|---|
| `X` | A list or data frame. |
| `sort` | Should the expansion be sorted? By default it is `FALSE`. |
| `unique` | Should unique values across each column or list element be taken? By default this is `TRUE`. |
| `as_dt` | Should result be a `data.table`? By default this is `TRUE`. If `FALSE` a list is returned. |

strings_as_factors
: Should strings be converted to factors before expansion? The default is FALSE but setting to TRUE can offer a significant speed improvement.

log_limit
: The maximum log10 limit for expanded number of rows. Anything >= this results in an error.

## Details

An important note is that currently NAs are sorted last and therefore a key is not set.

## Value

A data.table or list object.

## Examples

```
library(timeplyr)

crossed_join(list(1:3, -2:2))
crossed_join(iris)
```

---

# duplicate_rows *Find duplicate rows*

---

## Description

Find duplicate rows

## Usage

```
duplicate_rows(
  data,
  ...,
  .keep_all = FALSE,
  .both_ways = FALSE,
  .add_count = FALSE,
  .drop_empty = FALSE,
  sort = FALSE,
  .by = NULL,
  .cols = NULL
)

fduplicates(
  data,
  ...,
  .keep_all = FALSE,
  .both_ways = FALSE,
```

```
  .add_count = FALSE,
  .drop_empty = FALSE,
  sort = FALSE,
  .by = NULL,
  .cols = NULL
)

fduplicates2(
  data,
  ...,
  .keep_all = FALSE,
  .both_ways = FALSE,
  .add_count = FALSE,
  .drop_empty = FALSE,
  .by = NULL
)
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| ... | Variables used to find duplicate rows. |
| .keep_all | If TRUE then all columns of data frame are kept, default is FALSE. |
| .both_ways | If TRUE then duplicates and non-duplicate first instances are retained. The default is FALSE which returns only duplicate rows.<br>Setting this to TRUE can be particularly useful when examining the differences between duplicate rows. |
| .add_count | If TRUE then a count column is added to denote the number of duplicates (including first non-duplicate instance). The naming convention of this column follows dplyr::add_count(). |
| .drop_empty | If TRUE then empty rows with all NA values are removed. The default is FALSE. |
| sort | Should result be sorted? If FALSE (the default), then rows are returned in the exact same order as they appear in the data. If TRUE then the duplicate rows are sorted. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| .cols | (Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |

## Details

This function works like dplyr::distinct() in its handling of arguments and data-masking but returns duplicate rows. In certain situations in can be much faster than data %>% group_by() %>% filter(n() > 1) when there are many groups. fduplicates2() returns the same output but uses a different method which utilises joins and is written almost entirely using dplyr.

## Value

A data.frame of duplicate rows.

## See Also

fcount group_collapse fdistinct

## Examples

```
library(dplyr)
library(timeplyr)
library(ggplot2)

# Duplicates across all columns
diamonds %>%
  duplicate_rows()
# Alternatively with row ids
diamonds %>%
  filter(frowid(.) > 1)
# Diamonds with the same dimensions
diamonds %>%
  duplicate_rows(x, y, z)
# Can use tidyverse select notation
diamonds %>%
  duplicate_rows(across(where(is.factor)), .keep_all = FALSE)
# Similar to janitor::get_dupes()
diamonds %>%
  duplicate_rows(.add_count = TRUE)
# Keep the first instance of each duplicate row
diamonds %>%
  duplicate_rows(.both_ways = TRUE)
# Same as the below
diamonds %>%
  fadd_count(across(everything())) %>%
  filter(n > 1)
```

---

# edf *Grouped empirical cumulative distribution function applied to data*

---

## Description

Like `dplyr::cume_dist(x)` and `ecdf(x)(x)` but with added grouping and weighting functionality. You can calculate the empirical distribution of x using aggregated data by supplying frequency weights. No expansion occurs which makes this function extremely efficient for this type of data, of which plotting is a common application.

## Usage

```
edf(x, g = NULL, wt = NULL)
```

## Arguments

| | |
|---|---|
| x | Numeric vector. |
| g | Numeric vector of group IDs. |
| wt | Frequency weights. |

## Value

A numeric vector the same length as x.

## Examples

```
library(timeplyr)
library(dplyr)
library(ggplot2)

set.seed(9123812)
x <- sample(seq(-10, 10, 0.5), size = 10^2, replace = TRUE)
plot(sort(edf(x)))
all.equal(edf(x), ecdf(x)(x))
all.equal(edf(x), cume_dist(x))

# Manual ECDF plot using only aggregate data
y <- rnorm(100, 10)
grid <- time_span(y, time_by = 0.1, time_floor = TRUE)
counts <- time_countv(y, time_by = 0.1, time_floor = TRUE, complete = TRUE)$n
edf <- edf(grid, wt = counts)
# Trivial here as this is the same
all.equal(unname(cumsum(counts)/sum(counts)), edf)

# Full ecdf
tibble(x) %>%
  ggplot(aes(x = y)) +
  stat_ecdf()
# Approximation using aggregate only data
tibble(grid, edf) %>%
  ggplot(aes(x = grid, y = edf)) +
  geom_step()

# Grouped example
g <- sample(letters[1:3], size = 10^2, replace = TRUE)

edf1 <- tibble(x, g) %>%
  mutate(edf = cume_dist(x),
         .by = g) %>%
  pull(edf)
edf2 <- edf(x, g = g)
all.equal(edf1, edf2)
```

farrange *A* collapse *version of* dplyr::arrange()

## Description

This is a fast and near-identical alternative to dplyr::arrange() using the collapse package.

desc() is like dplyr::desc() but works faster when called directly on vectors.

## Usage

```
farrange(data, ..., .by = NULL, .by_group = FALSE, .cols = NULL)
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| ... | Variables to arrange by. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidyselect. |
| .by_group | If TRUE the sorting will be first done by the group variables. |
| .cols | (Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |

## Details

farrange() is inspired by collapse::roworder() but also supports dplyr style data-masking which makes it a closer replacement to dplyr::arrange().

You can use desc() interchangeably with dplyr and timeplyr.
arrange(iris, desc(Species)) uses dplyr's version.
farrange(iris, desc(Species)) uses timeplyr's version.

farrange() is faster when there are many groups or a large number of rows.

## Value

A sorted data.frame.

`fcount` *A fast replacement to dplyr::count()*

## Description

Near-identical alternative to `dplyr::count()`.

## Usage

```
fcount(
  data,
  ...,
  wt = NULL,
  sort = FALSE,
  order = TRUE,
  name = NULL,
  .by = NULL,
  .cols = NULL
)

fadd_count(
  data,
  ...,
  wt = NULL,
  sort = FALSE,
  order = TRUE,
  name = NULL,
  .by = NULL,
  .cols = NULL
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame. |
| `...` | Variables to group by. |
| `wt` | Frequency weights. Can be `NULL` or a variable:<br>• If `NULL` (the default), counts the number of rows in each group.<br>• If a variable, computes `sum(wt)` for each group. |
| `sort` | If `TRUE`, will show the largest groups at the top. |
| `order` | Should the groups be calculated as ordered groups? If `FALSE`, this will return the groups in order of first appearance, and in many cases is faster. If `TRUE` (the default), the groups are returned in sorted order, exactly the same way as `dplyr::count`. |
| `name` | The name of the new column in the output. If there's already a column called `n`, it will use `nn`. If there's a column called `n` and `nn`, it'll use `nnn`, and so on, adding `n`s until it gets a new name. |

| | |
|---|---|
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| .cols | (Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |

## Details

This is a fast and near-identical alternative to dplyr::count() using the collapse package. Unlike collapse::fcount(), this works very similarly to dplyr::count(). The only main difference is that anything supplied to wt is recycled and added as a data variable. Other than that everything works exactly as the dplyr equivalent.

fcount() and fadd_count() can be up to >100x faster than the dplyr equivalents.

## Value

A data.frame of frequency counts by group.

## Examples

```
library(timeplyr)
library(dplyr)

iris %>%
  fcount()
iris %>%
  fadd_count(name = "count") %>%
  fslice_head(n = 10)
iris %>%
  group_by(Species) %>%
  fcount()
iris %>%
  fcount(Species)
iris %>%
  fcount(across(where(is.numeric), mean))

### Sorting behaviour

# Sorted by group
starwars %>%
  fcount(hair_color)
# Sorted by frequency
starwars %>%
  fcount(hair_color, sort = TRUE)
# Groups sorted by order of first appearance (faster)
starwars %>%
  fcount(hair_color, order = FALSE)
```

fdistinct *Find distinct rows*

## Description

Like `dplyr::distinct()` but faster when lots of groups are involved.

## Usage

```
fdistinct(
  data,
  ...,
  .keep_all = FALSE,
  sort = FALSE,
  order = sort,
  .by = NULL,
  .cols = NULL
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame. |
| `...` | Variables used to find distinct rows. |
| `.keep_all` | If `TRUE` then all columns of data frame are kept, default is `FALSE`. |
| `sort` | Should result be sorted? Default is `FALSE`. When `order = FALSE` this option has no effect on the result. |
| `order` | Should the groups be calculated as ordered groups? Setting to `TRUE` may sometimes offer a speed benefit, but usually this is not the case. The default is `FALSE`. |
| `.by` | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| `.cols` | (Optional) alternative to `...` that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |

## Value

A `data.frame` of distinct groups.

## See Also

group_collapse duplicate_rows

## Examples

```
library(dplyr)
library(timeplyr)
library(ggplot2)

mpg %>%
  distinct(manufacturer)
mpg %>%
  fdistinct(manufacturer)
```

---

fexpand *Fast versions of* `tidyr::expand()` *and* `tidyr::complete()`.

---

## Description

Fast versions of `tidyr::expand()` and `tidyr::complete()`.

## Usage

```
fexpand(
  data,
  ...,
  expand_type = c("crossing", "nesting"),
  sort = FALSE,
  .by = NULL,
  keep_class = TRUE,
  log_limit = 8
)

fcomplete(
  data,
  ...,
  expand_type = c("crossing", "nesting"),
  sort = FALSE,
  .by = NULL,
  keep_class = TRUE,
  fill = NA,
  log_limit = 8
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame |
| `...` | Variables to expand |

| | |
|---|---|
| expand_type | Type of expansion to use where "nesting" finds combinations already present in the data (exactly the same as using distinct() but fexpand() allows new variables to be created on the fly and columns are sorted in the order given. "crossing" finds all combinations of values in the group variables. |
| sort | Logical. If TRUE expanded/completed variables are sorted. The default is FALSE. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| keep_class | Logical. If TRUE then the class of the input data is retained. If FALSE, which is sometimes faster, a data.table is returned. |
| log_limit | The maximum log10 number of rows that can be expanded. Anything exceeding this will throw an error. |
| fill | A named list containing value-name pairs to fill the named implicit missing values. |

## Details

For un-grouped data fexpand() is similar in speed to tidyr::expand(). When the data contain many groups, fexpand() is much much faster (see examples).

The 2 main differences between fexpand() and tidyr::expand() are that:

- tidyr style helpers like nesting() and crossing() are ignored. The type of expansion used is controlled through expand_type and applies to all supplied variables.
- Expressions are first calculated on the entire ungrouped dataset before being expanded but within-group expansions will work on variables that already exist in the dataset. For example, iris %>% group_by(Species) %>% fexpand(Sepal.Length, Sepal.Width) will perform a grouped expansion but iris %>% group_by(Species) %>% fexpand(range(Sepal.Length)) will not.

For efficiency, when supplying groups, expansion is done on a by-group basis only if there are 2 or more variables that aren't part of the grouping. The reason is that a by-group calculation does not need to be done with 1 expansion variable as all combinations across groups already exist against that 1 variable. When expand_type = "nesting" groups are ignored for speed purposes as the result is the same.

An advantage of fexpand() is that it returns a data frame with the same class as the input. It also uses data.table for memory efficiency and collapse for speed.

A future development for fcomplete() would be to only fill values of variables that correspond only to both additional completed rows and rows that match the expanded rows, are filled in. For example, iris %>% mutate(test = NA_real_) %>% complete(Sepal.Length = 0:100, fill = list(test = 0)) fills in all NA values of test, whereas iris %>% mutate(test = NA_real_) %>% fcomplete(Sepal.Length = 0:100, fill = list(test = 0)) should only fill in values of test that correspond to Sepal.Length values of 0:100.

An additional note to add when expand_type = "nesting" is that if one of the supplied variables in ... does not exist in the data, but can be recycled to the length of the data, then it is added and treated as a data variable.

## Value

A data.frame of expanded groups.

## Examples

```
library(timeplyr)
library(dplyr)
library(lubridate)
library(nycflights13)

flights %>%
  fexpand(origin, dest)
flights %>%
  fexpand(origin, dest, sort = FALSE)

# Grouped expansions example
# 1 extra group (carrier) this is very quick
flights %>%
  group_by(origin, dest, tailnum) %>%
  fexpand(carrier)
```

---

# fgroup_by *'collapse' version of* dplyr::group_by()

---

## Description

This works the exact same as dplyr::group_by() and typically performs around the same speed but uses slightly less memory.

## Usage

```
fgroup_by(
  data,
  ...,
  .add = FALSE,
  order = TRUE,
  .by = NULL,
  .cols = NULL,
  .drop = TRUE
)
```

## Arguments

| | |
|---|---|
| data | data frame. |
| ... | Variables to group by. |
| .add | Should groups be added to existing groups? Default is FALSE. |
| order | Should groups be ordered? If FALSE groups will be ordered based on first-appearance. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidyselect. |

| | |
|---|---|
| .cols | (Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |
| .drop | Should unused factor levels be dropped? Default is TRUE. |

## Details

fgroup_by() works almost exactly like the 'dplyr' equivalent. An attribute "sorted" (TRUE or FALSE) is added to the group data to signify if the groups are sorted or not.

## Value

A grouped_df.

---

# fn *Supplementary fast statistical functions,* collapse *style*

---

## Description

Supplementary fast statistical functions, collapse style

## Usage

```
fn(x, g = NULL, sort = TRUE, expand = FALSE, use.g.names = !expand)

fcummean(x, g = NULL, na.rm = FALSE, ...)

fnmiss(x, g = NULL, sort = TRUE, use.g.names = TRUE, na.rm = FALSE)

fprop_complete(x, g = NULL, sort = TRUE, use.g.names = TRUE, na.rm = FALSE)

fprop_missing(x, g = NULL, sort = TRUE, use.g.names = TRUE)
```

## Arguments

| | |
|---|---|
| x | A vector or data frame. In the case of fn() this can be left unused as long as g is not NULL, otherwise it is used as a template with which to calculate group sizes. For example, is x is a vector, lengths are calculated per-group, and if x is a data frame, numbers of rows are calculated per-group. |
| g | Object to be used for grouping, passed directly to collapse::GRP(). |
| sort | Should the grouped counts be ordered by the sorted groups? If FALSE the result is ordered by groups of first appearance. |
| expand | Should the grouped counts be expanded to match the length and order of the data? Default is FALSE. |
| use.g.names | If TRUE group names are added to the result as names. This only applies to fn(). Default is TRUE. |
| na.rm | Should NA values be removed? Default is FALSE. |
| ... | Additional parameters passed to collapse::fsum(). |

### Details

`fn()` Is different to the other `collapse` fast statistical functions because given a data frame, it operates on the entire data frame, instead of column-wise. It is similar to the the other statistical functions in that order of the returned groups matches that of `collapse::fnobs()`. For example, `collapse::GRPN(c(2, 2, 1), expand = FALSE)` returns `c(2, 1)` whereas `fn(g = c(2, 2, 1))` returns `c(1, 2)` which is similar to `collapse::fnobs(rep(1, 3), g = c(2, 2, 1))`.
While `fn()` is not entirely useful as a function, it is useful for internal code that utilises `GRP` objects.

---

## `frowid` *Fast grouped row numbers*

---

### Description

Very fast row numbers by group.

### Usage

```
frowid(x, ascending = TRUE)
```

### Arguments

| | |
|---|---|
| `x` | A vector, data frame or `GRP` object. |
| `ascending` | When `ascending = TRUE` the row IDs are in increasing order. When `ascending = FALSE` the row IDs are in decreasing order. |

### Details

`frowid()` is like `data.table::rowid()` but uses an alternative method for calculating row numbers. When `x` is a collapse `GRP` object, it is considerably faster. It is also faster for character vectors.

### Value

An integer vector of row IDs.

### See Also

row_id add_row_id

### Examples

```
library(timeplyr)
library(dplyr)
library(data.table)
library(nycflights13)

# Simple row numbers
head(row_id(flights))
# Row numbers by origin
```

```
head(frowid(flights$origin))
head(row_id(flights, origin))

# Fast duplicate rows
head(frowid(flights) > 1)

# With data frames, better to use row_id()
flights %>%
  add_row_id() %>% # Plain row ids
  add_row_id(origin, dest, .name = "grouped_row_id") # Row IDs by group
```

---

`fselect` *Fast* `dplyr::select()/dplyr::rename()`

---

## Description

`fselect()` operates the exact same way as `dplyr::select()` and can be used naturally with `tidy-select` helpers. It uses collapse to perform the actual selecting of variables and is considerably faster than dplyr for selecting exact columns, and even more so when supplying the `.cols` argument.

## Usage

```
fselect(data, ..., .cols = NULL)

frename(data, ..., .cols = NULL)
```

## Arguments

| | |
|---|---|
| `data` | A data frame. |
| `...` | Variables to select using `tidy-select`. See `?dplyr::select` for more info. |
| `.cols` | (Optional) faster alternative to `...` that accepts a named character vector or numeric vector.<br>No checks on duplicates column names are done when using `.cols`.<br>If speed is an expensive resource, it is recommended to use this. |

## Value

A `data.frame` of selected columns.

## Examples

```
library(timeplyr)
library(dplyr)

df <- slice_head(iris, n = 5)
fselect(df, Species, SL = Sepal.Length)
```

```
fselect(df, .cols = c("Species", "Sepal.Length"))
fselect(df, all_of(c("Species", "Sepal.Length")))
fselect(df, 5, 1)
fselect(df, .cols = c(5, 1))
df %>%
  fselect(where(is.numeric))
```

---

`fskim` *Fast alternative to* `skimr::skim()`

---

## Description

Inspired by the brilliant `skimr` package, this is a fast alternative that provides an un-grouped data frame summary.

## Usage

```
fskim(data, hist = FALSE)
```

## Arguments

| | |
|---|---|
| `data` | A data frame. |
| `hist` | Logical. If `TRUE`, histogram spark graphs are produced in the numeric summary. |

## Details

`collapse` is used to compute the summary statistics and `data.table` is used to wrangle the data frames.
Character vectors are internally converted to factors using `collapse::qF()`.

## Value

A list of length 7 with the elements:

- `nrow` - Number of rows
- `ncol` - Number of columns
- `logical` - A `tibble` summary of the logical columns.
- `numeric` - A `tibble` summary of the numeric columns.
- `date` - A `tibble` summary of the date columns.
- `datetime` - A `tibble` summary of the datetime columns.
- `categorical` - A `tibble` summary of the categorical columns.

## Examples

```
library(timeplyr)
library(nycflights13)

fskim(flights)
```

---

`fslice` *Faster* `dplyr::slice()`

---

## Description

When there are lots of groups, the `fslice()` functions are much faster.

## Usage

```
fslice(data, ..., .by = NULL, keep_order = FALSE, sort_groups = TRUE)

fslice_head(
  data,
  ...,
  n,
  prop,
  .by = NULL,
  keep_order = FALSE,
  sort_groups = TRUE
)

fslice_tail(
  data,
  ...,
  n,
  prop,
  .by = NULL,
  keep_order = FALSE,
  sort_groups = TRUE
)

fslice_min(
  data,
  order_by,
  ...,
  n,
  prop,
  .by = NULL,
  with_ties = TRUE,
  na_rm = FALSE,
```

```
  keep_order = FALSE,
  sort_groups = TRUE
)

fslice_max(
  data,
  order_by,
  ...,
  n,
  prop,
  .by = NULL,
  with_ties = TRUE,
  na_rm = FALSE,
  keep_order = FALSE,
  sort_groups = TRUE
)

fslice_sample(
  data,
  n,
  replace = FALSE,
  prop,
  .by = NULL,
  keep_order = FALSE,
  sort_groups = TRUE,
  weights = NULL,
  seed = NULL
)
```

## Arguments

| | |
|---|---|
| data | Data frame |
| ... | See ?dplyr::slice for details. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| keep_order | Should the sliced data frame be returned in its original order? The default is FALSE. |
| sort_groups | If TRUE (the default) the by-group slices will be done in order of the sorted groups. If FALSE the group order is determined by first-appearance in the data. |
| n | Number of rows. |
| prop | Proportion of rows. |
| order_by | Variables to order by. |
| with_ties | Should ties be kept together? The default is TRUE. |
| na_rm | Should missing values in fslice_max() and fslice_min() be removed? The default is FALSE. |

| | |
|---|---|
| replace | Should fslice_sample() sample with or without replacement? Default is FALSE, without replacement. |
| weights | Probability weights used in fslice_sample(). |
| seed | Seed number defining RNG state. If supplied, this is only applied **locally** within the function and the seed state isn't retained after sampling. To clarify, whatever seed state was in place before the function call, is restored to ensure seed continuity. If left NULL (the default), then the seed is never modified. |

## Details

fslice() and friends allow for more flexibility in how you order the by-group slicing.
Furthermore, you can control whether the returned data frame is sliced in the order of the supplied row indices, or whether the original order is retained (like dplyr::filter()).

In fslice(), when length(n) == 1, an optimised method is implemented that internally uses list_subset(), a fast function for extracting single elements from single-level lists that contain vectors of the same type, e.g. integer.

fslice_head() and fslice_tail() are very fast with large numbers of groups.

fslice_sample() is arguably more intuitive as it by default resamples each entire group without replacement, without having to specify a maximum group size like in dplyr::slice_sample().

## Value

A data.frame of specified rows.

## Examples

```
library(timeplyr)
library(dplyr)
library(nycflights13)

flights <- flights %>%
  group_by(origin, dest)

# First row repeated for each group
flights %>%
  fslice(1, 1)
# First row per group
flights %>%
  fslice_head(n = 1)
# Last row per group
flights %>%
  fslice_tail(n = 1)
# Earliest flight per group
flights %>%
  fslice_min(time_hour, with_ties = FALSE)
# Last flight per group
flights %>%
  fslice_max(time_hour, with_ties = FALSE)
# Random sample without replacement by group
# (or stratified random sampling)
```

```
flights %>%
  fslice_sample()
```

---

gcd *Greatest common divisor and smallest common multiple*

---

## Description

Fast greatest common divisor and smallest common multiple using the Euclidean algorithm.

`gcd()` returns the greatest common divisor.
`scm()` returns the smallest common multiple.
`gcd_diff()` returns the greatest common divisor of numeric differences.

## Usage

```
gcd(
  x,
  tol = sqrt(.Machine$double.eps),
  na_rm = TRUE,
  round = TRUE,
  break_early = TRUE
)

scm(x, tol = sqrt(.Machine$double.eps), na_rm = TRUE)

gcd_diff(
  x,
  lag = 1L,
  fill = NA,
  tol = sqrt(.Machine$double.eps),
  na_rm = TRUE,
  round = TRUE,
  break_early = TRUE
)
```

## Arguments

| | |
|---|---|
| `x` | A numeric vector. |
| `tol` | Tolerance. This must be a single positive number strictly less than 1. |
| `na_rm` | If `TRUE` the default, `NA` values are ignored. |
| `round` | If `TRUE` the output is rounded as `round(gcd, digits)` where digits is `ceiling(abs(log10(tol))) + 1`.<br>This can potentially reduce floating point errors on further calculations.<br>The default is `TRUE`. |

| | |
|---|---|
| break_early | This is experimental and applies only to floating-point numbers. When TRUE the algorithm will end once gcd > 0 && gcd < 2 * tol. This can offer a tremendous speed improvement. If FALSE the algorithm finishes once it has gone through all elements of x. The default is TRUE.<br>For integers, the algorithm always breaks early once gcd > 0 && gcd <= 1. |
| lag | Lag of differences. |
| fill | Value to initialise the algorithm for gcd_diff(). |

## Details

### Method:

*GCD:*
The GCD is calculated using a binary function that takes input GCD(gcd, x[i + 1]) where the output of this function is passed as input back into the same function iteratively along the length of x. The first gcd value is x[1].
Zeroes are handled in the following way:
GCD(0, 0) = 0
GCD(a, 0) = a

This has the nice property that zeroes are essentially ignored.

*SCM:*
This is calculated using the GCD and the formula is:
SCM(x, y) = (abs(x) / GCD(x, y) ) * abs(y)
If you want to calculate the gcd & lcm for 2 values instead of a vector of values, use the internal functions cpp_gcd2 and cpp_lcm2. You can then easily write a vectorised method using these.

## Value

A number representing the GCD or SCM.

## Examples

```
library(timeplyr)
library(bench)

gcd(c(0, 5, 25))
mark(gcd(c(0, 5, 25)))

x <- rnorm(10^6)
gcd(x)
gcd(x, round = TRUE)
mark(gcd(x))
```

`get_time_delay` *Get summary statistics of time delay*

## Description

The output is a `list` containing summary statistics of time delay between two date/datetime vectors. This can be especially useful in estimating reporting delay for example.

- **data** - A data frame containing the origin, end and calculated time delay.
- **unit** - The chosen time unit.
- **num** - The number of time units.
- **summary** - `tibble` with summary statistics.
- **delay** - `tibble` containing the empirical cumulative distribution function values by time delay.
- **plot** - A `ggplot` of the time delay distribution.

## Usage

```
get_time_delay(
  data,
  origin,
  end,
  time_by = 1L,
  time_type = getOption("timeplyr.time_type", "auto"),
  min_delay = -Inf,
  max_delay = Inf,
  probs = c(0.25, 0.5, 0.75, 0.95),
  .by = NULL,
  include_plot = TRUE,
  x_scales = "fixed",
  bw = "SJ",
  ...
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame. |
| `origin` | Origin date variable. |
| `end` | End date variable. |
| `time_by` | Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"`<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`. |

- Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`.

`time_type` If "auto", `periods` are used for the time expansion when days, weeks, months or years are specified, and `durations` are used otherwise.

`min_delay` The minimum acceptable delay, all delays less than this are removed before calculation. Default is `min_delay = -Inf`.

`max_delay` The maximum acceptable delay, all delays greater than this are removed before calculation. Default is `max_delay = Inf`.

`probs` Probabilities used in the quantile summary. Default is `probs = c(0.25, 0.5, 0.75, 0.95)`.

`.by` (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.

`include_plot` Should a `ggplot` graph of delay distributions be included in the output?

`x_scales` Option to control how the x-axis is displayed for multiple facets. Choices are "fixed" or "free_x".

`bw` The smoothing bandwidth selector for the Kernel Density estimator. If numeric, the standard deviation of the smoothing kernel. If character, a rule to choose the bandwidth. See `?stats::bw.nrd` for more details. The default has been set to "SJ" which implements the Sheather & Jones (1991) method, as recommended by the R team `?stats::density`. This differs from the default implemented by `stats::density()` which uses Silverman's rule-of-thumb.

`...` Further arguments to be passed on to `ggplot2::geom_density()`.

## Value

A list containing summary data, summary statistics and an optional `ggplot`.

## Examples

```
library(timeplyr)
library(outbreaks)
library(dplyr)

ebola_linelist <- ebola_sim_clean$linelist

# Incubation period distribution

# 95% of individuals experienced an incubation period of <= 26 days
inc_distr_days <- ebola_linelist %>%
  get_time_delay(date_of_infection,
                 date_of_onset,
                 time_by = "days")
head(inc_distr_days$data)
inc_distr_days$unit
inc_distr_days$num
inc_distr_days$summary
head(inc_distr_days$delay) # ECDF and freq by delay
inc_distr_days$plot
```

```
# Can change bandwidth selector
inc_distr_days <- ebola_linelist %>%
  get_time_delay(date_of_infection,
                 date_of_onset,
                 time_by = "day",
                 bw = "nrd")
inc_distr_days$plot

# Can choose any time units
inc_distr_weeks <- ebola_linelist %>%
  get_time_delay(date_of_infection,
                 date_of_onset,
                 time_by = "weeks",
                 bw = "nrd")
inc_distr_weeks$plot
```

---

`groups_equal` *Are groups equal?*

---

## Description

This function is a very fast utility for quickly checking if the group data between 2 data frames are identical.

## Usage

```
groups_equal(x, y)
```

## Arguments

| | |
|---|---|
| `x` | A `grouped_df`. |
| `y` | A `grouped_df`. |

## Value

A logical indicating whether the groups are identical or not.

## Examples

```
library(dplyr)
library(timeplyr)

df <- iris %>%
  group_by(Species)
df2 <- iris %>%
  fslice_sample(seed = 1777) %>%
  group_by(Species)
```

```
groups_equal(iris, iris)
groups_equal(df, df)
groups_equal(df, df2)
```

---

`group_collapse` *Key group information*

---

## Description

Key group information

## Usage

```
group_collapse(
  data,
  ...,
  order = TRUE,
  sort = FALSE,
  ascending = TRUE,
  .by = NULL,
  .cols = NULL,
  id = TRUE,
  size = TRUE,
  loc = TRUE,
  start = TRUE,
  end = TRUE,
  drop = TRUE
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame or vector. |
| `...` | Additional groups using tidy `data-masking` rules.<br>To specify groups using `tidyselect`, simply use the `.by` argument. |
| `order` | Should the groups be ordered? **THE PHYSICAL ORDER OF THE DATA IS NOT CHANGED.**<br>When order is `TRUE` (the default) the group IDs will be ordered but not sorted. If `FALSE` the order of the group IDs will be based on first appearance. |
| `sort` | Should the data frame be sorted by the groups? |
| `ascending` | Should groups be ordered in ascending order? Default is `TRUE` and only applies when `order = TRUE`. |
| `.by` | Alternative way of supplying groups using `tidyselect` notation. This is kept to be consistent with other functions. |

| | |
|---|---|
| .cols | (Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |
| id | Should group IDs be added? Default is TRUE. |
| size | Should group sizes be added? Default is TRUE. |
| loc | Should group locations be added? Default is TRUE. |
| start | Should group start locations be added? Default is TRUE. |
| end | Should group end locations be added? Default is TRUE. |
| drop | Should unused factor levels be dropped? Default is TRUE. |

## Details

group_collapse() is similar to dplyr::group_data() but differs in 3 key regards:

- The output tries to convey as much information about the groups as possible. By default, like dplyr, the groups are ordered, but unlike dplyr they are not sorted, which conveys information on order-of-first-appearance in the data. In addition to group locations, group sizes and start indices are returned.
- There is more flexibility in specifying how the groups are ordered and/or sorted.
- collapse is used to obtain the grouping structure, which is very fast.

There are 3 ways to specify the groups:

- Using ... which utilises tidy data-masking.
- Using .by which utilises tidyselect.
- Using .cols which accepts a named character/integer vector.

## Value

A tibble of unique groups and an integer ID uniquely identifying each group.

## Examples

```
library(timeplyr)
library(dplyr)

iris <- dplyr::as_tibble(iris)
group_collapse(iris) # No groups
group_collapse(iris, Species) # Species groups

iris %>%
  group_by(Species) %>%
  group_collapse() # Same thing

# Group entire data frame
group_collapse(iris, .by = everything())
```

group_id *Fast group IDs*

## Description

These are tidy-based functions for calculating group IDs, row IDs and group orders.

- `group_id()` returns an integer vector of group IDs the same size as the data.
- `row_id()` returns an integer vector of row IDs.
- `group_order()` returns the order of the groups.

The add_ variants add a column of group IDs/row IDs/group orders.

## Usage

```
group_id(
  data,
  ...,
  order = TRUE,
  ascending = TRUE,
  .by = NULL,
  .cols = NULL,
  as_qg = FALSE
)

add_group_id(
  data,
  ...,
  order = TRUE,
  ascending = TRUE,
  .by = NULL,
  .cols = NULL,
  .name = NULL,
  as_qg = FALSE
)

row_id(data, ..., ascending = TRUE, .by = NULL, .cols = NULL)

add_row_id(data, ..., ascending = TRUE, .by = NULL, .cols = NULL, .name = NULL)

group_order(data, ..., ascending = TRUE, .by = NULL, .cols = NULL)

add_group_order(
  data,
  ...,
  ascending = TRUE,
```

```
  .by = NULL,
  .cols = NULL,
  .name = NULL
)
```

## Arguments

| | |
|---|---|
| data | A data frame or vector. |
| ... | Additional groups using tidy data-masking rules.<br>To specify groups using tidyselect, simply use the .by argument. |
| order | Should the groups be ordered? **THE PHYSICAL ORDER OF THE DATA IS NOT CHANGED.**<br>When order is TRUE (the default) the group IDs will be ordered but not sorted. The expression<br>`identical(order(x, na.last = TRUE), order(group_id(x, order = TRUE)))`<br>or in the case of a data frame<br>`identical(order(x1, x2, x3, na.last = TRUE), order(group_id(data, x1, x2, x3, order = TRUE)))`<br>should always hold.<br>If FALSE the order of the group IDs will be based on first appearance. |
| ascending | Should the group order be ascending or descending? The default is TRUE.<br>For row_id() this determines if the row IDs are increasing or decreasing.<br>**NOTE** - When order = FALSE, the ascending argument is ignored. This is something that will be fixed in a later version. |
| .by | Alternative way of supplying groups using tidyselect notation. |
| .cols | (Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |
| as_qg | Should the group IDs be returned as a collapse "qG" class? The default (FALSE) always returns an integer vector. |
| .name | Name of the added ID column which should be a character vector of length 1. If .name = NULL (the default), add_group_id() will add a column named "group_id", and if one already exists, a unique name will be used. |

## Details

It's important to note for data frames, these functions by default assume no groups unless you supply them.

This means that when no groups are supplied:

- group_id(iris) returns a vector of ones
- row_id(iris) returns the plain row id numbers
- group_order(iris) == row_id(iris).

One can specify groups in the second argument like so:

- `group_id(iris, Species)`
- `row_id(iris, across(all_of("Species")))`
- `group_order(iris, across(where(is.numeric), desc))`

If you want `group_id` to always use all the columns of a data frame for grouping while simultaneously utilising the `group_id` methods, one can use the below function.

```
group_id2 <- function(data, ...){
 group_id(data, ..., .cols = names(data))
}
```

## Value

An integer vector.

## Examples

```
library(timeplyr)
library(dplyr)
library(ggplot2)

group_id(iris) # No groups
group_id(iris, Species) # Species groups
row_id(iris) # Plain row IDs
row_id(iris, Species) # Row IDs by group
# Order of Species + descending Petal.Width
group_order(iris, Species, desc(Petal.Width))
# Same as
order(iris$Species, -xtfrm(iris$Petal.Width))

# Tidy data-masking/tidyselect can be used
group_id(iris, across(where(is.numeric))) # Groups across numeric values
# Alternatively using tidyselect
group_id(iris, .by = where(is.numeric))

# Group IDs using a mixtured order
group_id(iris, desc(Species), Sepal.Length, desc(Petal.Width))

# add_ helpers
iris %>%
  distinct(Species) %>%
  add_group_id(Species)
iris %>%
  add_row_id(Species) %>%
  pull(row_id)

# Usage in data.table
library(data.table)
iris_dt <- as.data.table(iris)
iris_dt[, group_id := group_id(.SD, .cols = names(.SD)),
        .SDcols = "Species"]
```

```
# Or if you're using this often you can write a wrapper
set_add_group_id <- function(x, ..., .name = "group_id"){
  id <- group_id(x, ...)
  data.table::set(x, j = .name, value = id)
}
set_add_group_id(iris_dt, desc(Species))[]

mm_mpg <- mpg %>%
  select(manufacturer, model) %>%
  arrange(desc(pick(everything())))

# Sorted/non-sorted groups
mm_mpg %>%
  add_group_id(across(everything()),
               .name = "sorted_id", order = TRUE) %>%
  add_group_id(manufacturer, model,
               .name = "not_sorted_id", order = FALSE) %>%
  distinct()
```

---

growth *Rolling basic growth*

---

## Description

Calculate basic growth calculations on a rolling basis. `growth()` calculates the percent change between the totals of two numeric vectors when they're of equal length, otherwise the percent change between the means. `rolling_growth()` does the same calculation on 1 numeric vector, on a rolling basis. Pairs of windows of length `n`, lagged by the value specified by `lag` are compared in a similar manner. When `lag = n` then `data.table::frollsum()` is used, otherwise `data.table::frollmean()` is used.

## Usage

```
growth(x, y, na.rm = FALSE, log = FALSE, inf_fill = NULL)

rolling_growth(
  x,
  n = 1,
  lag = n,
  na.rm = FALSE,
  partial = TRUE,
  offset = NULL,
  weights = NULL,
  inf_fill = NULL,
  log = FALSE,
  ...
)
```

## Arguments

| | |
|---|---|
| x | Numeric vector. |
| y | numeric vector |
| na.rm | Should missing values be removed when calculating window? Defaults to FALSE. |
| log | If TRUE Growth (relative change) in total and mean events will be calculated on the log-scale. |
| inf_fill | Numeric value to replace Inf values with. Default behaviour is to keep Inf values. |
| n | Rolling window size, default is 1. |
| lag | Lag of basic growth comparison, default is the rolling window size. |
| partial | Should rates be calculated outwith the window using partial windows? If TRUE (the default), (n - 1) pairs of equally-sized rolling windows are compared, their size increasing by 1 up to size n, at which point the rest of the window pairs are all of size n. If FALSE all window-pairs will be of size n. |
| offset | Numeric vector of values to use as offset, e.g. population sizes or exposure times. |
| weights | Importance weights. These can either be length 1 or the same length as x. Currently, no normalisation of weights occurs. |
| ... | Further arguments to be passed on to frollmean. |

## Value

growth returns a numeric(1) and rolling_growth returns a numeric(length(x)).

## Examples

```
library(timeplyr)

set.seed(42)
# Growth rate is 6% per day
x <- 10 * (1.06)^(0:25)

# Simple growth from one day to the next
rolling_growth(x, n = 1)

# Growth comparing rolling 3 day cumulative
rolling_growth(x, n = 3)

# Growth comparing rolling 3 day cumulative, lagged by 1 day
rolling_growth(x, n = 3, lag = 1)

# Growth comparing windows of equal size
rolling_growth(x, n = 3, partial = FALSE)

# Seven day moving average growth
roll_mean(rolling_growth(x), window = 7, partial = FALSE)
```

growth_rate *Fast Growth Rates*

## Description

Calculate the rate of percentage change per unit time.

## Usage

```
growth_rate(x, na.rm = FALSE, log = FALSE, inf_fill = NULL)
```

## Arguments

| | |
|---|---|
| x | Numeric vector. |
| na.rm | Should missing values be removed when calculating window? Defaults to FALSE. When na.rm = TRUE the size of the rolling windows are adjusted to the number of non-NA values in each window. |
| log | If TRUE then growth rates are calculated on the log-scale. |
| inf_fill | Numeric value to replace Inf values with. Default behaviour is to keep Inf values. |

## Details

It is assumed that x is a vector of values with a corresponding time index that increases regularly with no gaps or missing values.

The output is to be interpreted as the average percent change per unit time.

For a rolling version that can calculate rates as you move through time, see roll_growth_rate.

For a more generalised method that incorporates time gaps and complex time windows, use time_roll_growth_rate.

The growth rate can also be calculated using the geometric mean of percent changes.

The below identity should always hold:

```
`tail(roll_growth_rate(x, window = length(x)), 1) == growth_rate(x)`
```

## Value

numeric(1)

## See Also

roll_growth_rate time_roll_growth_rate

## Examples

```
library(timeplyr)

set.seed(42)
initial_investment <- 100
years <- 1990:2000
# Assume a rate of 8% increase with noise
relative_increases <- 1.08 + rnorm(10, sd = 0.005)

assets <- Reduce(`*`, relative_increases, init = initial_investment, accumulate = TRUE)
assets

# Note that this is approximately 8%
growth_rate(assets)

# We can also calculate the growth rate via geometric mean

rel_diff <- exp(diff(log(assets)))
all.equal(rel_diff, relative_increases)

geometric_mean(rel_diff) == growth_rate(assets)

# Weighted growth rate

w <- c(rnorm(5)^2, rnorm(5)^4)
geometric_mean(rel_diff, weights = w)

# Rolling growth rate over the last n years
roll_growth_rate(assets)

# The same but using geometric means
exp(roll_mean(log(c(NA, rel_diff))))

# Rolling growth rate over the last 5 years
roll_growth_rate(assets, window = 5)
roll_growth_rate(assets, window = 5, partial = FALSE)

## Rolling growth rate with gaps in time

years2 <- c(1990, 1993, 1994, 1997, 1998, 2000)
assets2 <- assets[years %in% years2]

# Below does not incorporate time gaps into growth rate calculation
# But includes helpful warning
time_roll_growth_rate(assets2, window = 5, time = years2)
# Time step allows us to calculate correct rates across time gaps
time_roll_growth_rate(assets2, window = 5, time = years2, time_step = 1) # Time aware
```

---

gsum *Grouped statistical functions.*

---

## Description

These functions are wrappers around the collapse equivalents but always return a vector the same length and same order as x.
They all accept group IDs for grouped calculations.

## Usage

```
gsum(x, g = NULL, na.rm = TRUE, ...)

gmean(x, g = NULL, na.rm = TRUE, ...)

gmin(x, g = NULL, na.rm = TRUE, ...)

gmax(x, g = NULL, na.rm = TRUE, ...)

gsd(x, g = NULL, na.rm = TRUE, ...)

gvar(x, g = NULL, na.rm = TRUE, ...)

gmode(x, g = NULL, na.rm = TRUE, ...)

gmedian(x, g = NULL, na.rm = TRUE, ...)

gfirst(x, g = NULL, na.rm = TRUE, ...)

glast(x, g = NULL, na.rm = TRUE, ...)

gnobs(x, g = NULL, ...)
```

## Arguments

| | |
|---|---|
| x | An atomic vector. |
| g | Group IDs passed directly to collapse::GRP(). This can be a vector, list or data frame. |
| na.rm | Should NA values be removed? Default is TRUE. |
| ... | Additional parameters passed on to the collapse package equivalents, fsum(), fmean(), fmin(), fmax(), fsd(), fvar(), fmode(), fmedian(), ffirst(), flast() and fnobs() |

## Value

A vector the same length as x.

## Examples

```
library(timeplyr)
library(dplyr)
library(ggplot2)
```

```
# Dplyr
iris %>%
  mutate(mean = mean(Sepal.Length), .by = Species)
# Timeplyr
iris %>%
  mutate(mean = gmean(Sepal.Length, g = Species))

# One can utilise pick() to specify multiple groups
mpg %>%
  mutate(mean = gmean(displ, g = pick(manufacturer, model)))

# Alternatively you can create a unique ID for each group
mpg %>%
  add_group_id(manufacturer, model) %>%
  mutate(mean = gmean(displ, g = group_id))

# Another example

iris %>%
  add_group_id(Species, .name = "g") %>%
  mutate(min = gmin(Sepal.Length, g = g),
         max = gmax(Sepal.Length, g = g),
         sum = gsum(Sepal.Length, g = g),
         mean = gmean(Sepal.Length, g = g)) %>%
  # The below is equivalent to above
  mutate(min2 = min(Sepal.Length),
         max2 = max(Sepal.Length),
         sum2 = sum(Sepal.Length),
         mean2 = mean(Sepal.Length),
         .by = Species) %>%
  distinct(Species,
           min, min2,
           max, max2,
           sum, sum2,
           mean, mean2)
```

---

gunique *Grouped* `unique()`, `sort()` *and* `duplicated()`

---

## Description

These functions use `collapse` and are like the `collapse` counterpart but differ in that they accept a group g argument which allows for more flexible by-group sorting.

## Usage

```
gunique(x, g = NULL, sort = FALSE, order = TRUE, use.g.names = TRUE)
```

```
gduplicated(x, g = NULL, order = TRUE, all = FALSE)

gwhich_duplicated(x, g = NULL, order = TRUE, all = FALSE)

gsort(x, g = NULL, order = TRUE, use.g.names = TRUE)

gorder(x, g = NULL, order = TRUE)
```

## Arguments

| | |
|---|---|
| x | A vector or data frame. |
| g | Object used for grouping, passed directly to collapse::GRP().<br>This can for example be a vector or data frame. |
| sort | Should the result be sorted?<br>This only applies to gunique(). |
| order | Should the groups be treated as ordered groups? Default is TRUE. |
| use.g.names | Should group names be used? Default is TRUE. |
| all | If TRUE, gduplicated() returns all duplicated values, including the first occurrence. |

---

# iso_week *Efficient, simple and flexible ISO week calculation*

---

## Description

iso_week() is a flexible function to return formatted ISO weeks, with optional ISO year and ISO day. isoday() returns the day of the ISO week.

## Usage

```
iso_week(x, year = TRUE, day = FALSE)

isoday(x)
```

## Arguments

| | |
|---|---|
| x | Date vector. |
| year | Logical. If TRUE then ISO Year is returned along with the ISO week. |
| day | Logical. If TRUE then day of the week is returned with the ISO week, starting at 1, Monday, and ending at 7, Sunday. |

## Value

An ISO week vector of class character.

## Examples

```
library(timeplyr)
library(lubridate)

iso_week(today())
iso_week(today(), day = TRUE)
iso_week(today(), year = FALSE, day = TRUE)
iso_week(today(), year = FALSE, day = FALSE)
```

---

`is_date` *Utility functions for checking if date or datetime*

---

## Description

Utility functions for checking if date or datetime

## Usage

```
is_date(x)

is_datetime(x)

is_time(x)

is_time_or_num(x)
```

## Arguments

| | |
|---|---|
| `x` | Time variable.<br>Can be a `Date`, `POSIXt`, `numeric`, `integer`, `yearmon`, `yearqtr`, `year_month` or `year_quarter`. |

## Value

A logical of length 1.

`is_whole_number` *Are all numbers whole numbers?*

## Description

Are all numbers whole numbers?

## Usage

```
is_whole_number(x, tol = .Machine$double.eps, na.rm = TRUE)
```

## Arguments

| | |
|---|---|
| `x` | A numeric vector. |
| `tol` | tolerance value.<br>The default is `.Machine$double.eps`, essentially the lowest possible tolerance.<br>A more typical tolerance for double floating point comparisons in other comparisons is `sqrt(.Machine$double.eps)`. |
| `na.rm` | Should `NA` values be removed before calculation? Default is `TRUE`. |

## Details

This is a very efficient function that returns `FALSE` if any number is not a whole-number and `TRUE` if all of them are.

### Method:

`x` is defined as a whole number vector if all numbers satisfy `abs(x - round(x)) < tol`.

### `NA` handling:

`NA` values are handled in a custom way.
If `x` is an integer, `TRUE` is always returned even if `x` has missing values.
If `x` has both missing values and decimal numbers, `FALSE` is always returned.
If `x` has missing values, and only whole numbers and `na.rm = FALSE`, then `NA` is returned.
Basically `NA` is only returned if `na.rm = FALSE` and `x` is a double vector of only whole numbers and `NA` values.

Inspired by the discussion in this thread: check-if-the-number-is-integer

## Value

A logical vector of length 1.

## Examples

```
library(timeplyr)
library(dplyr)

# Has built-in tolerance
sqrt(2)^2 %% 1 == 0
is_whole_number(sqrt(2)^2)

is_whole_number(1)
is_whole_number(1.2)

x1 <- c(0.02, 0:10^5)
x2 <- c(0:10^5, 0.02)

is_whole_number(x1)
is_whole_number(x2)

# Somewhat more strict than all.equal

all.equal(10^9 + 0.0001, round(10^9 + 0.0001))
is_whole_number(10^9 + 0.0001)

# Can safely be used to select whole number variables
starwars %>%
  select(where(is_whole_number))

# To reduce the size of any data frame one can use the below code

df <- starwars %>%
  mutate(across(where(is_whole_number), as.integer))
```

---

`missing_dates` *Check for missing dates between first and last date*

---

## Description

Check for missing dates between first and last date

## Usage

```
missing_dates(x)

n_missing_dates(x)
```

## Arguments

| | |
|---|---|
| `x` | A date or datetime vector, or a data frame. |

### Value

A date vector if x is a vector, or a list if x is a `data.frame`.

---

`num_na` *Fast number of missing values*

---

### Description

A faster and more efficient alternative to `sum(is.na(x))`.
Long vectors, i.e vectors with length >= 2^31 are also supported.

### Usage

```
num_na(x)
```

### Arguments

| | |
|---|---|
| x | A vector. |

### Value

Number of `NA` values.

### Examples

```
library(timeplyr)
library(bench)

flights <- nycflights13::flights

# num_na is more efficient than using `sum(is.na())`
mark(vapply(flights, num_na, integer(1)),
     vapply(flights, function(x) sum(is.na(x)), integer(1)),
     iterations = 10)
```

---

`q_summarise` *Fast grouped quantile summary*

---

### Description

`collapse` and `data.table` are used for the calculations.

## Usage

```
q_summarise(
  data,
  ...,
  probs = seq(0, 1, 0.25),
  type = 7,
  pivot = c("wide", "long"),
  na.rm = TRUE,
  sort = TRUE,
  .by = NULL,
  .cols = NULL
)
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| ... | Variables used to calculate quantiles for. Tidy data-masking applies. |
| probs | Quantile probabilities. |
| type | An integer from 5-9 specifying which algorithm to use. See quantile for more details. |
| pivot | Should data be pivoted wide or long? Default is wide. |
| na.rm | Should NA values be removed? Default is TRUE. |
| sort | Should groups be sorted? Default is TRUE. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| .cols | (Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |

## Value

A data.table containing the quantile values for each group.

## See Also

stat_summarise

## Examples

```
library(timeplyr)
library(dplyr)

# Standard quantiles
iris %>%
  q_summarise(Sepal.Length)
# Quantiles by species
iris %>%
  q_summarise(Sepal.Length, .by = Species)
```

```
# Quantiles by species across multiple columns
iris %>%
  q_summarise(Sepal.Length, Sepal.Width,
              probs = c(0, 1),
              .by = Species)
# Long format if one desires, useful for ggplot2
iris %>%
  q_summarise(Sepal.Length, pivot = "long",
              .by = Species)
# Example with lots of groups
set.seed(20230606)
df <- data.frame(x = rnorm(10^5),
                 g = sample.int(10^5, replace = TRUE))
q_summarise(df, x, .by = g, sort = FALSE)
```

---

`roll_apply` *By-group rolling functions*

---

## Description

Apply any function on a rolling basis for each group using one-pass through the data.

## Usage

```
roll_apply(
  x,
  fun,
  before = Inf,
  after = 0L,
  g = NULL,
  partial = TRUE,
  default = NULL,
  unlist = FALSE
)
```

## Arguments

| | |
|---|---|
| `x` | Numeric vector, data frame, or list. |
| `fun` | A function. |
| `before` | A number denoting how many indices to look backward on a rolling basis. |
| `after` | A number denoting how many indices to look forward on a rolling basis. |
| `g` | Grouping object passed directly to `collapse::GRP()`. This can for example be a vector or data frame. |
| `partial` | Should calculations be done using partial windows? Default is `TRUE`. |
| `default` | Default value for each list element. |
| `unlist` | If `TRUE`, the result is passed to `unlist()`. The default is `FALSE`. |

### Details

roll_apply accepts any user function which makes it more flexible than the other rolling functions but much less efficient.
roll_apply2 is an alternative to roll_apply that instead accepts a vector of window sizes. The window sizes can be easily created using window_sequence().

### Value

A list the same length as x unless unlist is TRUE.

### See Also

time_roll_apply roll_sum roll_growth_rate

---

## roll_lag *Fast rolling grouped lags and differences*

---

### Description

Inspired by 'collapse', roll_lag and roll_diff operate similarly to flag and fdiff.

### Usage

```
roll_lag(x, n = 1L, g = NULL, fill = NULL)

roll_diff(x, n = 1L, g = NULL, fill = NULL)

lag_seq(size, n = 1L, partial = FALSE)

lag_(x, n = 1L, fill = NA)

lead_(x, n = 1L, fill = NA)

diff_(x, n = 1L, fill = NA)
```

### Arguments

| | |
|---|---|
| x | A vector. |
| n | Lag. Either length 1 or the same length as x. This can also be negative. |
| g | Grouping vector. This can be a vector, data frame or GRP object. |
| fill | Value to fill the first n elements. |
| size | Size of lag sequence. |
| partial | If TRUE, the sequence will increment from 0 up to the lag value. When calculating differences this can be useful, as passing this lag sequence to roll_diff will produce differences compared to the first value of x for the first n differences. |

## Details

While these may not be as fast the 'collapse' equivalents, they are adequately fast and efficient.
A key difference between `roll_lag` and `flag` is that `g` does not need to be sorted for the result to be correct.
Furthermore, a vector of lags can be supplied for a custom rolling lag. In this case, groups are ignored.
For time-based lags, see time_lag.

`lag_`, `lead_` and `diff_` are wrappers around the 'c++' functions that offer very low overhead of ~1 microsecond and thus are primarily for programmers.

## Value

A vector the same length as `x`.

## Examples

```
library(timeplyr)

x <- 1:10

roll_lag(x) # Lag
roll_lag(x, -1) # Lead
roll_diff(x) # Lag diff
roll_diff(x, -1) # Lead diff

# Using lag_seq()
roll_lag(x, lag_seq(length(x), 2))
roll_diff(x, lag_seq(length(x), 5, partial = TRUE))

# Like diff() but x/y instead of x-y
quotient <- function(x, n = 1L){
  x / roll_lag(x, n)
}
# People often call this a growth rate
# but it's just a percentage difference
# See ?roll_growth_rate for growth rate calculations
quotient(1:10)
```

---

# roll_na_fill *Fast grouped "locf" NA fill*

---

## Description

A fast and efficient by-group method for "last-observation-carried-forward" `NA` filling.

## Usage

```
roll_na_fill(x, g = NULL, fill_limit = Inf)

.roll_na_fill(x, fill_limit = Inf)
```

## Arguments

| | |
|---|---|
| x | A vector. |
| g | An object use for grouping x This may be a vector or data frame for example. |
| fill_limit | (Optional) maximum number of consecutive NAs to fill per NA cluster. Default is Inf. |

## Details

**Method:**

When supplying groups using g, this method uses `radixorder(g)` to specify how to loop through x, making this extremely efficient.

When x contains zero or all NA values, then x is returned with no copy made.

`.roll_na_fill()` is the same as `roll_na_fill()` but without a g argument and it performs no sanity checks. It is passed straight to c++ which makes it efficient for loops.

## Value

A filled vector of x the same length as x.

## Examples

```
library(timeplyr)
library(dplyr)
library(data.table)

words <- do.call(paste0,
                 do.call(expand.grid, rep(list(letters), 3)))
groups <- sample(words, size = 10^5, replace = TRUE)
x <- sample.int(10^2, 10^5, TRUE)
x[sample.int(10^5, 10^4)] <- NA

dt <- data.table(x, groups)

roll_na_fill(x, groups)

library(zoo)

  # Summary
# Latest version of vctrs with their vec_fill_missing
# Is the fastest but not most memory efficient
# For low repetitions and large vectors, data.table is best

# For large numbers of repetitions (groups) and data
# that is sorted by groups
```

```
# timeplyr is fastest

# No groups
bench::mark(e1 = dt[, filled1 := timeplyr::roll_na_fill(x)][]$filled1,
            e2 = dt[, filled2 := data.table::nafill(x, type = "locf")][]$filled2,
            e3 = dt[, filled3 := vctrs::vec_fill_missing(x)][]$filled3,
            e4 = dt[, filled4 := zoo::na.locf0(x)][]$filled4,
            e5 = dt[, filled5 := timeplyr::.roll_na_fill(x)][]$filled5)
# With group
bench::mark(e1 = dt[, filled1 := timeplyr::roll_na_fill(x, groups)][]$filled1,
         e2 = dt[, filled2 := data.table::nafill(x, type = "locf"), by = groups][]$filled2,
            e3 = dt[, filled3 := vctrs::vec_fill_missing(x), by = groups][]$filled3,
            e4 = dt[, filled4 := timeplyr::.roll_na_fill(x), by = groups][]$filled4)
# Data sorted by groups
setkey(dt, groups)
bench::mark(e1 = dt[, filled1 := timeplyr::roll_na_fill(x, groups)][]$filled1,
         e2 = dt[, filled2 := data.table::nafill(x, type = "locf"), by = groups][]$filled2,
            e3 = dt[, filled3 := vctrs::vec_fill_missing(x), by = groups][]$filled3,
            e4 = dt[, filled4 := timeplyr::.roll_na_fill(x), by = groups][]$filled4)
```

---

roll_sum *Fast by-group rolling functions*

---

## Description

An efficient method for rolling sum, mean and growth rate for many groups.

## Usage

```
roll_sum(
  x,
  window = Inf,
  g = NULL,
  partial = TRUE,
  weights = NULL,
  na.rm = TRUE,
  ...
)

roll_mean(
  x,
  window = Inf,
  g = NULL,
  partial = TRUE,
  weights = NULL,
  na.rm = TRUE,
  ...
```

```
)

roll_geometric_mean(
  x,
  window = Inf,
  g = NULL,
  partial = TRUE,
  weights = NULL,
  na.rm = TRUE,
  ...
)

roll_harmonic_mean(
  x,
  window = Inf,
  g = NULL,
  partial = TRUE,
  weights = NULL,
  na.rm = TRUE,
  ...
)

roll_growth_rate(
  x,
  window = Inf,
  g = NULL,
  partial = TRUE,
  na.rm = FALSE,
  log = FALSE,
  inf_fill = NULL
)
```

## Arguments

| | |
|---|---|
| x | Numeric vector, data frame, or list. |
| window | Rolling window size, default is Inf. |
| g | Grouping object passed directly to collapse::GRP(). This can for example be a vector or data frame. |
| partial | Should calculations be done using partial windows? Default is TRUE. |
| weights | Importance weights. Must be the same length as x. Currently, no normalisation of weights occurs. |
| na.rm | Should missing values be removed for the calculation? The default is TRUE. |
| ... | Additional arguments passed to data.table::frollmean and data.table::frollsum. |
| log | For roll_growth_rate: If TRUE then growth rates are calculated on the log-scale. |
| inf_fill | For roll_growth_rate: Numeric value to replace Inf values with. Default behaviour is to keep Inf values. |

## Details

`roll_sum` and `roll_mean` support parallel computations when `x` is a data frame of multiple columns. `roll_geometric_mean` and `roll_harmonic_mean` are convenience functions that utilise `roll_mean`. `roll_growth_rate` calculates the rate of percentage change per unit time on a rolling basis.

## Value

A numeric vector the same length as `x` when `x` is a vector, or a list when `x` is a `data.frame`.

## See Also

time_roll_mean

## Examples

```
library(timeplyr)

x <- 1:10
roll_sum(x) # Simple rolling total
roll_mean(x) # Simple moving average
roll_sum(x, window = 3)
roll_mean(x, window = 3)
roll_sum(x, window = 3, partial = FALSE)
roll_mean(x, window = 3, partial = FALSE)

# Plot of expected value of 'coin toss' over many flips
set.seed(42)
x <- sample(c(1, 0), 10^3, replace = TRUE)
ev <- roll_mean(x)
plot(ev)
abline(h = 0.5, lty = 2)

all.equal(roll_sum(iris$Sepal.Length, g = iris$Species),
          ave(iris$Sepal.Length, iris$Species, FUN = cumsum))
# The below is run using parallel computations where applicable
roll_sum(iris[, 1:4], window = 7, g = iris$Species)

  library(data.table)
  library(bench)
  df <- data.table(g = sample.int(10^4, 10^5, TRUE),
                   x = rnorm(10^5))
  mark(e1 = df[, mean := frollmean(x, n = 7,
                                   align = "right", na.rm = FALSE), by = "g"]$mean,
       e2 = df[, mean := roll_mean(x, window = 7, g = get("g"),
                                   partial = FALSE, na.rm = FALSE)]$mean)
```

sequence2 *Utilities for creating useful sequences*

## Description

sequence2 is an extension to sequence which accepts decimal number increments.
seq_id can be paired with sequence2 to group individual sequences.
seq_v is a vectorised version of seq.
window_sequence creates a vector of window sizes for rolling calculations.
lag_sequence creates a vector of lags for rolling calculations.
lead_sequence creates a vector of leads for rolling calculations.

## Usage

```
sequence2(size, from = 1L, by = 1L)

seq_id(size)

seq_v(from = 1L, to = 1L, by = 1L)

seq_size(from, to, by = 1L)

window_sequence(size, k, partial = TRUE, ascending = TRUE)

lag_sequence(size, k, partial = TRUE)

lead_sequence(size, k, partial = TRUE)
```

## Arguments

| | |
|---|---|
| size | Vector of sequence lengths. |
| from | Start of sequence(s). |
| by | Unit increment of sequence(s). |
| to | End of sequence(s). |
| k | Window/lag size. |
| partial | Should partial windows/lags be returned? Default is TRUE. |
| ascending | Should window sequence be ascending? Default is TRUE. |

## Details

sequence2() works in the same way as sequence() but can accept non-integer by values. It also recycles from and to, in the same way as sequence().
If any of the sequences contain values > .Machine$integer.max, then the result will always be a double vector.

from can be also be a date, date-time, or any object that supports addition and multiplication.

seq_v() is a vectorised version of seq() that strictly accepts only the arguments from, to and by.

## Value

A vector of length sum(size) except for seq_v which returns a vector of size sum((to - from) / (by + 1))

## Examples

```
library(timeplyr)

sequence(1:3)
sequence2(1:3)

sequence(1:3, by = 0.1)
sequence2(1:3, by = 0.1)

sequence(c(3, 2), by = c(-0.1, 0.1))
sequence2(c(3, 2), by = c(-0.1, 0.1))

# We can group sequences using seq_id
size <- c(7, 0, 3)
from <- 1
by <- c(-0.1, 0.1, 1/3)
x <- sequence2(size, from, by)
names(x) <- seq_id(size)
x

# Vectorised version of seq()
seq_v(1, 10, by = c(1, 0.5))
# Same as below
c(seq(1, 10, 1), seq(1, 10, 0.5))

# Programmers may use seq_size() to determine final sequence lengths

sizes <- seq_size(1, 10, by = c(1, 0.5))
print(paste(c("sequence sizes: (", sizes, ") total size:", sum(sizes)),
            collapse = " "))

# We can group sequences using seq_id

from <- Sys.Date()
to <- from + 10
by <- c(1, 2, 3)
x <- seq_v(from, to, by)
names(x) <- seq_id(seq_size(from, to, by))
x

# Utilities for rolling calculations
```

```
window_sequence(c(3, 5), 3)
window_sequence(c(3, 5), 3, partial = FALSE)
window_sequence(c(3, 5), 3, partial = TRUE, ascending = FALSE)
# One can for example use these in data.table::frollsum
```

---

`stat_summarise` *Fast grouped statistical summary for data frames.*

---

## Description

collapse and data.table are used for the calculations.

## Usage

```
stat_summarise(
  data,
  ...,
  stat = c("n", "nmiss", "ndistinct"),
  q_probs = NULL,
  na.rm = TRUE,
  sort = TRUE,
  .names = NULL,
  .by = NULL,
  .cols = NULL,
  as_tbl = FALSE
)

.stat_fns
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| ... | Variables to apply the statistical functions to. Tidy data-masking applies. |
| stat | A character vector of statistical summaries to apply. This can be one or more of the following:<br>"n", "nmiss", "ndistinct", "min", "max", "mean", "first", "last", "sd", "var", "mode", "median", "sum", "prop_complete". |
| q_probs | (Optional) Quantile probabilities. If supplied, q_summarise() is called and added to the result. |
| na.rm | Should NA values be removed? Default is TRUE. |
| sort | Should groups be sorted? Default is TRUE. |
| .names | An optional glue specification passed to stringr::glue(). If .names = NULL, then when there is 1 variable, the function name is used, i.e .names = "{.fn}", when there are multiple variables and 1 function, the variable names are used, i.e, .names = "{.col}" and in the case of multiple variables and functions. "{.col}_{.fn}" is used. |

| | |
|---|---|
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| .cols | (Optional) alternative to ... that accepts a named character vector or numeric vector. If speed is an expensive resource, it is recommended to use this. |
| as_tbl | Should the result be a tibble? Default is FALSE. |

## Format

.stat_fns

An object of class character of length 14.

## Details

stat_summarise() can apply multiple functions to multiple variables.

stat_summarise() is equivalent to
data %>% group_by(...) %>% summarise(across(..., list(...)))
but is faster and more efficient and accepts limited statistical functions.

## Value

A summary data.table containing the summary values for each group.

## See Also

q_summarise

## Examples

```
library(timeplyr)
library(dplyr)

stat_df <- iris %>%
  stat_summarise(Sepal.Length, .by = Species)
# Join quantile info too
q_df <- iris %>%
  q_summarise(Sepal.Length, .by = Species)
summary_df <- left_join(stat_df, q_df, by = "Species")
summary_df

# Multiple cols
iris %>%
  group_by(Species) %>%
  stat_summarise(across(contains("Width")),
            stat = c("min", "max", "mean", "sd"))
```

`time_aggregate` *Aggregate time to a higher unit*

## Description

Aggregate time to a higher unit for possibly many groups with respect to a time index.

## Usage

```
time_aggregate(
  x,
  time_by = NULL,
  g = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  direction = c("l2r", "r2l")
)
```

## Arguments

| | |
|---|---|
| `x` | Time vector.<br>Can be a `Date`, `POSIXt`, `numeric`, `integer`, `yearmon`, or `yearqtr` vector. |
| `time_by` | Time unit.<br>Must be one of the following:<br>• string, e.g `time_by = "day"` or `time_by = "2 weeks"`<br>• lubridate duration or period object, e.g. `days(1)` or `ddays(1)`.<br>• named list of length one, e.g. `list("days" = 7)`.<br>• Numeric vector, e.g. `time_by = 7`. |
| `g` | Grouping object passed directly to `collapse::GRP()`. This can for example be a vector or data frame. |
| `time_type` | If "auto", `periods` are used for the time expansion when days, weeks, months or years are specified, and `durations` are used otherwise. |
| `roll_month` | Control how impossible dates are handled when month or year arithmetic is involved. |
| `roll_dst` | See `?timechange::time_add` for the full list of details. |
| `direction` | Direction with which to aggregate time, "l2r" ("left-to-right") or "r2l" ("right-to-left"). If "l2r" (the default), then the minimum time is used as the reference time, otherwise the maximum time is used. |

## Details

`time_aggregate` aggregates time using distinct moving time range blocks of a specified time unit.

The actual calculation is extremely simple and essentially requires a subtraction, a rounding and an addition.

If for example `time_by = "week"` then all dates or datetimes will be shifted backwards (or forwards if direction is "r2l") to the nearest start of the week, where the start of week is based on `min(x)`. This is identical to building a weekly sequence and using this as breakpoints to cut `x`. No time expansion occurs so this is very efficient except when `periods` are used and there is a lot of data. In this case, provided the expansion is not too big, it may be more efficient to cut the data using the period sequence which can be achieved using `time_summarisev`.

## Value

A time aggregated vector the same class and length as `x`.

## See Also

time_summarisev

## Examples

```
library(timeplyr)
library(nycflights13)
library(lubridate)
library(dplyr)

sunique <- function(x) sort(unique(x))

hours <- sunique(flights$time_hour)
days <- as_date(hours)

# Aggregate by week or any time unit easily
unique(time_aggregate(hours, "week"))
unique(time_aggregate(hours, ddays(14)))
unique(time_aggregate(hours, "month"))
unique(time_aggregate(days, "month"))

# Left aligned
unique(time_aggregate(days, "quarter"))
# Right aligned
unique(time_aggregate(days, "quarter", direction = "r2l"))

# Very fast by group aggregation
week_by_tailnum <- time_aggregate(flights$time_hour, time_by = ddays(7),
                                  g = flights$tailnum)
# Confirm this has been done by group as each group will have a
# Different aggregate start date
flights %>%
  mutate(week_by_tailnum) %>%
  stat_summarise(week_by_tailnum, .by = tailnum, stat = "min",
```

```
                sort = FALSE)
```

---

time_by *Group by a time variable at a higher time unit*

---

## Description

time_by groups a time variable by a specified time unit like for example "days" or "weeks".
It can be used exactly like dplyr::group_by.

## Usage

```
time_by(
  data,
  time,
  time_by_unit = NULL,
  from = NULL,
  to = NULL,
  .name = "{.col}",
  .add = FALSE,
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  .time_by_group = TRUE
)

time_by_span(x)

time_by_var(x)

time_by_units(x)
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| time | Time variable (**data-masking**).<br>Can be a Date, POSIXt, numeric, integer, yearmon, or yearqtr. |
| time_by_unit | Time unit.<br>Must be one of the following:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• lubridate duration or period object, e.g. days(1) or ddays(1). |

- named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.
- Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`.

| | |
|---|---|
| `from` | (Optional) Start time. |
| `to` | (Optional) end time. |
| `.name` | An optional glue specification passed to `stringr::glue()` which can be used to concatenate strings to the time column name or replace it. |
| `.add` | Should the time groups be added to existing groups? Default is `FALSE`. |
| `time_type` | If "auto", `periods` are used for the time expansion when days, weeks, months or years are specified, and `durations` are used otherwise. If `durations` are used the output is always of class `POSIXt`. |
| `time_floor` | Should the start of each time sequence be floored to the nearest unit specified through the `time_by` argument? This is particularly useful for starting sequences at the beginning of a week or month for example. |
| `week_start` | day on which week starts following ISO conventions - 1 means Monday (default), 7 means Sunday. This is only used when `time_floor = TRUE`. |
| `roll_month` | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See `?timechange::time_add` for more details. |
| `roll_dst` | See `?timechange::time_add` for the full list of details. |
| `.time_by_group` | Should the time aggregations be built on a group-by-group basis (the default), or should the time variable be aggregated using the full data? If done by group, different groups may contain different time sequences. This only applies when `.add = TRUE`. |
| `x` | A `time_tbl_df`. |

## Value

A `time_tbl_df` which for practical purposes can be treated the same way as a dplyr `grouped_df`.

## Examples

```
library(dplyr)
library(timeplyr)
library(nycflights13)
library(lubridate)


# Basic usage
hourly_flights <- flights %>%
  time_by(time_hour) # Detects time granularity

hourly_flights
time_by_span(hourly_flights)
```

```
monthly_flights <- flights %>%
  time_by(time_hour, "month")
weekly_flights <- flights %>%
  time_by(time_hour, "week", time_floor = TRUE)

monthly_flights %>%
  count()

weekly_flights %>%
  summarise(n = n(), arr_delay = mean(arr_delay, na.rm = TRUE))

# To aggregate multiple variables, use time_aggregate or time_summarisev


flights %>%
  select(time_hour) %>%
  mutate(across(everything(), \(x) time_summarisev(x, time_by = dweeks(1)))) %>%
  count(time_hour)
```

---

`time_count` *Fast count time at higher time units.*

---

## Description

This function operates like `dplyr::count()` but with emphasis on a specified time variable.

## Usage

```
time_count(
  data,
  time = NULL,
  ...,
  time_by = NULL,
  from = NULL,
  to = NULL,
  complete = FALSE,
  wt = NULL,
  name = NULL,
  sort = FALSE,
  .by = NULL,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  include_interval = FALSE
)
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| time | Time variable. |
| ... | Additional variables to include. |
| time_by | Time unit.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. list("days" = 7). For the vectorized time functions, you can supply multiple values, e.g. list("days" = 1:10).<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g time_by = 1. |
| from | Time series start date. If NULL then min time is used. |
| to | Time series end date. If NULL then max time is used. |
| complete | **Deprecated**. Use time_complete() after time_count() to complete missing gaps in time (as well as optionally expand groups). |
| wt | Frequency weights. dplyr "data-masking" is used for variable selection. Can be NULL or a variable:<br>• If NULL (the default), counts the number of rows in each group.<br>• If a variable, computes sum(wt) for each group. |
| name | Character vector of length 1, specifying the name of the new column in the output. |
| sort | If TRUE the groups with largest counts will be sorted first. If FALSE the result is sorted by groups + time + ... groups. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| time_floor | Should from be floored to the nearest unit specified through the time_by argument? This is particularly useful for starting sequences at the beginning of a week or month for example. |
| week_start | day on which week starts following ISO conventions - 1 means Monday (default), 7 means Sunday. This is only used when time_floor = TRUE. |
| time_type | If "auto", periods are used for the time expansion when days, weeks, months or years are specified, and durations are used otherwise. |
| roll_month | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details. |
| roll_dst | See ?timechange::time_add for the full list of details. |
| include_interval | Logical. If TRUE then a column "interval" of the form time_min <= x < time_max is added showing the time interval in which the respective counts belong to. The rightmost interval will always be closed. |

## Details

`time_count` Creates an aggregated frequency time series where time can be aggregated to both lower and higher time units.

An important note is that when the data are grouped, time ranges are expanded on a group-by-group basis.
When groups are supplied through ..., the time range of the entire data is used to aggregate the time variable.

## Value

An object of class `data.frame` containing the aggregate time variable and corresponding counts.

## Examples

```
library(timeplyr)
library(dplyr)
library(lubridate)
library(nycflights13)

df <- flights %>%
  mutate(date = as_date(time_hour)) %>%
  select(year, month, day, origin, dest, date, time_hour)

# By default time_count() guesses the time granularity
df %>%
  time_count(time_hour)
# Aggregated to week level
df %>%
  time_count(time = date, time_by = "2 weeks")
df %>%
  time_count(time = date, time_by = list("months" = 3),
             from = dmy("15-01-2013"),
             time_floor = TRUE,
             include_interval = TRUE)
```

---

# time_cut *Cut dates and datetimes into regularly spaced date or datetime intervals*

---

## Description

`time_cut()` is very useful for plotting with dates and datetimes and always returns breaks of regular width.

## Usage

```
time_cut(
  x,
  n = 5,
  time_by = NULL,
  from = NULL,
  to = NULL,
  fmt = NULL,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  n_at_most = TRUE,
  as_factor = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_breaks(
  x,
  n = 5,
  time_by = NULL,
  from = NULL,
  to = NULL,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  n_at_most = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)
```

## Arguments

| | |
|---|---|
| x | Time variable.<br>Can be a Date, POSIXt, numeric, integer, yearmon, or yearqtr. |
| n | Number of breaks. |
| time_by | Time unit.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. list("days" = 7). For the vectorized time functions, you can supply multiple values, e.g. list("days" = 1:10).<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g time_by = 1. |
| from | Time series start date. |
| to | Time series end date. |

| | |
|---|---|
| fmt | (Optional) Date/datetime format for the factor labels. If supplied, this is passed to format(). |
| time_floor | Logical. Should the initial date/datetime be floored before building the sequence? |
| week_start | day on which week starts following ISO conventions - 1 means Monday (default), 7 means Sunday. This is only used when time_floor = TRUE. |
| n_at_most | **Deprecated**. No longer used. |
| as_factor | Logical. If TRUE the output is an ordered factor. Setting this to FALSE is sometimes much faster. |
| time_type | If "auto", periods are used for the time expansion when days, weeks, months or years are specified, and durations are used otherwise. |
| roll_month | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details. |
| roll_dst | See ?timechange::time_add for the full list of details. |

## Details

To specify exact widths, similar to ggplot2::cut_width(), supply time_by and n = Inf.
time_breaks() is a helper that returns only the time breaks.

By default time_cut() will try to find the prettiest way of cutting the interval by trying to cut the date/date-times into groups of the highest possible time units, starting at years and ending at milliseconds.

When x is a numeric vector, time_cut will behave similar to time_cut except for 3 things:

- The intervals are all right open and equal width, except for the rightmost interval which is closed with width <= the other widths.
- The left value of the leftmost interval is always min(x).
- Up to n breaks are created, i.e <= n breaks. This is to prioritise pretty breaks.

time_cut is a generalisation of time_summarisev such that the below identity should always hold:

```
identical(time_cut(x, n = Inf, as_factor = FALSE), time_summarisev(x))
```

## Value

time_breaks returns a vector of breaks.
time_cut returns either a factor or a vector the same class as x. In both cases it is the same length as x.

## Examples

```
library(timeplyr)
library(lubridate)
library(ggplot2)
library(dplyr)
```

```
time_cut(1:10, n = 5)
# Easily create custom time breaks
df <- nycflights13::flights %>%
  fslice_sample(n = 10, seed = 8192821) %>%
  select(time_hour) %>%
  farrange(time_hour) %>%
  mutate(date = as_date(time_hour))

# time_cut() and time_breaks() automatically find a
# suitable way to cut the data
time_cut(df$date)
# Works with datetimes as well
time_cut(df$time_hour, n = 5) # <= 5 breaks
# Custom formatting
time_cut(df$date, fmt = "%Y %b", time_by = "month")
# Just the breaks
time_breaks(df$date, n = 5, time_by = "month")

cut_dates <- time_cut(df$date)
date_breaks <- time_breaks(df$date)

# Grouping each interval into the start of its interval
identical(date_breaks[group_id(cut_dates)],
          time_cut(df$date, as_factor = FALSE))

# WHen n = Inf and as_factor = FALSE, it should be equivalent to using
# time_aggregate or time_summarisev
identical(time_cut(df$date, n = Inf, time_by = "month", as_factor = FALSE),
          time_summarisev(df$date, time_by = "month"))
identical(time_summarisev(df$date, time_by = "month"),
          time_aggregate(df$date, time_by = "month"))

# To get exact breaks at regular intervals, use time_expandv
weekly_breaks <- time_expandv(df$date,
                              time_by = "5 weeks",
                              week_start = 1, # Monday
                              time_floor = TRUE)
weekly_labels <- format(weekly_breaks, "%b-%d")
df %>%
  time_count(time = date, time_by = "week") %>%
  ggplot(aes(x = date, y = n)) +
  geom_bar(stat = "identity") +
  scale_x_date(breaks = weekly_breaks,
               labels = weekly_labels)
```

---

`time_diff` *Time differences by any time unit*

---

## Description

The time difference between 2 date or date-time vectors.

## Usage

```
time_diff(
  x,
  y,
  time_by = 1L,
  time_type = getOption("timeplyr.time_type", "auto")
)
```

## Arguments

| | |
|---|---|
| x | Start date or datetime. |
| y | End date or datetime. |
| time_by | Must be one of the three (Default is 1):<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. list("days" = 7). For the vectorized time functions, you can supply multiple values, e.g. list("days" = 1:10).<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g time_by = 1. |
| time_type | Time difference type: "auto", "duration" or "period". |

## Details

When time_by is a numeric vector, e.g time_by = 1 then base arithmetic using base::`-` is used, otherwise 'lubridate' style durations and periods are used.
Some more exotic time units such as quarters, fortnights, etcetera can be specified. See .time_units for more choices.

## Value

A numeric vector recycled to the length of max(length(x), length(y)).

## Examples

```
library(timeplyr)
library(lubridate)

time_diff(today(), today() + days(10),
          time_by = "days")
time_diff(today(), today() + days((0:3) * 7),
          time_by = weeks(1))
time_diff(today(), today() + days(100),
          time_by = list("days" = 1:100))
time_diff(1, 1 + 0:100, time_by = 3)

library(nycflights13)
library(bench)
```

```
# Period differences are much faster
mark(timeplyr = time_diff(flights$time_hour, today(), "weeks", time_type = "period"),
     lubridate = interval(flights$time_hour, today()) / weeks(1))
```

---

`time_diff_gcd` *Fast greatest common divisor of time differences*

---

## Description

Fast greatest common divisor of time differences

## Usage

```
time_diff_gcd(
  x,
  time_by = 1,
  time_type = getOption("timeplyr.time_type", "auto"),
  tol = sqrt(.Machine$double.eps)
)
```

## Arguments

| | |
|---|---|
| `x` | Time variable.<br>Can be a `Date`, `POSIXt`, `numeric`, `integer`, `yearmon`, or `yearqtr`. |
| `time_by` | Time unit (default is 1).<br>Must be one of the following:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"`<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`. |
| `time_type` | If "auto", `periods` are used if `x` is a Date and durations are used if `x` is a datetime. Otherwise numeric differences are calculated. |
| `tol` | Tolerance of comparison. The time differences are rounded using `digits = ceiling(abs(log10(tol)))` to try and avoid precision issues. |

## Value

A double vector of length 1 or length 0 if `length(x)` is 0.

## Examples

```
library(timeplyr)
library(lubridate)
library(cppdoubles)

time_diff_gcd(1:10)
time_diff_gcd(seq(0, 1, 0.2))

time_diff_gcd(time_seq(today(), today() + 100, time_by = "3 days"))
time_diff_gcd(time_seq(now(), len = 10^2, time_by = "125 seconds"))

# Monthly gcd using lubridate periods
quarter_seq <- time_seq(today(), len = 24, time_by = months(4))
time_diff_gcd(quarter_seq, time_by = months(1))
time_diff_gcd(quarter_seq, time_by = "months", time_type = "duration")

# Detects monthly granularity
double_equal(time_diff_gcd(as.vector(time(AirPassengers))), 1/12)
```

---

# time_distinct *A time based extension to* dplyr::distinct().

---

## Description

This works much the same as dplyr::distinct(), except that you can supply an additional time argument to allow for aggregating time to a higher unit.

## Usage

```
time_distinct(
  data,
  time = NULL,
  ...,
  time_by = NULL,
  from = NULL,
  to = NULL,
  .keep_all = FALSE,
  time_type = getOption("timeplyr.time_type", "auto"),
  include_interval = FALSE,
  .by = NULL,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  sort = FALSE
)
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| time | Time variable. |
| ... | Additional variables to include. |
| time_by | Time unit.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. list("days" = 7). For the vectorized time functions, you can supply multiple values, e.g. list("days" = 1:10).<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g time_by = 1. |
| from | Time series start date. |
| to | Time series end date. |
| .keep_all | If TRUE then all columns of data frame are kept, default is FALSE. |
| time_type | If "auto", periods are used for the time expansion when days, weeks, months or years are specified, and durations are used otherwise. |
| include_interval | Logical. If TRUE then a column "interval" of the form time_min <= x < time_max is added showing the time interval in which the respective counts belong to. The rightmost interval will always be closed. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| time_floor | Should from be floored to the nearest unit specified through the time_by argument? This is particularly useful for starting sequences at the beginning of a week or month for example. |
| week_start | day on which week starts following ISO conventions - 1 means Monday, 7 means Sunday (default). This is only used when time_floor = TRUE. |
| roll_month | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details. |
| roll_dst | See ?timechange::time_add for the full list of details. |
| sort | Should the result be sorted? Default is TRUE. If FALSE then original (input) order is kept. |

## Value

A data.frame of distinct aggregate time values across groups.

`time_elapsed` *Fast grouped time elapsed*

## Description

Calculate how much time has passed on a rolling or cumulative basis.

## Usage

```
time_elapsed(
  x,
  time_by = NULL,
  g = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  rolling = TRUE,
  fill = NA,
  na_skip = TRUE
)
```

## Arguments

| | |
|---|---|
| `x` | Time variable.<br>Can be a `Date`, `POSIXt`, `numeric`, `integer`, `yearmon`, or `yearqtr`. |
| `time_by` | Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"`<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`. |
| `g` | Object to be used for grouping `x`, passed onto `collapse::GRP()`. |
| `time_type` | Time type, either "auto", "duration" or "period". With larger data, it is recommended to use `time_type = "duration"` for speed and efficiency. |
| `rolling` | If `TRUE` (the default) then lagged time differences are calculated on a rolling basis, essentially like `diff()`.<br>If `FALSE` then time differences compared to the index (first) time are calculated. |
| `fill` | When `rolling = TRUE`, this is the value that fills the first elapsed time. The default is `NA`. |
| `na_skip` | Should `NA` values be skipped? Default is `TRUE`. |

## Details

time_elapsed() is quite efficient when there are many groups, especially if your data is sorted in order of those groups. In the case that g is supplied, it is most efficient when your data is sorted by g . When na_skip is TRUE and rolling is also TRUE, NA values are simply skipped and hence the time differences between the current value and the previous non-NA value are calculated. For example, c(3, 4, 6, NA, NA, 9) becomes c(NA, 1, 2, NA, NA, 3).
When na_skip is TRUE and rolling is FALSE, time differences between the current value and the first non-NA value of the series are calculated. For example, c(NA, NA, 3, 4, 6, NA, 8) becomes c(NA, NA, 0, 1, 3, NA, 5).

## Value

A numeric vector the same length as x.

## Examples

```
library(timeplyr)
library(dplyr)
library(lubridate)

x <- time_seq(today(), length.out = 25, time_by = "3 days")
time_elapsed(x)
time_elapsed(x, rolling = FALSE, time_by = "day")

# Grouped example
set.seed(99)
# ~ 100k groups, 1m rows
x <- sample(time_seq_v2(20, today(), "day"), 10^6, TRUE)
g <- sample.int(10^5, 10^6, TRUE)

time_elapsed(x, time_by = "day", g = g)
```

---

# time_episodes *Episodic calculation of time-since-event data*

---

## Description

This function assigns episodes to events based on a pre-defined threshold of a chosen time unit.

## Usage

```
time_episodes(
  data,
  time,
  time_by = NULL,
  window = 1,
  roll_episode = TRUE,
```

```
  switch_on_boundary = TRUE,
  fill = 0,
  .add = FALSE,
  event = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  .by = NULL
)
```

## Arguments

| | |
|---|---|
| data | A data frame. |
| time | Date or datetime variable to use for the episode calculation. Supply the variable using tidyselect notation. |
| time_by | Time units used to calculate episode flags. If time_by is NULL then a heuristic will try and estimate the highest order time unit associated with the time variable. If specified, then by must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. list("days" = 7). For the vectorized time functions, you can supply multiple values, e.g. list("days" = 1:10).<br>• Numeric vector. If by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g time_by = 1. |
| window | Single number defining the episode threshold. When rolling = TRUE events with a t_elapsed >= window since the last event are defined as a new episode.<br>When rolling = FALSE events with a t_elapsed >= window since the first event of the corresponding episode are defined as a new episode.<br>By default, window = 1 which assigns every event to a new episode. |
| roll_episode | Logical. Should episodes be calculated using a rolling or fixed window? If TRUE (the default), an amount of time must have passed (>= window) since the last event, with each new event effectively resetting the time at which you start counting.<br>If FALSE, the elapsed time is fixed and new episodes are defined based on how much cumulative time has passed since the first event of each episode. |
| switch_on_boundary | When an exact amount of time (specified in time_by) has passed, should there be an increment in ID?<br>The default is TRUE.<br>For example, if time_by = "days" and switch_on_boundary = FALSE, > 1 day must have passed, otherwise >= 1 day must have passed. |
| fill | Value to fill first time elapsed value. Only applicable when roll_episode = TRUE.<br>Default is 0. |
| .add | Should episodic variables be added to the data?<br>If FALSE (the default), then only the relevant variables are returned.<br>If TRUE, the episodic variables are added to the original data using dplyr::bind_cols().<br>In both cases, the order of the data is unchanged. |

| | |
|---|---|
| `event` | (**Optional**) List that encodes which rows are events, and which aren't. By default `time_episodes()` assumes every observation (row) is an event but this need not be the case.<br>`event` must be a named list of length 1 where the values of the list element represent the event. For example, if your events were coded as `0` and `1` in a variable named "evt" where `1` represents the event, you would supply `event = list(evt = 1)`. |
| `time_type` | Time type, either "auto", "duration" or "period". With larger data, it is recommended to use `time_type = "duration"` for speed and efficiency. |
| `.by` | (Optional). A selection of columns to group by for this operation. Columns are specified using `tidyselect`. |

## Details

`time_episodes()` calculates the time elapsed (rolling or fixed) between successive events, and flags these events as episodes or not based on how much time has passed.

An example of episodic analysis can include disease infections over time.

In this example, a positive test result represents an **event** and
a new infection represents a new **episode**.

It is assumed that after a pre-determined amount of time, a positive result represents a new episode of infection.

To perform simple time-since-event analysis, set `window` to `1`, which is the default.

The data are always sorted before calculation and then sorted back to the input order.

4 Key variables will be calculated:

- **ep_id** - An integer variable signifying which episode each event belongs to.
  Non-events are assigned `NA`.
  `ep_id` is an increasing integer starting at 1. In the infections scenario, 1 are positives within the first episode of infection, 2 are positives within the second episode of infection and so on.
- **ep_id_new** - An integer variable signifying the first instance of each new episode. This is an increasing integer where 0 signifies within-episode observations and >= 1 signifies the first instance of the respective episode.
- **t_elapsed** - The time elapsed since the last event.
  When `roll_episode = FALSE`, this becomes the time elapsed since the first event of the current episode. Time units are specified in the by argument.
- **ep_start** - Start date/datetime of the episode.

`data.table` and `collapse` are used for speed and efficiency.

## Value

A `data.frame` in the same order as it was given.

## See Also

time_elapsed time_seq_id

## Examples

```
library(timeplyr)
library(dplyr)
library(nycflights13)
library(lubridate)
library(ggplot2)

# Say we want to flag origin-destination pairs
# that haven't seen departures or arrivals for a week

events <- flights %>%
  mutate(date = as_date(time_hour)) %>%
  group_by(origin, dest) %>%
  time_episodes(date, time_by = "week", window = 1)
episodes <- events %>%
  filter(ep_id_new > 1)
nrow(fdistinct(episodes, origin, dest)) # 55 origin-destinations

# As expected summer months saw the least number of
# dry-periods
episodes %>%
  ungroup() %>%
  time_count(time = ep_start, time_by = "week", time_floor = TRUE) %>%
  ggplot(aes(x = ep_start, y = n)) +
  geom_bar(stat = "identity")
```

---

# time_expand *A time based extension to* `tidyr::complete()`.

---

## Description

A time based extension to `tidyr::complete()`.

## Usage

```
time_expand(
  data,
  time = NULL,
  ...,
  .by = NULL,
  time_by = NULL,
  from = NULL,
  to = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  expand_type = c("nesting", "crossing"),
  sort = TRUE,
```

```
  keep_class = TRUE,
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  log_limit = 8
)

time_complete(
  data,
  time = NULL,
  ...,
  .by = NULL,
  time_by = NULL,
  from = NULL,
  to = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  expand_type = c("nesting", "crossing"),
  sort = TRUE,
  keep_class = TRUE,
  fill = NA,
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  log_limit = 8
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame. |
| `time` | Time variable. |
| `...` | Groups to expand. |
| `.by` | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| `time_by` | Time unit.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"`<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`. |
| `from` | Time series start date. |
| `to` | Time series end date. |
| `time_type` | If "auto", `periods` are used for the time expansion when days, weeks, months or years are specified, and `durations` are used otherwise. |

| | |
|---|---|
| time_floor | Should from be floored to the nearest unit specified through the time_by argument? This is particularly useful for starting sequences at the beginning of a week or month for example. |
| week_start | day on which week starts following ISO conventions - 1 means Monday (default), 7 means Sunday. This is only used when floor_date = TRUE. |
| expand_type | Type of time expansion to use where "nesting" finds combinations already present in the data, "crossing" finds all combinations of values in the group variables. |
| sort | Logical. If TRUE expanded/completed variables are sorted. |
| keep_class | Logical. If TRUE then the class of the input data is retained. If FALSE, which is sometimes faster, a data.table is returned. |
| roll_month | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details. |
| roll_dst | See ?timechange::time_add for the full list of details. |
| log_limit | The maximum log10 number of rows that can be expanded. Anything exceeding this will throw an error. |
| fill | A named list containing value-name pairs to fill the named implicit missing values. |

## Details

This works much the same as tidyr::complete(), except that you can supply an additional time argument to allow for filling in time gaps, expansion of time, as well as aggregating time to a higher unit. lubridate is used for handling time, while data.table and collapse are used for the data frame expansion.

At the moment, within group combinations are ignored. This means when expand_type = nesting, existing combinations of supplied groups across the entire dataset are used, and when expand_type = crossing, all possible combinations of supplied groups across the **entire** dataset are used as well.

## Value

A data.frame of expanded time by or across groups.

## Examples

```
library(timeplyr)
library(dplyr)
library(lubridate)
library(nycflights13)

x <- flights$time_hour

time_num_gaps(x) # Missing hours

flights_count <- flights %>%
  fcount(time_hour)
```

```
# Fill in missing hours
flights_count %>%
  time_complete(time = time_hour)

# You can specify units too
flights_count %>%
  time_complete(time = time_hour, time_by = "hours")
flights_count %>%
  time_complete(time = as_date(time_hour), time_by = "days") #  Nothing to complete here

# Where time_expand() and time_complete() really shine is how fast they are with groups
flights %>%
  group_by(origin, dest) %>%
  time_expand(time = time_hour, time_by = dweeks(1))
```

---

time_expandv *Vector date and datetime functions*

---

## Description

These are atomic vector-based functions of the tidy equivalents which all have a "v" suffix to denote this. These are more geared towards programmers and allow for working with date and datetime vectors.

## Usage

```
time_expandv(
  x,
  time_by = NULL,
  from = NULL,
  to = NULL,
  g = NULL,
  use.g.names = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_span(
  x,
  time_by = NULL,
  from = NULL,
  to = NULL,
  g = NULL,
  use.g.names = TRUE,
```

```
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_completev(
  x,
  time_by = NULL,
  from = NULL,
  to = NULL,
  sort = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_summarisev(
  x,
  time_by = NULL,
  from = NULL,
  to = NULL,
  sort = FALSE,
  unique = FALSE,
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  include_interval = FALSE
)

time_countv(
  x,
  time_by = NULL,
  from = NULL,
  to = NULL,
  sort = TRUE,
  unique = TRUE,
  complete = FALSE,
  time_type = getOption("timeplyr.time_type", "auto"),
  include_interval = FALSE,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
```

```
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_span_size(
  x,
  time_by = NULL,
  from = NULL,
  to = NULL,
  g = NULL,
  use.g.names = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1)
)
```

## Arguments

| | |
|---|---|
| x | Time variable.<br>Can be a Date, POSIXt, numeric, integer, yearmon, or yearqtr. |
| time_by | Time unit.<br>Must be one of the following:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. list("days" = 7). For the vectorized time functions, you can supply multiple values, e.g. list("days" = 1:10).<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g time_by = 1. |
| from | Time series start date. |
| to | Time series end date. |
| g | Grouping object passed directly to collapse::GRP(). This can for example be a vector or data frame. |
| use.g.names | Should the result include group names? Default is TRUE. |
| time_type | If "auto", periods are used for the time expansion when days, weeks, months or years are specified, and durations are used otherwise. |
| time_floor | Should from be floored to the nearest unit specified through the time_by argument? This is particularly useful for starting sequences at the beginning of a week or month for example. |
| week_start | day on which week starts following ISO conventions - 1 means Monday (default), 7 means Sunday. This is only used when time_floor = TRUE. |
| roll_month | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details. |
| roll_dst | See ?timechange::time_add for the full list of details. |
| sort | Should the output be sorted? Default is TRUE. |

unique — Should the result be unique or match the length of the vector? Default is TRUE.

include_interval — Logical. If TRUE then the result is a tibble with a column "interval" of the form time_min <= x < time_max showing the time interval in which the aggregated time points belong to. The rightmost interval will always be closed.

complete — Logical. If TRUE implicit gaps in time are filled before counting and after time aggregation (controlled using time_by). The default is FALSE.

## Value

Vectors (typically the same class as x) of varying lengths depending on the arguments supplied. time_countv() returns a tibble.

## Examples

```
library(timeplyr)
library(dplyr)
library(lubridate)
library(nycflights13)

x <- unique(flights$time_hour)

# Number of missing hours
time_num_gaps(x)

# Same as above
time_span_size(x) - length(unique(x))

# Time sequence that spans the data
time_span(x) # Automatically detects hour granularity
time_span(x, time_by = "month")
time_span(x, time_by = list("quarters" = 1),
             to = today(),
             # Floor start of sequence to nearest month
             time_floor = TRUE)

# Complete missing gaps in time using time_completev
y <- time_completev(x, time_by = "hour")
identical(y[!y %in% x], time_gaps(x))

# Summarise time using time_summarisev
time_summarisev(y, time_by = "quarter")
time_summarisev(y, time_by = "quarter", unique = TRUE)
flights %>%
  fcount(quarter_start = time_summarisev(time_hour, "quarter"))
# Alternatively
time_countv(x, time_by = "quarter")
# If you want the above as an atomic vector just use tibble::deframe
```

time_gaps *Gaps in a regular time sequence*

## Description

`time_gaps()` checks for missing gaps in time for any regular date or datetime sequence.

## Usage

```
time_gaps(
  x,
  time_by = NULL,
  g = NULL,
  use.g.names = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  check_time_regular = FALSE
)

time_num_gaps(
  x,
  time_by = NULL,
  g = NULL,
  use.g.names = TRUE,
  na.rm = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  check_time_regular = FALSE
)

time_has_gaps(
  x,
  time_by = NULL,
  g = NULL,
  use.g.names = TRUE,
  na.rm = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  check_time_regular = FALSE
)
```

## Arguments

| | |
|---|---|
| x | A date, datetime or numeric vector. |
| time_by | Time unit.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"` |

- named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.
- Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`.

`g` Grouping object passed directly to `collapse::GRP()`. This can for example be a vector or data frame.

`use.g.names` Should the result include group names? Default is `TRUE`.

`time_type` Time type, either "auto", "duration" or "period". With larger data, it is recommended to use `time_type = "duration"` for speed and efficiency.

`check_time_regular` Should the time vector be checked to see if it is regular (with or without gaps)? Default is `FALSE`.

`na.rm` Should `NA` values be removed? Default is `TRUE`.

## Details

When `check_time_regular` is TRUE, `x` is passed to `time_is_regular`, which checks that the time elapsed between successive values are in increasing order and are whole numbers. For more strict checks, see `?time_is_regular`.

## Value

`time_gaps` returns a vector of time gaps.
`time_num_gaps` returns the number of time gaps.
`time_has_gaps` returns a logical(1) of whether there are gaps.

## Examples

```
library(timeplyr)
library(dplyr)
library(lubridate)
library(nycflights13)

missing_dates(flights$time_hour)
time_has_gaps(flights$time_hour)
time_num_gaps(flights$time_hour)
time_gaps(flights$time_hour)
time_num_gaps(flights$time_hour, g = flights$origin)

# Number of missing hours by origin and dest
flights %>%
  group_by(origin, dest) %>%
  summarise(n_missing = time_num_gaps(time_hour, "hours"))
```

`time_ggplot` *Quick time-series ggplot*

## Description

`time_ggplot()` is a neat way to quickly plot aggregate time-series data.

## Usage

```
time_ggplot(
  data,
  time,
  value,
  group = NULL,
  facet = FALSE,
  geom = ggplot2::geom_line,
  ...
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame |
| `time` | Time variable using `tidyselect`. |
| `value` | Value variable using `tidyselect`. |
| `group` | (Optional) Group variable using `tidyselect`. |
| `facet` | When groups are supplied, should multi-series be plotted separately or on the same plot? Default is `FALSE`, or together. |
| `geom` | `ggplot2` 'geom' type. Default is `geom_line()`. |
| `...` | Further arguments passed to the chosen 'geom'. |

## Value

A `ggplot`.

## See Also

ts_as_tibble

## Examples

```
library(dplyr)
library(timeplyr)
library(ggplot2)

# It's as easy as this
AirPassengers %>%
```

```
  ts_as_tibble() %>%
  time_ggplot(time, value)

# And this
EuStockMarkets %>%
  ts_as_tibble() %>%
  time_ggplot(time, value, group)

# zoo example
x.Date <- as.Date("2003-02-01") + c(1, 3, 7, 9, 14) - 1
x <- zoo::zoo(rnorm(5), x.Date)
x %>%
  ts_as_tibble() %>%
  time_ggplot(time, value)

# An example using raw data

ebola <- outbreaks::ebola_sim$linelist

# We can build a helper to count and complete
# Using the same time grid

count_and_complete <- function(.data, time, ...,
                               time_by = NULL, time_floor = TRUE){
  .data %>%
    time_count(!!dplyr::enquo(time), ..., time_by = time_by,
               time_floor = time_floor) %>%
    time_complete(!!dplyr::enquo(time), ..., time_by = time_by,
                  time_floor = time_floor, fill = list(n = 0))
}
ebola %>%
  count_and_complete(date_of_onset, outcome, time_by = "week") %>%
  time_ggplot(date_of_onset, n, geom = geom_blank) +
  geom_col(aes(fill = outcome))
```

---

`time_id` *Time ID*

---

## Description

Generate a time ID that signifies how many time steps away a time value is from the starting time point or more intuitively, this is the time passed since the first time point.

## Usage

```
time_id(
  x,
  time_by = NULL,
  g = NULL,
```

```
  na_skip = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  shift = 1L
)
```

## Arguments

| | |
|---|---|
| x | Time variable.<br>Can be a Date, POSIXt, numeric, integer, yearmon, or yearqtr. |
| time_by | Time unit.<br>This signifies the granularity of the time data with which to measure gaps in the sequence. If your data is daily for example, supply time_by = "days". If weekly, supply time_by = "week". Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. list("days" = 7). For the vectorized time functions, you can supply multiple values, e.g. list("days" = 1:10).<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g time_by = 1. |
| g | Object used for grouping x. This can for example be a vector or data frame. g is passed directly to collapse::GRP(). |
| na_skip | Should NA values be skipped? Default is TRUE. |
| time_type | If "auto", periods are used for the time expansion when days, weeks, months or years are specified, and durations are used otherwise. |
| shift | Value used to shift the time IDs. Typically this is 1 to ensure the IDs start at 1 but can be 0 or even negative if for example your time values are going backwards in time. |

## Details

This is heavily inspired by collapse::timeid but differs in 3 ways:

- The time steps need not be the greatest common divisor of successive differences
- The starting time point may not necessarily be the earliest chronologically and thus time_id can generate negative IDs.
- g can be supplied to calculate IDs by group.

time_id(c(3, 2, 1)) is not the same as collapse::timeid(c(3, 2, 1)). In general time_id(sort(x)) should be equal to collapse::timeid(sort(x)). The time difference GCD is always calculated using all the data and not by-group.

## Value

An integer vector the same length as x.

## See Also

time_elapsed time_seq_id

---

`time_is_regular` *Is time a regular sequence? (Experimental)*

---

## Description

This function is a fast way to check if a time vector is a regular sequence, possibly for many groups. Regular in this context means that the lagged time differences are a whole multiple of the specified time unit.
This means `x` can be a regular sequence with or without gaps in time.

## Usage

```
time_is_regular(
  x,
  time_by = NULL,
  g = NULL,
  use.g.names = TRUE,
  na.rm = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  allow_gaps = TRUE,
  allow_dups = TRUE
)
```

## Arguments

| | |
|---|---|
| `x` | A vector. Can be a `Date`, `POSIXt`, `numeric`, `integer`, `yearmon`, or `yearqtr`. |
| `time_by` | Time unit.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"`<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`. |
| `g` | Grouping object passed directly to `collapse::GRP()`. This can for example be a vector or data frame.<br>Note that when g is supplied the output is a logical with length matching the number of unique groups. |
| `use.g.names` | Should the result include group names? Default is `TRUE`. |
| `na.rm` | Should `NA` values be removed before calculation? Default is `TRUE`. |

| | |
|---|---|
| time_type | If "auto", periods are used for the time expansion when days, weeks, months or years are specified, and durations are used otherwise. If durations are used the output is always of class POSIXt. |
| allow_gaps | Should gaps be allowed? Default is TRUE. |
| allow_dups | Should duplicates be allowed? Default is TRUE. |

## Value

A logical vector the same length as the number of supplied groups.

## Examples

```
library(timeplyr)
library(lubridate)
library(dplyr)

x <- 1:5
y <- c(1, 1, 2, 3, 5)

time_is_regular(x)
time_is_regular(y)

increment <- 1

# No duplicates allowed
time_is_regular(x, increment, allow_dups = FALSE)
time_is_regular(y, increment, allow_dups = FALSE)

# No gaps allowed
time_is_regular(x, increment, allow_gaps = FALSE)
time_is_regular(y, increment, allow_gaps = FALSE)

# Grouped
eu_stock <- ts_as_tibble(EuStockMarkets)
eu_stock <- eu_stock %>%
  mutate(date = as_date(
    date_decimal(time)
  ))

time_is_regular(eu_stock$date, g = eu_stock$group,
                time_by = 1)
# This makes sense as no trading occurs on weekends and holidays
time_is_regular(eu_stock$date, g = eu_stock$group,
                time_by = 1,
                allow_gaps = FALSE)
```

time_lag *Time-lagged values*

## Description

Time-lagged values

## Usage

```
time_lag(
  x,
  k = 1L,
  time = seq_along(x),
  g = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)
```

## Arguments

| | |
|---|---|
| x | Vector. |
| k | Lag size, must be one of the following:<br>• string, e.g "day" or "2 weeks"<br>• lubridate duration or period object, e.g. days(1) or ddays(1).<br>• named list of length one, e.g. list("days" = 7).<br>• Numeric vector, e.g. 7. |
| time | (Optional) time index.<br>Can be a Date, POSIXt, numeric, integer, yearmon, or yearqtr vector. |
| g | Grouping object passed directly to collapse::GRP(). This can for example be a vector or data frame. |
| time_type | If "auto", periods are used for the time expansion when lubridate periods are specified or when days, weeks, months or years are specified, and durations are used otherwise. |
| roll_month | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details. |
| roll_dst | See ?timechange::time_add for the full list of details. |

## Value

A vector of length(x) lagged by a specified time unit.

## Examples

```
library(timeplyr)

x <- 1:10
t <- time_seq(Sys.Date(), len = 10, time_by = "3 days")

dplyr::lag(x)
time_lag(x)
time_lag(x, time = t, k = "3 days")

# No values exist at t-1 days
time_lag(x, time = t, k = 1)
```

---

`time_mutate` *A time based extension to* `dplyr::mutate()`.

---

## Description

This works much the same as `dplyr::mutate()`, except that you can supply an additional `time` argument to allow for aggregating time to a higher unit.

## Usage

```
time_mutate(
  data,
  time = NULL,
  ...,
  time_by = NULL,
  from = NULL,
  to = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  include_interval = FALSE,
  .by = NULL,
  .keep = c("all", "used", "unused", "none"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame. |
| `time` | Time variable. |
| `...` | Additional variables to include. |

| | |
|---|---|
| time_by | Time unit.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks"<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. list("days" = 7). For the vectorized time functions, you can supply multiple values, e.g. list("days" = 1:10).<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g time_by = 1. |
| from | Time series start date. |
| to | Time series end date. |
| time_type | If "auto", periods are used for the time expansion when days, weeks, months or years are specified, and durations are used otherwise. |
| include_interval | Logical. If TRUE then a column "interval" of the form time_min <= x < time_max is added showing the time interval in which the respective counts belong to. The rightmost interval will always be closed. |
| .by | (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select. |
| .keep | Control which columns are retained. See ?dplyr::mutate for more details. |
| time_floor | Should from be floored to the nearest unit specified through the time_by argument? This is particularly useful for starting sequences at the beginning of a week or month for example. |
| week_start | day on which week starts following ISO conventions - 1 means Monday (default), 7 means Sunday. This is only used when floor_date = TRUE. |
| roll_month | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details. |
| roll_dst | See ?timechange::time_add for the full list of details. |

## Value

A data.frame with added columns.

## Examples

```
library(timeplyr)
library(dplyr)
library(lubridate)
library(nycflights13)

# Like the other time_ functions, it allows for an additional time variable to
# aggregate by
flights %>%
  fdistinct(time_hour) %>%
  time_mutate(time = across(time_hour, as_date),
```

```
              time_by = "month", .keep = "none",
              include_interval = TRUE) %>%
    fdistinct()
```

---

`time_roll_diff` *Lagged time differences*

---

## Description

`time_roll_diff` is like `diff()` but always returns a `numeric(length(x))`.

## Usage

```
time_roll_diff(
  time,
  time_by = 1,
  lag = 1L,
  g = NULL,
  time_type = getOption("timeplyr.time_type", "auto")
)
```

## Arguments

| | |
|---|---|
| `time` | Time variable.<br>Can be a `Date`, `POSIXt`, `numeric`, `integer`, `yearmon`, or `yearqtr`. |
| `time_by` | Time unit.<br>Must be one of the following:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"`<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`. |
| `lag` | A number indicating the lag size. Negative values are allowed. |
| `g` | Grouping object passed directly to `collapse::GRP()`. This can for example be a vector or data frame. |
| `time_type` | If "auto", `periods` are used for the time expansion when days, weeks, months or years are specified, and `durations` are used otherwise. |

## Details

`time_elapsed` is very similar to `time_roll_diff` but is more general in that it supports cumulative time differencing, `NA` filling as well as `NA` skipping.

## Value

A numeric vector the same length as `x`. on the arguments supplied.

## See Also

time_elapsed

---

`time_roll_sum` *Fast time-based by-group rolling sum/mean - Currently experimental*

---

## Description

`time_roll_sum` and `time_roll_mean` are efficient methods for calculating a rolling sum and mean respectively given many groups and with respect to a date or datetime time index.
It is always aligned "right".
`time_roll_window` splits `x` into windows based on the index.
`time_roll_window_size` returns the window sizes for all indices of `x`.
`time_roll_apply` is a generic function that applies any function on a rolling basis with respect to a time index.

`time_roll_growth_rate` can efficiently calculate by-group rolling growth rates with respect to a date/datetime index.

## Usage

```
time_roll_sum(
  x,
  window = Inf,
  time = seq_along(x),
  weights = NULL,
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE,
  na.rm = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  ...
)

time_roll_mean(
  x,
  window = Inf,
  time = seq_along(x),
  weights = NULL,
  g = NULL,
```

```
  partial = TRUE,
  close_left_boundary = FALSE,
  na.rm = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  ...
)

time_roll_growth_rate(
  x,
  window = Inf,
  time = seq_along(x),
  time_step = NULL,
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE,
  na.rm = TRUE,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_roll_window_size(
  time,
  window = Inf,
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_roll_window(
  x,
  window = Inf,
  time = seq_along(x),
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_roll_apply(
  x,
```

```
  window = Inf,
  fun,
  time = seq_along(x),
  g = NULL,
  partial = TRUE,
  unlist = FALSE,
  close_left_boundary = FALSE,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)
```

## Arguments

| | |
|---|---|
| x | Numeric vector. |
| window | Time window size (Default is Inf). Must be one of the following:<br>• string, e.g window = "day" or window = "2 weeks"<br>• lubridate duration or period object, e.g. days(1) or ddays(1).<br>• named list of length one, e.g. list("days" = 7).<br>• Numeric vector, e.g. window = 7. |
| time | (Optional) time index.<br>Can be a Date, POSIXt, numeric, integer, yearmon, or yearqtr vector. |
| weights | Importance weights. Must be the same length as x. Currently, no normalisation of weights occurs. |
| g | Grouping object passed directly to collapse::GRP(). This can for example be a vector or data frame. |
| partial | Should calculations be done using partial windows? Default is TRUE. |
| close_left_boundary | Should the left boundary be closed? For example, if you specify window = "day" and time = c(today(), today() + 1),<br>a value of FALSE would result in the window vector c(1, 1) whereas a value of TRUE would result in the window vector c(1, 2). |
| na.rm | Should missing values be removed for the calculation? The default is TRUE. |
| time_type | If "auto", periods are used for the time expansion when lubridate periods are specified or when days, weeks, months or years are specified, and durations are used otherwise. |
| roll_month | Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details. |
| roll_dst | See ?timechange::time_add for the full list of details. |
| ... | Additional arguments passed to data.table::frollmean and data.table::frollsum. |
| time_step | An optional but **important** argument that follows the same input rules as window.<br>It is currently only used only in time_roll_growth_rate.<br>If this is supplied, the time differences across gaps in time are incorporated into the growth rate calculation. See **details** for more info. |

| | |
|---|---|
| fun | A function. |
| unlist | Should the output of time_roll_apply be unlisted with unlist? Default is FALSE. |

## Details

It is much faster if your data are already sorted such that !is.unsorted(order(g, x)) is TRUE.

**Growth rates:**

For growth rates across time, one can use time_step to incorporate gaps in time into the calculation.

For example:
x <- c(10, 20)
t <- c(1, 10)
k <- Inf
time_roll_growth_rate(x, time = t, window = k) = c(1, 2) whereas
time_roll_growth_rate(x, time = t, window = k, time_step = 1) = c(1, 1.08)
The first is a doubling from 10 to 20, whereas the second implies a growth of 8% for each time step from 1 to 10.
This allows us for example to calculate daily growth rates over the last x months, even with missing days.

## Value

A vector the same length as time.

## Examples

```
library(timeplyr)
library(lubridate)
library(dplyr)

time <- time_seq(today(), today() + weeks(3),
                 time_by = "3 days")
set.seed(99)
x <- sample.int(length(time))

roll_mean(x, window = 7)
roll_sum(x, window = 7)

time_roll_mean(x, window = ddays(7), time = time)
time_roll_sum(x, window = days(7), time = time)

# Alternatively and more verbosely
x_chunks <- time_roll_window(x, window = 7, time = time)
x_chunks
vapply(x_chunks, mean, 0)

# Interval (x - 3 x]
time_roll_sum(x, window = ddays(3), time = time)
```

```
# An example with an irregular time series

t <- today() + days(sort(sample(1:30, 20, TRUE)))
time_elapsed(t, days(1)) # See the irregular elapsed time
x <- rpois(length(t), 10)

tibble(x, t) %>%
  mutate(sum = time_roll_sum(x, time = t, window = days(3))) %>%
  time_ggplot(t, sum)


### Rolling mean example with many time series

# Sparse time with duplicates
index <- sort(sample(seq(now(), now() + dyears(3), by = "333 hours"),
                     250, TRUE))
x <- matrix(rnorm(length(index) * 10^3),
            ncol = 10^3, nrow = length(index),
            byrow = FALSE)

zoo_ts <- zoo::zoo(x, order.by = index)

# Normally you might attempt something like this
apply(x, 2,
      function(x){
        time_roll_mean(x, window = dmonths(1), time = index)
      }
)
# Unfortunately this is too slow and inefficient


# Instead we can pivot it longer and code each series as a separate group
tbl <- ts_as_tibble(zoo_ts)

tbl %>%
  mutate(monthly_mean = time_roll_mean(value, window = dmonths(1),
                                       time = time, g = group))
```

---

`time_seq` *Time based version of* `base::seq()`

---

## Description

Time based version of `base::seq()`

## Usage

```
time_seq(
```

```
  from,
  to,
  time_by,
  length.out = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  week_start = getOption("lubridate.week.start", 1),
  time_floor = FALSE,
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)

time_seq_sizes(
  from,
  to,
  time_by,
  time_type = getOption("timeplyr.time_type", "auto")
)

time_seq_v(
  from,
  to,
  time_by,
  time_type = getOption("timeplyr.time_type", "auto"),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1)
)

time_seq_v2(
  sizes,
  from,
  time_by,
  time_type = getOption("timeplyr.time_type", "auto"),
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary")
)
```

## Arguments

| | |
|---|---|
| from | Start date/datetime of sequence. |
| to | End date/datetime of sequence. |
| time_by | Time unit increment.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g time_by = "days" or time_by = "2 weeks" |

- named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.
- Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`.

`length.out` Length of the sequence.

`time_type` If "auto", `periods` are used for the time expansion when days, weeks, months or years are specified, and `durations` are used otherwise. If `durations` are used the output is always of class `POSIXt`.

`week_start` day on which week starts following ISO conventions - 1 means Monday (default), 7 means Sunday. This is only used when `time_floor = TRUE`.

`time_floor` Should `from` be floored to the nearest unit specified through the `time_by` argument? This is particularly useful for starting sequences at the beginning of a week or month for example.

`roll_month` Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See `?timechange::time_add` for more details.

`roll_dst` See `?timechange::time_add` for the full list of details.

`sizes` Time sequence sizes.

## Details

This works like `seq()`, but using `timechange` for the period calculations and `base::seq.POSIXT()` for the duration calculations. In many ways it is improved over `seq` as dates and/or datetimes can be supplied with no errors to the start and end points. Examples like,
`time_seq(now(), length.out = 10, by = "0.5 days", seq_type = "dur")` and
`time_seq(today(), length.out = 10, by = "0.5 days", seq_type = "dur")`
produce more expected results compared to
`seq(now(), length.out = 10, by = "0.5 days")` or
`seq(today(), length.out = 10, by = "0.5 days")`.

For a vectorized implementation with multiple start/end times, use `time_seq_v()`/`time_seq_v2()`

`time_seq_sizes()` is a convenience function to calculate time sequence lengths, given start/end times.

## Value

`time_seq` returns a time sequence.
`time_seq_sizes` returns an integer vector of sequence sizes.
`time_seq_v` returns time sequences.
`time_seq_v2` also returns time sequences.

## See Also

seq_id time_seq_id

## Examples

```
library(timeplyr)
library(lubridate)

# Dates
today <- today()
now <- now()

time_seq(today, today + years(1), time_by = "day")
time_seq(today, length.out = 10, time_by = "day")
time_seq(today, length.out = 10, time_by = "hour")

time_seq(today, today + years(1), time_by = list("days" = 1)) # Alternative
time_seq(today, today + years(1), time_by = "week")
time_seq(today, today + years(1), time_by = "fortnight")
time_seq(today, today + years(1), time_by = "year")
time_seq(today, today + years(10), time_by = "year")
time_seq(today, today + years(100), time_by = "decade")

# Datetimes
time_seq(now, now + years(1), time_by = "12 hours")
time_seq(now, now + years(1), time_by = "day")
time_seq(now, now + years(1), time_by = "week")
time_seq(now, now + years(1), time_by = "fortnight")
time_seq(now, now + years(1), time_by = "year")
time_seq(now, now + years(10), time_by = "year")
time_seq(now, today + years(100), time_by = "decade")

# You can seamlessly mix dates and datetimes with no errors.
time_seq(now, today + days(3), time_by = "day")
time_seq(now, today + days(3), time_by = "hour")
time_seq(today, now + days(3), time_by = "day")
time_seq(today, now + days(3), time_by = "hour")

# Choose between durations or periods

start <- dmy(31012020)
# If time_type is left as is,
# periods are used for days, weeks, months and years.
time_seq(start, time_by = "month", length.out = 12,
         time_type = "period")
time_seq(start, time_by = "month", length.out = 12,
         time_type = "duration")
# Notice how strange base R version is.
seq(start, by = "month", length.out = 12)

# Roll forward or backward impossible dates

leap <- dmy(29022020) # Leap day
end <- dmy(01032021)
# 3 different options
time_seq(leap, to = end, time_by = "year",
```

```
         roll_month = "NA")
time_seq(leap, to = end, time_by = "year",
         roll_month = "postday")
time_seq(leap, to = end, time_by = "year",
         roll_month = getOption("timeplyr.roll_month", "preday"))
```

---

`time_seq_id` *Generate a unique identifier for a regular time sequence with gaps*

---

## Description

A unique identifier is created every time a specified amount of time has passed, or in the case of regular sequences, when there is a gap in time.

## Usage

```
time_seq_id(
  x,
  time_by = NULL,
  threshold = 1,
  g = NULL,
  na_skip = TRUE,
  rolling = TRUE,
  switch_on_boundary = FALSE,
  time_type = getOption("timeplyr.time_type", "auto")
)
```

## Arguments

| | |
|---|---|
| `x` | Date, datetime or numeric vector. |
| `time_by` | Time unit.<br>This signifies the granularity of the time data with which to measure gaps in the sequence. If your data is daily for example, supply `time_by = "days"`. If weekly, supply `time_by = "week"`. Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"`<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`. |
| `threshold` | Threshold such that when the time elapsed exceeds this, the sequence ID is incremented by 1. For example, if `time_by = "days"` and `threshold = 2`, then when 2 days have passed, a new ID is created. Furthermore, `threshold` generally need not be supplied as |

time_by = "3 days" & threshold = 1
is identical to
time_by = "days" & threshold = 3.

g — Object used for grouping x. This can for example be a vector or data frame. g is passed directly to collapse::GRP().

na_skip — Should NA values be skipped? Default is TRUE.

rolling — When this is FALSE, a new ID is created every time a cumulative amount of time has passed. Once that amount of time has passed, a new ID is created, the clock "resets" and we start counting from that point.

switch_on_boundary — When an exact amount of time (specified in time_by) has passed, should there an increment in ID? The default is FALSE. For example, if time_by = "days" and switch_on_boundary = FALSE, > 1 day must have passed, otherwise >= 1 day must have passed.

time_type — If "auto", periods are used for the time expansion when days, weeks, months or years are specified, and durations are used otherwise.

## Details

time_seq_id() Assumes x is regular and in ascending or descending order. To check this condition formally, use time_is_regular().

## Value

An integer vector of length(x).

## Examples

```
library(dplyr)
library(timeplyr)
library(lubridate)

# Weekly sequence, with 2 gaps in between
x <- time_seq(today(), length.out = 10, time_by = "week")
x <- x[-c(3, 7)]
# A new ID when more than a week has passed since the last time point
time_seq_id(x, time_by = "week")
# A new ID when >= 2 weeks has passed since the last time point
time_seq_id(x, time_by = "weeks", threshold = 2, switch_on_boundary = TRUE)
# A new ID when at least 4 cumulative weeks have passed
time_seq_id(x, time_by = "4 weeks",
            switch_on_boundary = TRUE, rolling = FALSE)
# A new ID when more than 4 cumulative weeks have passed
time_seq_id(x, time_by = "4 weeks",
            switch_on_boundary = FALSE, rolling = FALSE)
```

`time_summarise` *A time based extension to* `dplyr::summarise()`

## Description

This works much the same as `dplyr::summarise()`, except that you can supply an additional `time` argument to allow for aggregating time to a higher unit.

## Usage

```
time_summarise(
  data,
  time = NULL,
  ...,
  time_by = NULL,
  from = NULL,
  to = NULL,
  time_type = getOption("timeplyr.time_type", "auto"),
  include_interval = FALSE,
  .by = NULL,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1),
  roll_month = getOption("timeplyr.roll_month", "preday"),
  roll_dst = getOption("timeplyr.roll_dst", "boundary"),
  sort = TRUE
)
```

## Arguments

| | |
|---|---|
| `data` | A data frame. |
| `time` | Time variable. |
| `...` | Additional variables to include. |
| `time_by` | Time unit.<br>Must be one of the three:<br>• string, specifying either the unit or the number and unit, e.g `time_by = "days"` or `time_by = "2 weeks"`<br>• named list of length one, the unit being the name, and the number the value of the list, e.g. `list("days" = 7)`. For the vectorized time functions, you can supply multiple values, e.g. `list("days" = 1:10)`.<br>• Numeric vector. If time_by is a numeric vector and x is not a date/datetime, then arithmetic is used, e.g `time_by = 1`. |
| `from` | Time series start date. |
| `to` | Time series end date. |
| `time_type` | If "auto", `periods` are used for the time expansion when days, weeks, months or years are specified, and `durations` are used otherwise. |

include_interval
: Logical. If TRUE then a column "interval" of the form time_min <= x < time_max is added showing the time interval in which the respective counts belong to. The rightmost interval will always be closed.

.by
: (Optional). A selection of columns to group by for this operation. Columns are specified using tidy-select.

time_floor
: Should from be floored to the nearest unit specified through the time_by argument? This is particularly useful for starting sequences at the beginning of a week or month for example.

week_start
: day on which week starts following ISO conventions - 1 means Monday, 7 means Sunday (default). This is only used when time_floor = TRUE.

roll_month
: Control how impossible dates are handled when month or year arithmetic is involved. Options are "preday", "boundary", "postday", "full" and "NA". See ?timechange::time_add for more details.

roll_dst
: See ?timechange::time_add for the full list of details.

sort
: Should the result be sorted? Default is TRUE. If FALSE then original (input) order is kept. The sorting only applies to groups and time variable.

## Value

A summarised data.frame.

## Examples

```
library(timeplyr)
library(dplyr)
library(lubridate)
library(nycflights13)

# Works the same way as summarise()
# Monthly average arrival time
flights %>%
  mutate(date = as_date(time_hour)) %>%
  time_summarise(mean_arr_time = mean(arr_time, na.rm = TRUE),
                 time = date,
                 time_by = "month",
                 include_interval = TRUE)
# Example of monthly summary using zoo's yearmon

flights %>%
  mutate(yearmon = zoo::as.yearmon(as_date(time_hour))) %>%
  time_summarise(time = yearmon,
                 n = n(),
                 mean_arr_time = mean(arr_time, na.rm = TRUE),
                 include_interval = TRUE)
```

top_n_tbl *Top N most/least frequent values*

## Description

Inspired by `forcats::fct_lump_n` and by the lack of a good alternative.
These are very fast and memory efficient.

## Usage

```
top_n_tbl(x, n = 5, na_rm = FALSE, with_ties = FALSE)

top_n(x, n = 5, na_rm = FALSE, with_ties = FALSE)

bottom_n_tbl(x, n = 5, na_rm = FALSE, with_ties = FALSE)

bottom_n(x, n = 5, na_rm = FALSE, with_ties = FALSE)

lump_top_n(
  x,
  n = 5,
  na_rm = FALSE,
  with_ties = FALSE,
  as_factor = TRUE,
  other_category = "Other"
)

lump_bottom_n(
  x,
  n = 5,
  na_rm = FALSE,
  with_ties = FALSE,
  as_factor = TRUE,
  other_category = "Other"
)
```

## Arguments

| | |
|---|---|
| `x` | vector |
| `n` | integer Number of categories to include. |
| `na_rm` | logical Should `NA` values be removed? Default is `FALSE`. |
| `with_ties` | logical Should ties be kept? Default is `FALSE`. |
| `as_factor` | logical Should the result be a factor? Default is `TRUE`. |
| `other_category` | character Name of the other category. Default is "Other". |

### Details

top_n returns a vector of the most frequent values, with an added attribute of counts named "n".
top_n_tbl returns a data frame of top n values and associated counts.
lump_top_n returns a factor such that any values not in the top n values are placed into a separate category "Other".

### Value

top_n/bottom_n return a vector the same class as x.
top_n_tbl/bottom_n_tbl return a 2-col data.frame.
lump_top_n/lump_bottom_n return a factor (or character vector).

### Examples

```
library(dplyr)
library(timeplyr)

### Top 3 hair colours
timeplyr::top_n(starwars$hair_color, n = 3)

starwars %>%
  count(hair_col = lump_top_n(hair_color, n = 3))

top_n_tbl(starwars$hair_color, n = 3)
```

---

## ts_as_tibble *Turn* ts *into a* tibble

---

### Description

While a method already exists in the tibble package, this method works differently in 2 ways:

- The time variable associated with the time-series is also returned.
- The returned tibble is always in long format, even when the time-series is multivariate.

### Usage

```
ts_as_tibble(x, name = "time", value = "value", group = "group")

## Default S3 method:
ts_as_tibble(x, name = "time", value = "value", group = "group")

## S3 method for class 'mts'
ts_as_tibble(x, name = "time", value = "value", group = "group")

## S3 method for class 'xts'
ts_as_tibble(x, name = "time", value = "value", group = "group")
```

```
## S3 method for class 'zoo'
ts_as_tibble(x, name = "time", value = "value", group = "group")

## S3 method for class 'timeSeries'
ts_as_tibble(x, name = "time", value = "value", group = "group")
```

## Arguments

| | |
|---|---|
| x | An object of class ts, mts, zoo, xts or timeSeries. |
| name | Name of the output time column. |
| value | Name of the output value column. |
| group | Name of the output group column when there are multiple series. |

## Value

A 2-column tibble containing the time index and values for each time index. In the case where there are multiple series, this becomes a 3-column tibble with an additional "group" column added.

## See Also

time_ggplot

## Examples

```
library(timeplyr)
library(ggplot2)
library(dplyr)

# Using the examples from ?ts

# Univariate
uts <- ts(cumsum(1 + round(rnorm(100), 2)),
          start = c(1954, 7), frequency = 12)
uts_tbl <- ts_as_tibble(uts)

## Multivariate
mts <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 1), frequency = 12)
mts_tbl <- ts_as_tibble(mts)

uts_tbl %>%
  time_ggplot(time, value)

mts_tbl %>%
  time_ggplot(time, value, group, facet = TRUE)

# zoo example
x.Date <- as.Date("2003-02-01") + c(1, 3, 7, 9, 14) - 1
x <- zoo::zoo(rnorm(5), x.Date)
ts_as_tibble(x)
x <- zoo::zoo(matrix(1:12, 4, 3), as.Date("2003-01-01") + 0:3)
```

```
ts_as_tibble(x)
```

---

`unit_guess` *Guess time unit and extract basic information.*

---

## Description

This is a simple R function to convert time units to a common unit, with number and scale. See `.time_units` for a list of accepted time units.

## Usage

```
unit_guess(x)
```

## Arguments

| | |
|---|---|
| `x` | This can be 1 of 4 options:<br>• A string, e.g. "7 days"<br>• lubridate duration or period object, e.g. `days(1)` or `ddays(1)`.<br>• A list, e.g. list("days" = 7)<br>• A number, e.g. 5 |

## Value

A list of length 3, including the unit, number and scale.

## Examples

```
library(timeplyr)

# Single units
unit_guess("days")
unit_guess("hours")

# Multi-units
unit_guess("7 days")
unit_guess("0.5 hours")

# Negative units
unit_guess("-7 days")
unit_guess("-.12 days")

# Exotic units
unit_guess("fortnights")
unit_guess("decades")
.extra_time_units

# list input is accepted
```

```
unit_guess(list("months" = 12))
# With a list, a vector of numbers is accepted
unit_guess(list("months" = 1:10))
unit_guess(list("days" = -10:10 %% 7))

# Numbers also accepted
unit_guess(100)
```

---

`year_month` *Fast methods for creating year-months and year-quarters*

---

## Description

These are experimental methods for working with year-months and year-quarters inspired by 'zoo' and 'tsibble'.

## Usage

```
year_month(x)

year_quarter(x)

YM(length = 0L)

YQ(length = 0L)
```

## Arguments

| | |
|---|---|
| `x` | A `year_month`, `year_quarter`, or any other time-based object. |
| `length` | Length of `year_month` or `year_quarter`. |

## Details

The biggest difference is that the underlying data is simply the number of months/quarters since epoch. This makes integer arithmetic very simple, and allows for fast sequence creation as well as fast coercion to `year_month` and `year_quarter` from numeric vectors.

Printing method is also fast.

## Examples

```
library(timeplyr)
library(lubridate)

x <- year_month(today())

# Adding 1 adds 1 month
x + 1
```

```
# Adding 12 adds 1 year
x + 12
# Sequence of yearmonths
x + 0:12

# If you unclass, do the same arithmetic, and coerce back to year_month
# The result is always the same
year_month(unclass(x) + 1)
year_month(unclass(x) + 12)

# Initialise a year_month or year_quarter to the specified length
YM(0)
YQ(0)
YM(3)
YQ(3)
```

# Index