Package 'timeplyr'

May 8, 2026

Title Fast Tidy Tools for Date and Date-Time Manipulation

Version 1.1.2

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 (>= 4.1.0)

Imports cheapr (>= 1.3.2), cli, collapse (>= 2.0.0), cppdoubles (>= 0.2.0), data.table (>= 1.14.8), dplyr (>= 1.1.0), fastplyr (>= 0.9.9), ggplot2 (>= 3.4.0), lifecycle, lubridate (>= 1.9.0), pillar (>= 1.7.0), rlang (>= 1.0.0), scales, stringr (>= 1.4.0), timechange (>= 0.2.0), vctrs (>= 0.6.0)

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

LinkingTo cpp11, tzdb

Config/testthat/edition 3

Encoding UTF-8

RoxygenNote 7.3.3

NeedsCompilation yes

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

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

Repository CRAN

Date/Publication 2026-02-10 07:50:02 UTC

Contents

`.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

[Deprecated]

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.

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"
)
```

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.

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, "day")
calendar(my_seq)
```

`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,
  timespan = 1L,
  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.
`timespan`	timespan.
`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 = "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 = "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 = "weeks",
                  bw = "nrd")
inc_distr_weeks$plot
```

---

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 <- function(x, na.rm = TRUE, weights = NULL){
  exp(collapse::fmean(log(x), na.rm = na.rm, w = weights))
}

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
```

`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. 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^(2/3), na.rm = TRUE)
```

Arguments

`x`	A numeric vector.
`tol`	tolerance value. The default is `.Machine$double.eps^(2/3)`, an arbitrarily small tolerance.
`na.rm`	Should `NA` values be ignored? 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 Date-Time vector.

Value

A `Date` vector.

reset_timeplyr_options *Reset 'timeplyr' options*

Description

Reset 'timeplyr' options

Usage

```
reset_timeplyr_options()
```

Value

Resets the timeplyr global options (prefixed with `"timeplyr."`):
roll_month & roll_dst.

resolution *Time resolution & granularity*

Description

The definitions of resolution and granularity may evolve over time but currently the resolution defines the smallest timespan that differentiates two non-fractional instances in time. The granularity defines the smallest common time difference. A practical example would be when using dates to record data with a monthly frequency. In this case the granularity is 1 month, whereas the resolution of the data type `Date` is 1 day. Therefore the resolution depends only on the data type whereas the granularity depends on the frequency with which the data is recorded.

Usage

```
resolution(x, ...)

granularity(x, ...)
```

Arguments

`x`	Time vector. E.g. a `Date`, `POSIXt`, `numeric` or any time-based vector.
`...`	Further arguments passed to methods.

Details

For dates and date-times, the argument `exact = TRUE` can be used to detect monthly/yearly granularity. In some cases this can be slow and memory-intensive so it is advised to set this to `FALSE` in these cases.

The default for dates is `exact = TRUE` whereas the default for date-times is `exact = FALSE`.

Value

A timespan object.

`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, ...)

## Default S3 method:
roll_lag(x, n = 1L, g = NULL, fill = NULL, ...)

## S3 method for class 'ts'
roll_lag(x, n = 1L, g = NULL, fill = NULL, ...)

## S3 method for class 'zoo'
roll_lag(x, n = 1L, g = NULL, fill = NULL, ...)

roll_diff(x, n = 1L, ...)

## Default S3 method:
roll_diff(x, n = 1L, g = NULL, fill = NULL, differences = 1L, ...)

## S3 method for class 'ts'
roll_diff(x, n = 1L, g = NULL, fill = NULL, differences = 1L, ...)

## S3 method for class 'zoo'
roll_diff(x, n = 1L, g = NULL, fill = NULL, differences = 1L, ...)

diff_(
  x,
  n = 1L,
  differences = 1L,
  order = NULL,
  run_lengths = NULL,
  fill = NULL
)
```

Arguments

x	A vector or data frame.

n	Lag. This will be recycled to match the length of x and can be negative.
...	Arguments passed onto appropriate method.
g	Grouping vector. This can be a vector, data frame or GRP object.
fill	Value to fill the first n elements.
differences	Number indicating the number of times to recursively apply the differencing algorithm. If `length(n) == 1`, i.e the lag is a scalar integer, an optimised method is used which avoids recursion entirely. If `length(n) != 1` then simply recursion is used.
order	Optionally specify an ordering with which to apply the lags/differences. This is useful for example when applying lags chronologically using an unsorted time variable.
run_lengths	Optional integer vector of run lengths that defines the size of each lag run. For example, supplying `c(5, 5)` applies lags to the first 5 elements and then essentially resets the bounds and applies lags to the next 5 elements as if they were an entirely separate and standalone vector. This is particularly useful in conjunction with the order argument to perform a by-group lag.

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.

`roll_diff()` silently returns NA when there is integer overflow. Both `roll_lag()` and `roll_diff()` apply recursively to list elements.

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 cheapr::lag_sequence()
# Differences lagged at 5, first 5 differences are compared to x[1]
roll_diff(x, cheapr::lag_sequence(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)
```

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.

Value

A filled vector of `x` the same length as `x`.

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)
```

`timespan` *Timespans*

Description

Timespans

Usage

```
timespan(units, num = 1L, ...)

new_timespan(units, num = 1L)

is_timespan(x)
```

```
timespan_unit(x)

timespan_num(x)
```

Arguments

units	A unit of time, e.g. "days", "3 weeks", lubridate::weeks(3), or just a numeric vector.
num	Number of units. E.g. units = "days" and num = 3 produces a timespan width of 3 days.
...	Further arguments passed onto methods.
x	A timespan.

Details

timespan() can be used to create objects of class 'timespan' which are used widely in timeplyr.

new_timespan() is a bare-bones version that does no checking or string parsing and is intended for fast timespan creation.

timespan_unit() is a helper that extracts the unit of time of the timespan.

timespan_num() is a helper that extracts the number of units of time.

Value

A timespan object.

Examples

```
library(timeplyr)

timespan("week")
timespan("day")
timespan("decade")

# Multiple units of time

timespan("10 weeks")
timespan("1.5 hours")

# These are all equivalent
timespan(NULL, 3);timespan(3);timespan(NA_character_, 3)
```

time_add *Add/subtract timespans to dates and date-times*

Description

A very fast method of adding time units to dates and date-times.

Usage

```
time_add(
  x,
  timespan,
  n = 1L,
  roll_month = getOption("timeplyr.roll_month", "xlast"),
  roll_dst = getOption("timeplyr.roll_dst", c("NA", "xfirst"))
)

time_subtract(
  x,
  timespan,
  n = 1L,
  roll_month = getOption("timeplyr.roll_month", "xlast"),
  roll_dst = getOption("timeplyr.roll_dst", c("NA", "xfirst"))
)

time_floor(x, timespan, week_start = getOption("lubridate.week.start", 1))

time_ceiling(
  x,
  timespan,
  week_start = getOption("lubridate.week.start", 1),
  change_on_boundary = is_date(x)
)

time_round(x, timespan, week_start = getOption("lubridate.week.start", 1))
```

Arguments

x	Time vector. E.g. a Date, POSIXt, numeric or any time-based vector.
timespan	timespan.
n	[numeric(1)] - Number of timespans. This is mostly sugar as this can easily be specified by timespan().
roll_month	See ?timechange::time_add. Additional choices include xlast (default) and xfirst. These work conceptually similar to skipped DST intervals.
roll_dst	See ?timechange::time_add.

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.

change_on_boundary — ?timechange::time_floor

Details

The methods are continuously being improved over time. Date arithmetic should be very fast regardless of the timespan supplied. Date-time arithmetic, specifically when supplied days, weeks, months and years, is being improved.

Value

A date, date-time, or other time-based vector.

---

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, width = NULL, .name = NULL, .add = TRUE)

time_tbl_time_col(x)
```

Arguments

data	A data frame.
time	Time variable (**data-masking**). E.g., a Date, POSIXt, numeric or any time variable.
width	A timespan.
.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 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(fastplyr)
library(nycflights13)
library(lubridate)

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

hourly_flights

monthly_flights <- flights |>
  time_by(time_hour, "month")
weekly_flights <- flights |>
  time_by(time_hour, "week")

monthly_flights |>
  f_count()

weekly_flights |>
  f_summarise(n = n(), arr_delay = mean(arr_delay, na.rm = TRUE))

# To aggregate multiple variables, use `time_cut_width`

flights |>
  f_count(week = time_cut_width(time_hour, months(3)))
```

`time_cut_n` *Cut dates and datetimes into regularly spaced date or datetime intervals*

Description

Useful functions especially for when plotting time-series.

`time_cut_n` makes approximately `n` groups of equal time range. It prioritises the highest time unit possible, making axes look less cluttered and thus prettier.

`time_breaks` returns only the breakpoints.

`time_breakpoints` is a newer and faster alternative to `time_breaks` which differs in that it calls `range()` on the input data and therefore need only work with a vector of 2 values, unlike `time_breaks` which requires more data points to create better looking breaks.

Usage

```
time_cut_n(
  x,
  n = 5,
  timespan = NULL,
  from = NULL,
  to = NULL,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1)
)

time_cut_width(x, timespan = granularity(x), from = NULL, to = NULL)

time_breaks(
  x,
  n = 5,
  timespan = NULL,
  from = NULL,
  to = NULL,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1)
)

time_breakpoints(x, n = 10)

time_cut(
  x,
  n = 5,
  timespan = NULL,
  from = NULL,
  to = NULL,
  time_floor = FALSE,
  week_start = getOption("lubridate.week.start", 1)
)
```

Arguments

x	Time vector. E.g. a Date, POSIXt, numeric or any time-based vector.
n	Number of breaks.
timespan	timespan.
from	Start time.
to	End time.
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.

Details

To retrieve regular time breaks that simply spans the range of `x`, use `time_seq()`to manually specify the range and time width or `time_grid()` to use the range of the supplied data.

By default `time_cut_n()` 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.

`time_breakpoints` does the same but using a different internal method.

Value

`time_breaks` and `time_breakpoints` both return a vector of breakpoints
`time_cut_n` and `time_cut_width` returns a `time_interval`

Examples

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

time_cut_n(1:10, n = 5)

# Easily create custom time breaks
df <- nycflights13::flights |>
  f_slice_sample(n = 100) |>
  with_local_seed(.seed = 8192821) |>
  f_select(time_hour) |>
  fastplyr::f_arrange(time_hour) |>
  mutate(date = as_date(time_hour))

# time_cut_n() and time_breaks() automatically find a
# suitable way to cut the data
time_cut_n(df$date) |>
  interval_count()
# Works with datetimes as well
time_cut_n(df$time_hour, n = 5) |>
  interval_count()
time_cut_n(df$date, timespan = "month") |>
  interval_count()
# Just the breaks
time_breaks(df$date, n = 5, timespan = "month")

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

# To get exact breaks at regular intervals, use time_grid
weekly_breaks <- time_grid(
  df$date, "5 weeks",
  from = floor_date(min(df$date), "week", week_start = 1)
```

```
)
weekly_labels <- format(weekly_breaks, "%b-%d")
df |>
  time_by(date, "week", .name = "date") |>
  f_count() |>
  mutate(date = interval_start(date)) |>
  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, timespan = 1L)
```

Arguments

`x`	Start date or datetime.
`y`	End date or datetime.
`timespan`	A timespan used to divide the difference.

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), "days")
time_diff(today(), today() + days((0:3) * 7), weeks(1))
time_diff(today(), today() + days(100), timespan("days", 1:100))
time_diff(1, 1 + 0:100, 3)
```

`time_elapsed` *Fast grouped time elapsed*

Description

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

Usage

```
time_elapsed(
  x,
  timespan = granularity(x),
  g = NULL,
  rolling = TRUE,
  fill = NA,
  na_skip = TRUE
)
```

Arguments

`x`	Time vector. E.g. a `Date`, `POSIXt`, `numeric` or any time-based vector.
`timespan`	timespan.
`g`	Object to be used for grouping `x`, passed onto `collapse::GRP()`.
`rolling`	If `TRUE` (the default) then lagged time differences are calculated on a rolling basis, essentially like `diff()`. 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 = "3 days")
time_elapsed(x)
time_elapsed(x, "days", rolling = FALSE)

# Grouped example
set.seed(99)
g <- sample.int(3, 25, TRUE)

time_elapsed(x, "days", 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,
  .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:

- 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 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.
When `rolling = FALSE` events with a `t_elapsed >= window` since the first event of the corresponding episode are defined as a new episode.
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.
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?
The default is `TRUE`.
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`.
Default is `0`.

`.add` Should episodic variables be added to the data?
If `FALSE` (the default), then only the relevant variables are returned.
If `TRUE`, the episodic variables are added to the original data. 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.
`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)`.

`.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, which means one is not interested in episodes, simply use `time_elapsed()` instead.

To find implicit missing gaps in time, set `window` to `1` and `switch_on_boundary` to `FALSE`. Any event classified as an episode in this scenario is an event following a gap in time.

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, "week", window = 1)
```

```
events

episodes <- events |>
  filter(ep_id_new > 1)
nrow(fastplyr::f_distinct(episodes, origin, dest)) # 55 origin-destinations

# As expected summer months saw the least number of
# dry-periods
episodes |>
  ungroup() |>
  time_by(ep_start, "week", .name = "ep_start") |>
  count(ep_start = interval_start(ep_start)) |>
  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,
  sort = TRUE
)

time_complete(
  data,
  time = NULL,
  ...,
  .by = NULL,
  time_by = NULL,
  from = NULL,
  to = NULL,
  sort = TRUE,
  fill = NULL
)
```

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	A timespan.
from	Time series start date.
to	Time series end date.
sort	Logical. If TRUE expanded/completed variables are sorted.
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 completing implicit time gaps and creating time sequences by group.

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 |>
  fastplyr::f_count(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_gaps` *Gaps in a regular time sequence*

Description

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

Usage

```
time_gaps(
  x,
  timespan = granularity(x),
  g = NULL,
  use.g.names = TRUE,
  check_time_regular = FALSE
)

time_num_gaps(
  x,
  timespan = granularity(x),
  g = NULL,
  use.g.names = TRUE,
  na.rm = TRUE,
  check_time_regular = FALSE
)

time_has_gaps(
  x,
  timespan = granularity(x),
  g = NULL,
  use.g.names = TRUE,
  na.rm = TRUE,
  check_time_regular = FALSE
)
```

Arguments

`x`	Time vector. E.g. a `Date`, `POSIXt`, `numeric` or any time-based vector.
`timespan`	timespan.
`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`.

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(fastplyr)
library(lubridate)
library(nycflights13)
missing_dates(flights$time_hour)
time_has_gaps(flights$time_hour)
time_num_gaps(flights$time_hour)
length(time_gaps(flights$time_hour))
time_num_gaps(flights$time_hour, g = flights$origin)

# Number of missing hours by origin and dest
flights |>
  f_group_by(origin, dest) |>
  f_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_tbl

Examples

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

# It's as easy as this
AirPassengers |>
  ts_as_tbl() |>
  time_ggplot(time, value)

# And this
EuStockMarkets |>
  ts_as_tbl() |>
  time_ggplot(time, value, group)

# Converting this to monthly averages

EuStockMarkets |>
  ts_as_tbl() |>
  mutate(month = year_month_decimal(time)) |>
  summarise(avg = mean(value),
            .by = c(group, month)) |>
  time_ggplot(month, avg, 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_tbl() |>
  time_ggplot(time, value)
```

`time_grid` *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_grid(x, timespan = granularity(x), from = NULL, to = NULL)

time_complete_missing(x, timespan = granularity(x))

time_grid_size(x, timespan = granularity(x), from = NULL, to = NULL)
```

Arguments

`x`	Time vector. E.g. a `Date`, `POSIXt`, `numeric` or any time-based vector.
`timespan`	timespan.
`from`	Start time.
`to`	End time.

Value

Vectors (typically the same class as `x`) of varying lengths depending on the arguments supplied.

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_grid_size(x) - length(unique(x))

# Time sequence that spans the data
length(time_grid(x)) # Automatically detects hour granularity
time_grid(x, "month")
time_grid(x, from = floor_date(min(x), "month"), to = today(),
          timespan = timespan("month"))

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

# Summarise time into higher intervals
quarters <- time_cut_width(y, "quarter")
interval_count(quarters)
```

`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, timespan = granularity(x), g = NULL, na_skip = TRUE, shift = 1L)
```

Arguments

`x`	Time vector. E.g. a `Date`, `POSIXt`, `numeric` or any time-based vector.
`timespan`	timespan.
`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`.
`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_interval` *S3-based Time Intervals (Currently very experimental and so subject to change)*

Description

Inspired by both 'lubridate' and 'ivs', `time_interval` objects are lightweight S3 objects of a fixed width. This enables fast and flexible representation of time data such as months, weeks, and more. They are all left closed, right open intervals.

Usage

```
time_interval(start = integer(), width = resolution(start))

is_time_interval(x)

new_time_interval(start, width)

interval_start(x)

interval_end(x)

interval_width(x)

interval_count(x)

interval_range(x)
```

Arguments

start	Start time. E.g a Date, POSIXt, numeric and more.
width	Interval width supplied as a timespan. By default this is the resolution of a time vector so for example, a date's resolution is exactly 1 day, therefore time_interval(Sys.Date()) simply represents today's date as an interval.
x	A time_interval.

Details

Currently because of limitations with the S3/S4 system, one can't use time intervals directly with lubridate periods. To navigate around this, timeplyr::timespan() can be used. e.g. instead of interval / weeks(3), use interval / timespan(weeks(3)) or even interval / "3 weeks". where interval is a time_interval.

To perform interval algebra it is advised to use the 'ivs' package. To convert a time_interval into an ivs_iv, use ivs::iv(interval_start(x), interval_end(x)).

Value

An object of class time_interval.
is_time_interval returns a logical of length 1.
interval_start returns the start times.
interval_end returns the end times.
interval_width returns the width of the interval as a timespan.
interval_count returns a data frame of unique intervals and their counts.
interval_range returns a the range of the interval.
new_time_interval is a bare-bones version of time_interval() that performs no checks.

See Also

interval_start

Examples

```
library(dplyr)
library(timeplyr)
library(lubridate)
x <- 1:10
int <- time_interval(x, 100)
int

month_start <- floor_date(today(), unit = "months")
month_int <- time_interval(month_start, "month")
month_int

interval_start(month_int)
interval_end(month_int)

# Divide an interval into different time units
```

```
time_interval(today(), years(10)) / timespan("year")

# Cutting Sepal Length into blocks of width 1
int <- time_cut_width(iris$Sepal.Length, 1)
interval_count(int)
```

`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,
  timespan = granularity(x),
  g = NULL,
  use.g.names = TRUE,
  na.rm = TRUE,
  allow_gaps = FALSE,
  allow_dups = FALSE
)
```

Arguments

`x`	Time vector. E.g. a `Date`, `POSIXt`, `numeric` or any time-based vector.
`timespan`	timespan.
`g`	Grouping object passed directly to `collapse::GRP()`. This can for example be a vector or data frame. 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`.
`allow_gaps`	Should gaps be allowed? Default is `FALSE`.
`allow_dups`	Should duplicates be allowed? Default is `FALSE`.

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)

# No duplicates or gaps allowed by default
time_is_regular(x)
time_is_regular(y)

increment <- 1

# duplicates and gaps allowed
time_is_regular(x, increment, allow_dups = TRUE, allow_gaps = TRUE)
time_is_regular(y, increment, allow_dups = TRUE, allow_gaps = TRUE)

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

# Grouped
eu_stock <- ts_as_tbl(EuStockMarkets)
eu_stock <- eu_stock |>
  mutate(date = as_date(
    date_decimal(time)
  ))

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

`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 = timespan(Inf),
  time = NULL,
  weights = NULL,
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE,
  na.rm = TRUE,
  ...
)

time_roll_mean(
  x,
  window = timespan(Inf),
  time = NULL,
  weights = NULL,
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE,
  na.rm = TRUE,
  ...
)

time_roll_growth_rate(
  x,
  window = timespan(Inf),
  time = NULL,
  time_step = NULL,
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE,
  na.rm = TRUE
)

time_roll_window_size(
  time,
  window = timespan(Inf),
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE
```

```
)

time_roll_window(
  x,
  window = timespan(Inf),
  time = NULL,
  g = NULL,
  partial = TRUE,
  close_left_boundary = FALSE
)

time_roll_apply(
  x,
  window = timespan(Inf),
  fun,
  time = NULL,
  g = NULL,
  partial = TRUE,
  unlist = FALSE,
  close_left_boundary = FALSE
)
```

Arguments

x	Numeric vector.
window	Time window size as a timespan.
time	(Optional) time index. 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), 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.
...	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. It is currently only used only in time_roll_growth_rate. 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)
library(fastplyr)

time <- time_seq(today(), today() + weeks(3), "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 = days(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 = days(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)
```

```
new_tbl(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_tbl(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 = NULL,
  to = NULL,
  time_by = NULL,
  length.out = NULL,
  roll_month = getOption("timeplyr.roll_month", "xlast"),
```

```
  roll_dst = getOption("timeplyr.roll_dst", c("NA", "xfirst"))
)

time_seq_sizes(from, to, timespan)

time_seq_v(
  from,
  to,
  timespan,
  roll_month = getOption("timeplyr.roll_month", "xlast"),
  roll_dst = getOption("timeplyr.roll_dst", c("NA", "xfirst"))
)

time_seq_v2(
  sizes,
  from,
  timespan,
  roll_month = getOption("timeplyr.roll_month", "xlast"),
  roll_dst = getOption("timeplyr.roll_dst", c("NA", "xfirst"))
)
```

Arguments

from	Start time.
to	End time.
time_by	A timespan. This argument may be renamed in the future.
length.out	Length of the sequence.
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.
timespan	timespan.
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.

Examples

```
library(timeplyr)
library(lubridate)

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

time_seq(today, today %m+% months(1), time = "day")
time_seq(today, length.out = 10, time = "day")
time_seq(today, length.out = 10, time = "hour")

time_seq(today, today %m+% months(1), time = timespan("days", 1)) # Alternative
time_seq(today, today + years(1), time = "week")
time_seq(today, today + years(1), time = "fortnight")
time_seq(today, today + years(1), time = "year")
time_seq(today, today + years(10), time = "year")
time_seq(today, today + years(100), time = "decade")

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

# You can seamlessly mix dates and datetimes with no errors.
time_seq(now, today + days(3), time = "day")
time_seq(now, today + days(3), time = "hour")
time_seq(today, now + days(3), time = "day")
time_seq(today, now + days(3), time = "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 = months(1), length.out = 12)
time_seq(start, time = dmonths(1), length.out = 12)
```

```
# 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 = "year",
         roll_month = "NA")
time_seq(leap, to = end, time = "year",
         roll_month = "postday")
time_seq(leap, to = end, time = "year",
         roll_month = getOption("timeplyr.roll_month", "xlast"))
```

`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,
  timespan = granularity(x),
  threshold = 1,
  g = NULL,
  na_skip = TRUE,
  rolling = TRUE,
  switch_on_boundary = FALSE
)
```

Arguments

`x`	Time vector. E.g. a `Date`, `POSIXt`, `numeric` or any time-based vector.
`timespan`	timespan.
`threshold`	Threshold such that when the time elapsed exceeds this, the sequence ID is incremented by 1. For example, if `timespan = "days"` and `threshold = 2`, then when 2 days have passed, a new ID is created. Furthermore, `threshold` generally need not be supplied as `timespan = "3 days"` & `threshold = 1` is identical to `timespan = "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.

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 = "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)
# A new ID when >= 2 weeks has passed since the last time point
time_seq_id(x, threshold = 2, switch_on_boundary = TRUE)
# A new ID when at least 4 cumulative weeks have passed
time_seq_id(x, timespan = "4 weeks",
            switch_on_boundary = TRUE, rolling = FALSE)
# A new ID when more than 4 cumulative weeks have passed
time_seq_id(x, timespan = "4 weeks",
            switch_on_boundary = FALSE, rolling = FALSE)
```

---

transform_year_month *Additional ggplot2 scales*

Description

Additional scales and transforms for use with year_months and year_quarters in ggplot2.

Usage

```
transform_year_month()

transform_year_quarter()

scale_x_year_month(...)

scale_x_year_quarter(...)

scale_y_year_month(...)

scale_y_year_quarter(...)
```

Arguments

...	Arguments passed to `scale_x_continuous` and `scale_y_continuous`.

Value

A ggplot2 scale or transform.

---

ts_as_tbl *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_tbl(x, name = "time", value = "value", group = "group")

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

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

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

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

## S3 method for class 'timeSeries'
ts_as_tbl(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_tbl(uts)

## Multivariate
mts <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 1), frequency = 12)
mts_tbl <- ts_as_tbl(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_tbl(x)
x <- zoo::zoo(matrix(1:12, 4, 3), as.Date("2003-01-01") + 0:3)
ts_as_tbl(x)
```

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)

year_month_decimal(x)

decimal_year_month(x)

YQ(length = 0L)

year_quarter_decimal(x)

decimal_year_quarter(x)
```

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