Package 'tsibble'

January 31, 2020

Type Package

Title Tidy Temporal Data Frames and Tools

Version 0.8.6

Description Provides a 'tbl_ts' class (the 'tsibble') for
temporal data in an data- and model-oriented format. The 'tsibble' provides
tools to easily manipulate and analyse temporal data, such as filling in
time gaps and aggregating over calendar periods.

License GPL-3

URL https://tsibble.tidyverts.org

BugReports https://github.com/tidyverts/tsibble/issues

Depends R (>= 3.2.0)

Imports anytime (>= 0.3.1),
dplyr (>= 0.8.1),
lifecycle,
lubridate (>= 1.7.0),
purrr (>= 0.2.3),
rlang (>= 0.2.0),
tibble (>= 2.0.1),
tidyselect,
vctrs (>= 0.2.0)

Suggests covr,
crayon,
furrr,
ggplot2 (>= 2.2.0),
hms,
knitr,
nanotime,
nycflights13 (>= 1.0.0),
pillar (>= 1.0.1),
rmarkdown,
spelling,
testthat,
tidyr (>= 1.0.0),
timeDate

VignetteBuilder knitr

RdMacros lifecycle

ByteCompile true

Encoding UTF-8

Language en-GB

LazyData true

Roxygen list(markdown = TRUE)

RoxygenNote 7.0.2

R **topics documented:**

---

tsibble-package *tsibble: tidy temporal data frames and tools*

Description

The **tsibble** package provides a data class of `tbl_ts` to represent tidy temporal data. A tsibble consists of a time index, key, and other measured variables in a data-centric format, which is built on top of the tibble.

Index

An extensive range of indices are supported by tsibble: native time classes in R (such as `Date`, `POSIXct`, and `difftime`) and tsibble's new additions (such as yearweek, yearmonth, and year-quarter). Some commonly-used classes have built-in support too, including `ordered`, `hms::hms`, `zoo::yearmon`, `zoo::yearqtr`, and `nanotime`.

For a `tbl_ts` of regular interval, a choice of index representation has to be made. For example, a monthly data should correspond to time index created by yearmonth or `zoo::yearmon`, instead of `Date` or `POSIXct`. Because months in a year ensures the regularity, 12 months every year. However, if using `Date`, a month containing days ranges from 28 to 31 days, which results in irregular time space. This is also applicable to year-week and year-quarter.

Tsibble supports arbitrary index classes, as long as they can be ordered from past to future. To support a custom class, you need to define `index_valid()` for the class and calculate the interval through `interval_pull()`.

Key

Key variable(s) together with the index uniquely identifies each record:

- Empty: an implicit variable. `NULL` resulting in a univariate time series.

- A single variable: For example, `data(pedestrian)` use the bare `Sensor` as the key.

- Multiple variables: For example, Declare `key = c(Region, State, Purpose)` for `data(tourism)`. Key can be created in conjunction with tidy selectors like `starts_with()`.

Interval

The interval function returns the interval associated with the tsibble.

- Regular: the value and its time unit including "nanosecond", "microsecond", "millisecond", "second", "minute", "hour", "day", "week", "month", "quarter", "year". An unrecognisable time interval is labelled as "unit".

- Irregular: as_tsibble(regular = FALSE) gives the irregular tsibble. It is marked with !.

- Unknown: Not determined (?), if it's an empty tsibble, or one entry for each key variable.

An interval is obtained based on the corresponding index representation:

- integerish numerics between 1582 and 2499: "year" (Y). Note the year of 1582 saw the beginning of the Gregorian Calendar switch.

- yearquarter/yearqtr: "quarter" (Q)

- yearmonth/yearmon: "month" (M)

- yearweek: "week" (W)

- Date: "day" (D)

- difftime: "week" (W), "day" (D), "hour" (h), "minute" (m), "second" (s)

- POSIXt/hms: "hour" (h), "minute" (m), "second" (s), "millisecond" (us), "microsecond" (ms)

- nanotime: "nanosecond" (ns)

- other numerics &ordered (ordered factor): "unit" When the interval cannot be obtained due to the mismatched index format, an error is issued.

The interval is invariant to subsetting, such as filter(), slice(), and [.tbl_ts. But if the result is an empty tsibble, the interval is always unknown. When joining a tsibble with other data sources and aggregating to different time scales, the interval gets re-calculated.

Time zone

Time zone corresponding to index will be displayed if index is POSIXct. ? means that the obtained time zone is a zero-length character "".

Print options

The tsibble package fully utilises the print method from the tibble. Please refer to tibble::tibble-package to change display options.

Author(s)

Maintainer: Earo Wang <earo.wang@gmail.com> (ORCID)

Authors:

- Di Cook (ORCID) [thesis advisor]

- Rob Hyndman (ORCID) [thesis advisor]

- Mitchell O'Hara-Wild (ORCID)

See Also

Useful links:

- https://tsibble.tidyverts.org

- Report bugs at https://github.com/tidyverts/tsibble/issues

Examples

```
# create a tsibble w/o a key ----
tsibble(
  date = as.Date("2017-01-01") + 0:9,
  value = rnorm(10)
)

# create a tsibble with one key ----
tsibble(
  qtr = rep(yearquarter("2010-01") + 0:9, 3),
  group = rep(c("x", "y", "z"), each = 10),
  value = rnorm(30),
  key = group
)
```

---

as.ts.tbl_ts *Coerce a tsibble to a time series*

Description

Stable

Usage

```
## S3 method for class 'tbl_ts'
as.ts(x, value, frequency = NULL, fill = NA_real_, ...)
```

Arguments

x	A tbl_ts object.
value	A measured variable of interest to be spread over columns, if multiple measures.
frequency	A smart frequency with the default NULL. If set, the preferred frequency is passed to ts().
fill	A value to replace missing values.
...	Ignored for the function.

Value

A ts object.

Examples

```
# a monthly series
x1 <- as_tsibble(AirPassengers)
as.ts(x1)

# equally spaced over trading days, not smart enough to guess frequency
x2 <- as_tsibble(EuStockMarkets)
head(as.ts(x2, frequency = 260))
```

as_tibble.tbl_ts   *Coerce to a tibble or data frame*

Description

Coerce to a tibble or data frame

Usage

```
## S3 method for class 'tbl_ts'
as_tibble(x, ...)

## S3 method for class 'tbl_ts'
as.data.frame(x, row.names = NULL, optional = FALSE, ...)
```

Arguments

x   A tbl_ts.

...   Ignored.

row.names   NULL or a character vector giving the row names for the data frame. Missing values are not allowed.

optional   logical. If TRUE, setting row names and converting column names (to syntactic names: see make.names) is optional. Note that all of R's **base** package as.data.frame() methods use optional only for column names treatment, basically with the meaning of data.frame(*,check.names = !optional). See also the make.names argument of the matrix method.

Examples

```
as_tibble(pedestrian)
```

as_tsibble   *Coerce to a tsibble object*

Description

Stable

Usage

```
as_tsibble(
  x,
  key = NULL,
  index,
  regular = TRUE,
  validate = TRUE,
  .drop = TRUE,
  ...
)
```

```
## S3 method for class 'data.frame'
as_tsibble(
  x,
  key = NULL,
  index,
  regular = TRUE,
  validate = TRUE,
  .drop = TRUE,
  ...
)

## S3 method for class 'ts'
as_tsibble(x, ..., tz = "UTC")

## S3 method for class 'mts'
as_tsibble(x, ..., tz = "UTC", pivot_longer = TRUE, gather = deprecated())
```

Arguments

x	Other objects to be coerced to a tsibble (tbl_ts).
key	Unquoted variable(s) that uniquely determine time indices. NULL for empty key, and c() for multiple variables. It works with tidy selector (e.g. dplyr::starts_with()).
index	A bare (or unquoted) variable to specify the time index variable.
regular	Regular time interval (TRUE) or irregular (FALSE). The interval is determined by the greatest common divisor of index column, if TRUE.
validate	TRUE suggests to verify that each key or each combination of key variables leads to unique time indices (i.e. a valid tsibble). If you are sure that it's a valid input, specify FALSE to skip the checks.
.drop	If TRUE, empty key groups are dropped.
...	Other arguments passed on to individual methods.
tz	Time zone. May be useful when a ts object is more frequent than daily.
pivot_longer	TRUE gives a "longer" form of the data, otherwise as is.
gather	**Defunct** Please use pivot_longer instead.

Value

A tsibble object.

See Also

tsibble

Examples

```
# coerce tibble to tsibble w/o a key
tbl1 <- tibble(
  date = as.Date("2017-01-01") + 0:9,
  value = rnorm(10)
)
as_tsibble(tbl1)
```

```
# supply the index to suppress the message
as_tsibble(tbl1, index = date)

# coerce tibble to tsibble with a single variable for key
# "date" is automatically considered as the index var, and "group" is the key
tbl2 <- tibble(
  mth = rep(yearmonth("2017-01") + 0:9, 3),
  group = rep(c("x", "y", "z"), each = 10),
  value = rnorm(30)
)
as_tsibble(tbl2, key = group)
as_tsibble(tbl2, key = group, index = mth)

# create a tsibble with multiple variables for key
tbl3 <- tibble(
  mth = rep(yearmonth("2010 Jan") + 0:8, each = 3),
  xyz = rep(c("x", "y", "z"), each = 9),
  abc = rep(letters[1:3], times = 9),
  value = rnorm(27)
)
as_tsibble(tbl3, key = c(xyz, abc))
# coerce ts to tsibble
as_tsibble(AirPassengers)
as_tsibble(sunspot.year)
as_tsibble(sunspot.month)
as_tsibble(austres)
# coerce mts to tsibble
z <- ts(matrix(rnorm(300), 100, 3), start = c(1961, 1), frequency = 12)
as_tsibble(z)
as_tsibble(z, pivot_longer = FALSE)
```

| build_tsibble | *Low-level constructor for a tsibble object* |

Description

build_tsibble() creates a tbl_ts object with more controls. It is useful for creating a tbl_ts internally inside a function, and it allows developers to determine if the time needs ordering and the interval needs calculating.

Usage

```
build_tsibble(
  x,
  key = NULL,
  key_data = NULL,
  index,
  index2 = index,
  ordered = NULL,
  interval = TRUE,
  validate = TRUE,
  .drop = key_drop_default(x)
)
```

Arguments

x	A `data.frame`, `tbl_df`, `tbl_ts`, or other tabular objects.
key	Unquoted variable(s) that uniquely determine time indices. NULL for empty key, and `c()` for multiple variables. It works with tidy selector (e.g. `dplyr::starts_with()`).
key_data	A data frame containing key variables and `.rows`. When a data frame is supplied, the argument key will be ignored.
index	A bare (or unquoted) variable to specify the time index variable.
index2	A candidate of `index` to update the index to a new one when index_by. By default, it's identical to `index`.
ordered	The default of NULL arranges the key variable(s) first and then index from past to future. TRUE suggests to skip the ordering as x in the correct order. FALSE checks the ordering and may give a warning.
interval	TRUE automatically calculates the interval, and FALSE for irregular interval. Use the specified interval via `new_interval()` as is.
validate	TRUE suggests to verify that each key or each combination of key variables leads to unique time indices (i.e. a valid tsibble). If you are sure that it's a valid input, specify FALSE to skip the checks.
.drop	If TRUE, empty key groups are dropped.

Examples

```
# Prepare `pedestrian` to use a new index `Date` ----
pedestrian %>%
  build_tsibble(
    key = !!key_vars(.), index = !!index(.), index2 = Date,
    interval = interval(.)
  )
```

---

count_gaps	*Count implicit gaps*

Description

Count implicit gaps

Usage

```
count_gaps(.data, .full = FALSE, .name = c(".from", ".to", ".n"))
```

Arguments

.data	A `tbl_ts`.
.full	FALSE to find gaps for each series within its own period. TRUE to find gaps over the entire time span of the data.
.name	Strings to name new columns.

Value

A tibble contains:

- the "key" of the tbl_ts
- ".from": the starting time point of the gap
- ".to": the ending time point of the gap
- ".n": the number of implicit missing observations during the time period

See Also

Other implicit gaps handling: [fill_gaps()](), [has_gaps()](), [scan_gaps()]()

Examples

```
ped_gaps <- pedestrian %>%
  count_gaps(.full = TRUE)
ped_gaps
if (!requireNamespace("ggplot2", quietly = TRUE)) {
  stop("Please install the ggplot2 package to run these following examples.")
}
library(ggplot2)
ggplot(ped_gaps, aes(x = Sensor, colour = Sensor)) +
  geom_linerange(aes(ymin = .from, ymax = .to)) +
  geom_point(aes(y = .from)) +
  geom_point(aes(y = .to)) +
  coord_flip() +
  theme(legend.position = "bottom")
```

---

difference   *Lagged differences*

Description

Stable

Usage

```
difference(x, lag = 1, differences = 1, default = NA, order_by = NULL)
```

Arguments

x	A numeric vector.
lag	An positive integer indicating which lag to use.
differences	An positive integer indicating the order of the difference.
default	Value used for non-existent rows, defaults to NA.
order_by	Override the default ordering to use another vector.

Value

A numeric vector of the same length as x.

See Also

[dplyr::lead](#) and [dplyr::lag](#)

Examples

```
# examples from base
difference(1:10, 2)
difference(1:10, 2, 2)
x <- cumsum(cumsum(1:10))
difference(x, lag = 2)
difference(x, differences = 2)
# Use order_by if data not already ordered (example from dplyr)
library(dplyr, warn.conflicts = FALSE)
tsbl <- tsibble(year = 2000:2005, value = (0:5)^2, index = year)
scrambled <- tsbl %>% slice(sample(nrow(tsbl)))

wrong <- mutate(scrambled, diff = difference(value))
arrange(wrong, year)

right <- mutate(scrambled, diff = difference(value, order_by = year))
arrange(right, year)
```

---

fill_gaps   *Turn implicit missing values into explicit missing values*

Description

Stable

Usage

```
fill_gaps(.data, ..., .full = FALSE)
```

Arguments

.data	A tsibble.
...	A set of name-value pairs. The values provided will only replace missing values that were marked as "implicit", and will leave previously existing NA untouched.
	• empty: filled with default NA.
	• filled by values or functions.
.full	FALSE to insert NA for each series within its own period. TRUE to fill NA over the entire time span of the data (a.k.a. fully balanced panel).

See Also

[tidyr::fill](#), [tidyr::replace_na](#) for handling missing values NA.

Other implicit gaps handling: [count_gaps()](#), [has_gaps()](#), [scan_gaps()](#)

Examples

```
library(dplyr)
harvest <- tsibble(
  year = c(2010, 2011, 2013, 2011, 2012, 2014),
  fruit = rep(c("kiwi", "cherry"), each = 3),
  kilo = sample(1:10, size = 6),
  key = fruit, index = year
)

# gaps as default `NA`
fill_gaps(harvest, .full = TRUE)
full_harvest <- fill_gaps(harvest, .full = FALSE)
full_harvest

# replace gaps with a specific value
harvest %>%
  fill_gaps(kilo = 0L)

# replace gaps using a function by variable
harvest %>%
  fill_gaps(kilo = sum(kilo))

# replace gaps using a function for each group
harvest %>%
  group_by_key() %>%
  fill_gaps(kilo = sum(kilo))

# leaves existing `NA` untouched
harvest[2, 3] <- NA
harvest %>%
  group_by_key() %>%
  fill_gaps(kilo = sum(kilo, na.rm = TRUE))

# replace NA
pedestrian %>%
  group_by_key() %>%
  fill_gaps(Count = as.integer(median(Count)))

if (!requireNamespace("tidyr", quietly = TRUE)) {
  stop("Please install the 'tidyr' package to run these following examples.")
}
# use fill() to fill `NA` by previous/next entry
pedestrian %>%
  group_by_key() %>%
  fill_gaps() %>%
  tidyr::fill(Count, .direction = "down")
```

---

    filter_index   *A shorthand for filtering time index for a tsibble*

Description

This shorthand respects time zones and encourages compact expressions.

Usage

```
filter_index(.data, ..., .preserve = FALSE)
```

Arguments

.data	A tsibble.
...	Formulas that specify start and end periods (inclusive), or strings.

- ~ end or . ~ end: from the very beginning to a specified ending period.
- start ~ end: from specified beginning to ending periods.
- start ~ .: from a specified beginning to the very end of the data. Supported index type: POSIXct (to seconds), Date, yearweek, yearmonth/yearmon, yearquarter/yearqtr, hms/difftime & numeric.

.preserve	when FALSE (the default), the grouping structure is recalculated based on the resulting data, otherwise it is kept as is.

System Time Zone ("Europe/London")

There is a known issue of an extra hour gained for a machine setting time zone to "Europe/London", regardless of the time zone associated with the POSIXct inputs. It relates to *anytime* and *Boost*. Use Sys.timezone() to check if the system time zone is "Europe/London". It would be recommended to change the global environment "TZ" to other equivalent names: GB, GB-Eire, Europe/Belfast, Europe/Guernsey, Europe/Isle_of_Man and Europe/Jersey as documented in ?Sys.timezone(), using Sys.setenv(TZ = "GB") for example.

See Also

time_in for a vector of time index

Examples

```
# from the starting time to the end of Feb, 2015
pedestrian %>%
  filter_index(~"2015-02")

# entire Feb 2015, & from the beginning of Aug 2016 to the end
pedestrian %>%
  filter_index("2015-02", "2016-08" ~ .)

# multiple time windows
pedestrian %>%
  filter_index(~"2015-02", "2015-08" ~ "2015-09", "2015-12" ~ "2016-02")

# entire 2015
pedestrian %>%
  filter_index("2015")

# specific
pedestrian %>%
  filter_index("2015-03-23" ~ "2015-10")
pedestrian %>%
  filter_index("2015-03-23" ~ "2015-10-31")
pedestrian %>%
  filter_index("2015-03-23 10" ~ "2015-10-31 12")
```

---

future_slide()   *Sliding window in parallel*

Description

Questioning

The rolling window family will be deprecated in the future. Please consider using the slider package.

Multiprocessing equivalents of slide(), tile(), stretch() prefixed by future_.

- Variants for corresponding types: future_*_lgl(), future_*_int(), future_*_dbl(), future_*_chr(), future_*_dfr(), future_*_dfc().

- Extra arguments .progress and .options for enabling progress bar and the future specific options to use with the workers.

Details

It requires the package **furrr** to be installed. Please refer to furrr for performance and detailed usage.

Examples

```
if (!requireNamespace("furrr", quietly = TRUE)) {
  stop("Please install the furrr package to run these following examples.")
}
## Not run:
library(furrr)
plan(multiprocess)
my_diag <- function(...) {
  data <- list(...)
  fit <- lm(Count ~ Time, data = data)
  tibble(fitted = fitted(fit), resid = residuals(fit))
}
pedestrian %>%
  group_by_key() %>%
  nest() %>%
  mutate(diag = future_map(data, ~ future_pslide_dfr(., my_diag, .size = 48)))

## End(Not run)
# nocov start
```

---

future_stretch()   *Stretching window in parallel*

Description

Questioning

The rolling window family will be deprecated in the future. Please consider using the slider package.

Multiprocessing equivalents of `slide()`, `tile()`, `stretch()` prefixed by future_.

- Variants for corresponding types: future_*_lgl(), future_*_int(), future_*_dbl(), future_*_chr(), future_*_dfr(), future_*_dfc().
- Extra arguments `.progress` and `.options` for enabling progress bar and the future specific options to use with the workers.

---

future_tile()   *Tiling window in parallel*

Description

Questioning

The rolling window family will be deprecated in the future. Please consider using the slider package. Multiprocessing equivalents of `slide()`, `tile()`, `stretch()` prefixed by future_.

- Variants for corresponding types: future_*_lgl(), future_*_int(), future_*_dbl(), future_*_chr(), future_*_dfr(), future_*_dfc().
- Extra arguments `.progress` and `.options` for enabling progress bar and the future specific options to use with the workers.

---

group_by_key   *Group by key variables*

Description

Stable

Usage

```
group_by_key(.data, ..., .drop = key_drop_default(.data))
```

Arguments

`.data`	A `tbl_ts` object.
`...`	Ignored.
`.drop`	When `.drop` = TRUE, empty groups are dropped. See `group_by_drop_default()` for what the default value is for this argument.

Examples

```
tourism %>%
  group_by_key()
```

---

guess_frequency   *Guess a time frequency from other index objects*

Description

Stable

A possible frequency passed to the `ts()` function

Usage

```
guess_frequency(x)
```

Arguments

x   An index object including "yearmonth", "yearquarter", "Date" and others.

Details

If a series of observations are collected more frequently than weekly, it is more likely to have multiple seasonalities. This function returns a frequency value at its smallest. For example, hourly data would have daily, weekly and annual frequencies of 24, 168 and 8766 respectively, and hence it gives 24.

References

https://robjhyndman.com/hyndsight/seasonal-periods/

Examples

```
guess_frequency(yearquarter(seq(2016, 2018, by = 1 / 4)))
guess_frequency(yearmonth(seq(2016, 2018, by = 1 / 12)))
guess_frequency(seq(as.Date("2017-01-01"), as.Date("2017-01-31"), by = 1))
guess_frequency(seq(
  as.POSIXct("2017-01-01 00:00"), as.POSIXct("2017-01-10 23:00"),
  by = "1 hour"
))
```

---

has_gaps   *Does a tsibble have implicit gaps in time?*

Description

Does a tsibble have implicit gaps in time?

Usage

```
has_gaps(.data, .full = FALSE, .name = ".gaps")
```

Arguments

`.data`	A `tbl_ts`.
`.full`	`FALSE` to find gaps for each series within its own period. `TRUE` to find gaps over the entire time span of the data.
`.name`	Strings to name new columns.

Value

A tibble contains "key" variables and new column `.gaps` of TRUE/FALSE.

See Also

Other implicit gaps handling: [count_gaps](), [fill_gaps](), [scan_gaps]()

Examples

```
harvest <- tsibble(
  year = c(2010, 2011, 2013, 2011, 2012, 2013),
  fruit = rep(c("kiwi", "cherry"), each = 3),
  kilo = sample(1:10, size = 6),
  key = fruit, index = year
)
has_gaps(harvest)
has_gaps(harvest, .full = TRUE)
```

---

holiday_aus	*Australian national and state-based public holiday*

Description

Australian national and state-based public holiday

Usage

```
holiday_aus(year, state = "national")
```

Arguments

`year`	A vector of integer(s) indicating year(s).
`state`	A state in Australia including "ACT", "NSW", "NT", "QLD", "SA", "TAS", "VIC", "WA", as well as "national".

Details

Not documented public holidays:

- AFL public holidays for Victoria
- Queen's Birthday for Western Australia
- Royal Queensland Show for Queensland, which is for Brisbane only

This function requires "timeDate" to be installed.

Value

A tibble consisting of `holiday` labels and their associated dates in the year(s).

References

[Public holidays](#)

Examples

```
holiday_aus(2016, state = "VIC")
holiday_aus(2013:2016, state = "ACT")
```

---

index   *Return index variable from a tsibble*

Description

Return index variable from a tsibble

Usage

```
index(x)

index_var(x)

index2(x)

index2_var(x)
```

Arguments

x   A tsibble object.

Examples

```
index(pedestrian)
index_var(pedestrian)
```

---

index_by   *Group by time index and collapse with* summarise()

Description

Stable

`index_by()` is the counterpart of `group_by()` in temporal context, but it only groups the time index. The following operation is applied to each partition of the index, similar to `group_by()` but dealing with index only. `index_by()` + `summarise()` will update the grouping index variable to be the new index. Use `ungroup()` to remove the index grouping vars.

Usage

```
index_by(.data, ...)
```

Arguments

.data	A `tbl_ts`.
...	If empty, grouping the current index. If not empty, a single expression is required for either an existing variable or a name-value pair. A lambda expression is supported, for example `~ as.Date(.)` where `.` refers to the index variable. The index functions that can be used, but not limited:

- lubridate::year: yearly aggregation
- yearquarter: quarterly aggregation
- yearmonth: monthly aggregation
- yearweek: weekly aggregation
- as.Date or lubridate::as_date: daily aggregation
- lubridate::ceiling_date, lubridate::floor_date, or lubridate::round_date: fine-resolution aggregation
- Extract time components functions, such as `lubridate::hour()` & `lubridate::day()`
- other index functions from other packages or self-defined functions

Details

- A `index_by()`-ed tsibble is indicated by `@` in the "Groups" when displaying on the screen.

Examples

```
pedestrian %>% index_by()
# Monthly counts across sensors
library(dplyr, warn.conflicts = FALSE)
monthly_ped <- pedestrian %>%
  group_by_key() %>%
  index_by(Year_Month = ~ yearmonth(.)) %>%
  summarise(
    Max_Count = max(Count),
    Min_Count = min(Count)
  )
monthly_ped
index(monthly_ped)

# Using existing variable
pedestrian %>%
  group_by_key() %>%
  index_by(Date) %>%
  summarise(
    Max_Count = max(Count),
    Min_Count = min(Count)
  )

# Attempt to aggregate to 4-hour interval, with the effects of DST
pedestrian %>%
  group_by_key() %>%
  index_by(Date_Time4 = ~ lubridate::floor_date(., "4 hour")) %>%
  summarise(Total_Count = sum(Count))
```

```
library(lubridate, warn.conflicts = FALSE)
# Annual trips by Region and State
tourism %>%
  index_by(Year = ~ year(.)) %>%
  group_by(Region, State) %>%
  summarise(Total = sum(Trips))

# Rouding to financial year, using a custom function
financial_year <- function(date) {
  year <- year(date)
  ifelse(quarter(date) <= 2, year, year + 1)
}
tourism %>%
  index_by(Year = ~ financial_year(.)) %>%
  summarise(Total = sum(Trips))
```

---

index_valid   *Add custom index support for a tsibble*

Description

Stable

S3 method to add an index type support for a tsibble.

Usage

```
index_valid(x)
```

Arguments

x   An object of index type that the tsibble supports.

Details

This method is primarily used for adding an index type support in as_tsibble.

Value

TRUE/FALSE or NA (unsure)

See Also

interval_pull for obtaining interval for regularly spaced time.

Examples

```
index_valid(seq(as.Date("2017-01-01"), as.Date("2017-01-10"), by = 1))
```

---

interval   *Meta-information of a tsibble*

Description

- `interval()` returns an interval of a tsibble.
- `is_regular` checks if a tsibble is spaced at regular time or not.
- `is_ordered` checks if a tsibble is ordered by key and index.

Usage

```
interval(x)

is_regular(x)

is_ordered(x)
```

Arguments

x   A tsibble object.

Examples

```
interval(pedestrian)
is_regular(pedestrian)
is_ordered(pedestrian)
```

---

interval_pull   *Pull time interval from a vector*

Description

Stable

Assuming regularly spaced time, the `interval_pull()` returns a list of time components as the "interval" class.

Usage

```
interval_pull(x)
```

Arguments

x   A vector of `POSIXct`, `Date`, `yearweek`, `yearmonth`, `yearquarter`, `difftime`/`hms`, `ordered`, `integer`, `numeric`, and `nanotime`.

Details

Extend tsibble to support custom time indexes by defining S3 generics `index_valid()` and `interval_pull()` for them.

Value

an "interval" class (a list) includes "year", "quarter", "month", "week", "day", "hour", "minute", "second", "millisecond", "microsecond", "nanosecond", "unit".

Examples

```
x <- seq(as.Date("2017-10-01"), as.Date("2017-10-31"), by = 3)
interval_pull(x)
```

---

is_duplicated   *Test duplicated observations determined by key and index variables*

Description

Stable

- `is_duplicated()`: a logical scalar if the data exist duplicated observations.
- `are_duplicated()`: a logical vector, the same length as the row number of data.
- `duplicates()`: identical key-index data entries.

Usage

```
is_duplicated(data, key = NULL, index)

are_duplicated(data, key = NULL, index, from_last = FALSE)

duplicates(data, key = NULL, index)
```

Arguments

data	A data frame for creating a tsibble.
key	Unquoted variable(s) that uniquely determine time indices. NULL for empty key, and c() for multiple variables. It works with tidy selector (e.g. `dplyr::starts_with()`).
index	A bare (or unquoted) variable to specify the time index variable.
from_last	TRUE does the duplication check from the last of identical elements.

Examples

```
harvest <- tibble(
  year = c(2010, 2011, 2013, 2011, 2012, 2014, 2014),
  fruit = c(rep(c("kiwi", "cherry"), each = 3), "cherry"),
  kilo = sample(1:10, size = 7)
)
is_duplicated(harvest, key = fruit, index = year)
are_duplicated(harvest, key = fruit, index = year)
are_duplicated(harvest, key = fruit, index = year, from_last = TRUE)
duplicates(harvest, key = fruit, index = year)
```

---

is_tsibble *If the object is a tsibble*

Description

Stable

Usage

```
is_tsibble(x)

is_grouped_ts(x)
```

Arguments

x   An object.

Value

TRUE if the object inherits from the tbl_ts class.

Examples

```
# A tibble is not a tsibble ----
tbl <- tibble(
  date = seq(as.Date("2017-10-01"), as.Date("2017-10-31"), by = 1),
  value = rnorm(31)
)
is_tsibble(tbl)

# A tsibble ----
tsbl <- as_tsibble(tbl, index = date)
is_tsibble(tsbl)
```

---

key *Return key variables*

Description

key() returns a list of symbols; key_vars() gives a character vector.

Usage

```
key(x)

key_vars(x)
```

Arguments

x   A tsibble.

Examples

```
key(pedestrian)
key_vars(pedestrian)

key(tourism)
key_vars(tourism)
```

---

measures   *Return measured variables*

Description

Return measured variables

Usage

```
measures(x)

measured_vars(x)
```

Arguments

x   A tbl_ts.

Examples

```
measures(pedestrian)
measures(tourism)

measured_vars(pedestrian)
measured_vars(tourism)
```

---

new_data   *New tsibble data and append new observations to a tsibble*

Description

Stable

append_row(): add new rows to the end of a tsibble by filling a key-index pair and NA for measured variables.

append_case() is an alias of append_row().

Usage

```
new_data(.data, n = 1L, ...)

## S3 method for class 'tbl_ts'
new_data(.data, n = 1L, keep_all = FALSE, ...)

append_row(.data, n = 1L, ...)
```

Arguments

`.data`	A `tbl_ts`.
`n`	An integer indicates the number of key-index pair to append.
`...`	Passed to individual S3 method.
`keep_all`	If `TRUE` keep all the measured variables as well as index and key, otherwise only index and key.

Examples

```
new_data(pedestrian)
new_data(pedestrian, keep_all = TRUE)
new_data(pedestrian, n = 3)
tsbl <- tsibble(
  date = rep(as.Date("2017-01-01") + 0:2, each = 2),
  group = rep(letters[1:2], 3),
  value = rnorm(6),
  key = group
)
append_row(tsbl)
append_row(tsbl, n = 2)
```

---

new_interval	*Create a time interval*

Description

`new_interval()` creates an interval object with the specified values.

Usage

```
new_interval(...)
```

Arguments

`...`	A list of time units to be included in the interval and their amounts. "year", "quarter", "month", "week", "day", "hour", "minute", "second", "millisecond", "microsecond", "nanosecond", "unit" are supported.

Value

an "interval" class

Examples

```
new_interval(hour = 1, minute = 30)
new_interval(NULL) # irregular interval
new_interval() # unknown interval
```

---

new_tsibble   *Create a subclass of a tsibble*

Description

Create a subclass of a tsibble

Usage

```
new_tsibble(x, ..., class = NULL)
```

Arguments

x	A `tbl_ts`, required.
...	Name-value pairs defining new attributes other than a tsibble.
class	Subclasses to assign to the new object, default: none.

---

partial_slider   *Partially splits the input to a list according to the rolling window size.*

Description

Partially splits the input to a list according to the rolling window size.

Usage

```
partial_slider(
  .x,
  .size = 1,
  .step = 1,
  .fill = NA,
  .align = "right",
  .bind = FALSE
)

partial_pslider(
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .align = "right",
  .bind = FALSE
)
```

Arguments

`.x`	An object to slide over.
`.size`	An integer for window size. If positive, moving forward from left to right; if negative, moving backward (from right to left).
`.step`	A positive integer for calculating at every specified step instead of every single step.
`.fill`	A value to fill at the left/center/right of the data range depending on `.align` (NA by default). NULL means no filling.
`.align`	Align index at the "**r**ight", "**c**entre"/"center", or "**l**eft" of the window. If `.size` is even for center alignment, "centre-right" & "centre-left" is needed.
`.bind`	If `.x` is a list, should `.x` be combined before applying `.f`? If `.x` is a list of data frames, row binding is carried out.
`...`	Additional arguments passed on to the mapped function.

Examples

```
x <- c(1, NA_integer_, 3:5)
slider(x, .size = 3)
partial_slider(x, .size = 3)
```

---

pedestrian	*Pedestrian counts in the city of Melbourne*

Description

A dataset containing the hourly pedestrian counts from 2015-01-01 to 2016-12-31 at 4 sensors in the city of Melbourne.

Usage

```
pedestrian
```

Format

A tsibble with 66,071 rows and 5 variables:

- **Sensor**: Sensor names (key)
- **Date_Time**: Date time when the pedestrian counts are recorded (index)
- **Date**: Date when the pedestrian counts are recorded
- **Time**: Hour associated with Date_Time
- **Counts**: Hourly pedestrian counts

References

Melbourne Open Data Portal

Examples

```
library(dplyr)
data(pedestrian)
# make implicit missingness to be explicit ----
pedestrian %>% fill_gaps()
# compute daily maximum counts across sensors ----
pedestrian %>%
  group_by_key() %>%
  index_by(Date) %>% # group by Date and use it as new index
  summarise(MaxC = max(Count))
```

---

scan_gaps   *Scan a tsibble for implicit missing observations*

Description

Scan a tsibble for implicit missing observations

Usage

```
scan_gaps(.data, .full = FALSE)
```

Arguments

.data	A tbl_ts.
.full	FALSE to find gaps for each series within its own period. TRUE to find gaps over the entire time span of the data.

See Also

Other implicit gaps handling: [count_gaps()](), [fill_gaps()](), [has_gaps()]()

Examples

```
scan_gaps(pedestrian)
```

---

slide   *Sliding window calculation*

Description

Questioning

The rolling window family will be deprecated in the future. Please consider using the slider package.

Rolling window with overlapping observations:

- slide() always returns a list.
- slide_lgl(), slide_int(), slide_dbl(), slide_chr() use the same arguments as slide(), but return vectors of the corresponding type.
- slide_dfr() & slide_dfc() return data frames using row-binding & column-binding.

Usage

```
slide(
  .x,
  .f,
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE
)

slide_dfr(
  .x,
  .f,
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE,
  .id = NULL
)

slide_dfc(
  .x,
  .f,
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE
)
```

Arguments

.x	An object to slide over.
.f	A function, formula, or vector (not necessarily atomic).

If a **function**, it is used as is.

If a **formula**, e.g. ~ .x + 2, it is converted to a function. There are three ways to refer to the arguments:

- For a single argument function, use .
- For a two argument function, use .x and .y
- For more arguments, use ..1, ..2, ..3 etc

This syntax allows you to create very compact anonymous functions.

If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position;

	use a list to index by position and name at different levels. If a component is not present, the value of .default will be returned.
...	Additional arguments passed on to the mapped function.
.size	An integer for window size. If positive, moving forward from left to right; if negative, moving backward (from right to left).
.step	A positive integer for calculating at every specified step instead of every single step.
.fill	A value to fill at the left/center/right of the data range depending on .align (NA by default). NULL means no filling.
.partial	if TRUE, partial sliding.
.align	Align index at the "**r**ight", "**c**entre"/"center", or "**l**eft" of the window. If .size is even for center alignment, "centre-right" & "centre-left" is needed.
.bind	If .x is a list, should .x be combined before applying .f? If .x is a list of data frames, row binding is carried out.
.id	Either a string or NULL. If a string, the output will contain a variable with that name, storing either the name (if .x is named) or the index (if .x is unnamed) of the input. If NULL, the default, no variable will be created. Only applies to _dfr variant.

Details

The slide() function attempts to tackle more general problems using the purrr-like syntax. For some specialist functions like mean and sum, you may like to check out for **RcppRoll** for faster performance.

slide() is intended to work with list (and column-wise data frame). To perform row-wise sliding window on data frame, please check out [pslide()](#).

- .partial = TRUE allows for partial sliding. Window contains observations outside of the vector will be treated as value of .fill, which will be passed to .f.

- .partial = FALSE restricts calculations to be done on complete sliding windows. Window contains observations outside of the vector will return the value .fill.

Value

if .fill != NULL, it always returns the same length as input.

See Also

- [future_slide](#) for parallel processing
- [tile](#) for tiling window without overlapping observations
- [stretch](#) for expanding more observations

Other sliding window functions: [slide2()](#)

Examples

```
x <- 1:5
lst <- list(x = x, y = 6:10, z = 11:15)
slide_dbl(x, mean, .size = 2)
slide_dbl(x, mean, .size = 2, align = "center")
slide_lgl(x, ~ mean(.) > 2, .size = 2)
slide(lst, ~., .size = 2)
```

---

slide2   *Sliding window calculation over multiple inputs simultaneously*

Description

Questioning

The rolling window family will be deprecated in the future. Please consider using the slider package.

Rolling window with overlapping observations:

- `slide2()` and `pslide()` always returns a list.
- `slide2_lgl()`, `slide2_int()`, `slide2_dbl()`, `slide2_chr()` use the same arguments as `slide2()`, but return vectors of the corresponding type.
- `slide2_dfr()` `slide2_dfc()` return data frames using row-binding & column-binding.

Usage

```
slide2(
  .x,
  .y,
  .f,
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE
)

slide2_dfr(
  .x,
  .y,
  .f,
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE,
  .id = NULL
)

slide2_dfc(
  .x,
  .y,
  .f,
  ...,
  .size = 1,
  .step = 1,
```

```
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE
)

pslide(
  .l,
  .f,
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE
)

pslide_dfr(
  .l,
  .f,
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE,
  .id = NULL
)

pslide_dfc(
  .l,
  .f,
  ...,
  .size = 1,
  .step = 1,
  .fill = NA,
  .partial = FALSE,
  .align = "right",
  .bind = FALSE
)
```

Arguments

`.x, .y`	Objects to slide over simultaneously.
`.f`	A function, formula, or vector (not necessarily atomic).
	If a **function**, it is used as is.
	If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:
	• For a single argument function, use `.`
	• For a two argument function, use `.x` and `.y`

- For more arguments, use `..1, ..2, ..3` etc

This syntax allows you to create very compact anonymous functions.

If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.

`...`	Additional arguments passed on to the mapped function.
`.size`	An integer for window size. If positive, moving forward from left to right; if negative, moving backward (from right to left).
`.step`	A positive integer for calculating at every specified step instead of every single step.
`.fill`	A value to fill at the left/center/right of the data range depending on `.align` (NA by default). NULL means no filling.
`.partial`	if TRUE, partial sliding.
`.align`	Align index at the "**r**ight", "**c**entre"/"center", or "**l**eft" of the window. If `.size` is even for center alignment, "centre-right" & "centre-left" is needed.
`.bind`	If `.x` is a list, should `.x` be combined before applying `.f`? If `.x` is a list of data frames, row binding is carried out.
`.id`	Either a string or NULL. If a string, the output will contain a variable with that name, storing either the name (if `.x` is named) or the index (if `.x` is unnamed) of the input. If NULL, the default, no variable will be created. Only applies to _dfr variant.
`.l`	A list of vectors, such as a data frame. The length of `.l` determines the number of arguments that `.f` will be called with. List names will be used if present.

See Also

- tile2 for tiling window without overlapping observations
- stretch2 for expanding more observations

Other sliding window functions: slide()

Examples

```
x <- 1:5
y <- 6:10
z <- 11:15
lst <- list(x = x, y = y, z = z)
df <- as.data.frame(lst)
slide2(x, y, sum, .size = 2)
slide2(lst, lst, ~., .size = 2)
slide2(df, df, ~., .size = 2)
pslide(lst, ~., .size = 1)
pslide(list(lst, lst), ~., .size = 2)

###
# row-wise sliding over data frame
###

if (!requireNamespace("tidyr", quietly = TRUE)) {
  stop("Please install the 'tidyr' package to run these following examples.")
```

```
}
library(tidyr)
library(dplyr)
my_df <- data.frame(
  group = rep(letters[1:2], each = 8),
  x = c(1:8, 8:1),
  y = 2 * c(1:8, 8:1) + rnorm(16),
  date = rep(as.Date("2016-06-01") + 0:7, 2)
)

slope <- function(...) {
  data <- list(...)
  fm <- lm(y ~ x, data = data)
  coef(fm)[[2]]
}

my_df %>%
  group_by(group) %>%
  nest() %>%
  mutate(slope = purrr::map(data, ~ pslide_dbl(., slope, .size = 2))) %>%
  unnest(slope)
## window over 2 months
pedestrian %>%
  filter(Sensor == "Southern Cross Station") %>%
  index_by(yrmth = yearmonth(Date_Time)) %>%
  nest(data = -yrmth) %>%
  ungroup() %>%
  mutate(ma = slide_dbl(data, ~ mean(.$Count), .size = 2, .bind = TRUE))
# row-oriented workflow
## Not run:
my_diag <- function(...) {
  data <- list(...)
  fit <- lm(Count ~ Time, data = data)
  tibble(fitted = fitted(fit), resid = residuals(fit))
}
pedestrian %>%
  filter_index("2015-01") %>%
  group_by_key() %>%
  nest() %>%
  mutate(diag = purrr::map(data, ~ pslide_dfr(., my_diag, .size = 48)))

## End(Not run)
```

---

slider   *Splits the input to a list according to the rolling window size.*

Description

Splits the input to a list according to the rolling window size.

Usage

```
slider(.x, .size = 1, .step = 1, .bind = FALSE)

pslider(..., .size = 1, .step = 1, .bind = FALSE)
```

Arguments

`.x`	An objects to be split.
`.size`	An integer for window size. If positive, moving forward from left to right; if negative, moving backward (from right to left).
`.step`	A positive integer for calculating at every specified step instead of every single step.
`.bind`	If `.x` is a list or data frame, the input will be flattened to a list of data frames.
`...`	Multiple objects to be split in parallel.

See Also

[partial_slider](#), [partial_pslider](#) for partial sliding

Examples

```
x <- 1:5
y <- 6:10
z <- 11:15
lst <- list(x = x, y = y, z = z)
df <- as.data.frame(lst)

slider(x, .size = 2)
slider(lst, .size = 2)
pslider(list(x, y), list(y))
slider(df, .size = 2)
pslider(df, df, .size = 2)
```

---

slide_tsibble	*Perform sliding windows on a tsibble by row*

Description

Questioning

Usage

```
slide_tsibble(.x, .size = 1, .step = 1, .id = ".id")
```

Arguments

`.x`	A tsibble.
`.size`	A positive integer for window size.
`.step`	A positive integer for calculating at every specified step instead of every single step.
`.id`	A character naming the new column `.id` containing the partition.

Rolling tsibble

> slide_tsibble(), tile_tsibble(), and stretch_tsibble() provide fast and shorthand for
> rolling over a tsibble by observations. That said, if the supplied tsibble has time gaps, these rolling
> helpers will ignore those gaps and proceed.
>
> They are useful for preparing the tsibble for time series cross validation. They all return a tsibble
> including a new column .id as part of the key. The output dimension will increase considerably
> with slide_tsibble() and stretch_tsibble(), which is likely to run out of memory when the
> data is large. Alternatively, you could construct cross validation using pslide() and pstretch()
> to avoid the memory issue.

See Also

> Other rolling tsibble: [stretch_tsibble](#)(), [tile_tsibble](#)()

Examples

```
harvest <- tsibble(
  year = rep(2010:2012, 2),
  fruit = rep(c("kiwi", "cherry"), each = 3),
  kilo = sample(1:10, size = 6),
  key = fruit, index = year
)
harvest %>%
  slide_tsibble(.size = 2)
```

---

stretch	*Stretching window calculation*

Description

> **Questioning**
>
> **The rolling window family will be deprecated in the future. Please consider using the [slider](#)**
> **package.**
>
> Fixing an initial window and expanding more observations:
>
> - stretch() always returns a list.
> - stretch_lgl(), stretch_int(), stretch_dbl(), stretch_chr() use the same arguments
>   as stretch(), but return vectors of the corresponding type.
> - stretch_dfr() stretch_dfc() return data frames using row-binding & column-binding.

Usage

```
stretch(
  .x,
  .f,
  ...,
  .step = 1,
  .init = 1,
  .fill = NA,
  .bind = FALSE,
```

```
    .size = deprecated()
)

stretch_dfr(
  .x,
  .f,
  ...,
  .step = 1,
  .init = 1,
  .fill = NA,
  .bind = FALSE,
  .id = NULL
)

stretch_dfc(.x, .f, ..., .step = 1, .init = 1, .fill = NA, .bind = FALSE)
```

Arguments

`.x`	An object to slide over.
`.f`	A function, formula, or vector (not necessarily atomic).
	If a **function**, it is used as is.
	If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:
	• For a single argument function, use `.`
	• For a two argument function, use `.x` and `.y`
	• For more arguments, use `..1`, `..2`, `..3` etc
	This syntax allows you to create very compact anonymous functions.
	If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to the mapped function.
`.step`	A positive integer for incremental step.
`.init`	A positive integer for an initial window size.
`.fill`	A value to fill at the left/center/right of the data range depending on `.align` (NA by default). NULL means no filling.
`.bind`	If `.x` is a list, should `.x` be combined before applying `.f`? If `.x` is a list of data frames, row binding is carried out.
`.size`	**Defunct** Please use `.step` instead.
`.id`	Either a string or NULL. If a string, the output will contain a variable with that name, storing either the name (if `.x` is named) or the index (if `.x` is unnamed) of the input. If NULL, the default, no variable will be created.
	Only applies to `_dfr` variant.

Value

if `.fill != NULL`, it always returns the same length as input.

See Also

- [future_stretch](#) for stretching window in parallel
- [slide](#) for sliding window with overlapping observations
- [tile](#) for tiling window without overlapping observations

Other stretching window functions: [stretch2()](#)

Examples

```
x <- 1:5
stretch_dbl(x, mean, .step = 2)
stretch_lgl(x, ~ mean(.) > 2, .step = 2)
lst <- list(x = x, y = 6:10, z = 11:15)
stretch(lst, ~., .step = 2, .fill = NULL)
```

---

stretch2   *Stretching window calculation over multiple simultaneously*

Description

Questioning

The rolling window family will be deprecated in the future. Please consider using the [slider](#) package.

Fixing an initial window and expanding more observations:

- stretch2() and pstretch() always returns a list.
- stretch2_lgl(), stretch2_int(), stretch2_dbl(), stretch2_chr() use the same arguments as stretch2(), but return vectors of the corresponding type.
- stretch2_dfr() stretch2_dfc() return data frames using row-binding & column-binding.

Usage

```
stretch2(.x, .y, .f, ..., .step = 1, .init = 1, .fill = NA, .bind = FALSE)

stretch2_dfr(
  .x,
  .y,
  .f,
  ...,
  .step = 1,
  .init = 1,
  .fill = NA,
  .bind = FALSE,
  .id = NULL
)

stretch2_dfc(.x, .y, .f, ..., .step = 1, .init = 1, .fill = NA, .bind = FALSE)

pstretch(.l, .f, ..., .step = 1, .init = 1, .fill = NA, .bind = FALSE)
```

```
pstretch_dfr(
  .l,
  .f,
  ...,
  .step = 1,
  .init = 1,
  .fill = NA,
  .bind = FALSE,
  .id = NULL
)

pstretch_dfc(.l, .f, ..., .step = 1, .init = 1, .fill = NA, .bind = FALSE)
```

Arguments

`.x`	Objects to slide over simultaneously.
`.y`	Objects to slide over simultaneously.
`.f`	A function, formula, or vector (not necessarily atomic).
	If a **function**, it is used as is.
	If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:
	• For a single argument function, use `.`
	• For a two argument function, use `.x` and `.y`
	• For more arguments, use `..1`, `..2`, `..3` etc
	This syntax allows you to create very compact anonymous functions.
	If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to the mapped function.
`.step`	A positive integer for calculating at every specified step instead of every single step.
`.init`	A positive integer for an initial window size.
`.fill`	A value to fill at the left/center/right of the data range depending on `.align` (NA by default). NULL means no filling.
`.bind`	If `.x` is a list, should `.x` be combined before applying `.f`? If `.x` is a list of data frames, row binding is carried out.
`.id`	Either a string or NULL. If a string, the output will contain a variable with that name, storing either the name (if `.x` is named) or the index (if `.x` is unnamed) of the input. If NULL, the default, no variable will be created.
	Only applies to _dfr variant.
`.l`	A list of vectors, such as a data frame. The length of `.l` determines the number of arguments that `.f` will be called with. List names will be used if present.

See Also

- [slide2](#) for sliding window with overlapping observations
- [tile2](#) for tiling window without overlapping observations

Other stretching window functions: [stretch()](#)

Examples

```
x <- 1:5
y <- 6:10
z <- 11:15
lst <- list(x = x, y = y, z = z)
df <- as.data.frame(lst)
stretch2(x, y, sum, .step = 2)
stretch2(lst, lst, ~., .step = 2)
stretch2(df, df, ~., .step = 2)
pstretch(lst, sum, .step = 1)
pstretch(list(lst, lst), ~., .step = 2)

###
# row-wise stretching over data frame
###

x <- as.Date("2017-01-01") + 0:364
df <- data.frame(x = x, y = seq_along(x))

tibble(
  data = pstretch(df, function(...) as_tibble(list(...)), .init = 10)
)
```

---

stretcher	*Split the input to a list according to the stretching window size.*

Description

Split the input to a list according to the stretching window size.

Usage

```
stretcher(.x, .step = 1, .init = 1, .bind = FALSE)

pstretcher(..., .step = 1, .init = 1, .bind = FALSE)
```

Arguments

.x	An objects to be split.
.step	A positive integer for incremental step.
.init	A positive integer for an initial window size.
.bind	If .x is a list, should .x be combined before applying .f? If .x is a list of data frames, row binding is carried out.
...	Multiple objects to be split in parallel.

Examples

```
x <- 1:5
y <- 6:10
z <- 11:15
lst <- list(x = x, y = y, z = z)
df <- as.data.frame(lst)
```

```
stretcher(x, .step = 2)
stretcher(lst, .step = 2)
stretcher(df, .step = 2)
pstretcher(df, df, .step = 2)
```

---

stretch_tsibble   *Perform stretching windows on a tsibble by row*

Description

Questioning

Usage

```
stretch_tsibble(.x, .step = 1, .init = 1, .id = ".id")
```

Arguments

.x	A tsibble.
.step	A positive integer for incremental step.
.init	A positive integer for an initial window size.
.id	A character naming the new column .id containing the partition.

Rolling tsibble

slide_tsibble(), tile_tsibble(), and stretch_tsibble() provide fast and shorthand for rolling over a tsibble by observations. That said, if the supplied tsibble has time gaps, these rolling helpers will ignore those gaps and proceed.

They are useful for preparing the tsibble for time series cross validation. They all return a tsibble including a new column .id as part of the key. The output dimension will increase considerably with slide_tsibble() and stretch_tsibble(), which is likely to run out of memory when the data is large. Alternatively, you could construct cross validation using pslide() and pstretch() to avoid the memory issue.

See Also

Other rolling tsibble: [slide_tsibble](), [tile_tsibble]()

Examples

```
harvest <- tsibble(
  year = rep(2010:2012, 2),
  fruit = rep(c("kiwi", "cherry"), each = 3),
  kilo = sample(1:10, size = 6),
  key = fruit, index = year
)
harvest %>%
  stretch_tsibble()
```

---

tile   *Tiling window calculation*

Description

Questioning

The rolling window family will be deprecated in the future. Please consider using the slider package.

Tiling window without overlapping observations:

- `tile()` always returns a list.
- `tile_lgl()`, `tile_int()`, `tile_dbl()`, `tile_chr()` use the same arguments as `tile()`, but return vectors of the corresponding type.
- `tile_dfr()` `tile_dfc()` return data frames using row-binding & column-binding.

Usage

```
tile(.x, .f, ..., .size = 1, .bind = FALSE)

tile_dfr(.x, .f, ..., .size = 1, .bind = FALSE, .id = NULL)

tile_dfc(.x, .f, ..., .size = 1, .bind = FALSE)
```

Arguments

`.x`	An object to slide over.
`.f`	A function, formula, or vector (not necessarily atomic).
	If a **function**, it is used as is.
	If a **formula**, e.g. `~ .x + 2`, it is converted to a function. There are three ways to refer to the arguments:
	• For a single argument function, use `.`
	• For a two argument function, use `.x` and `.y`
	• For more arguments, use `..1`, `..2`, `..3` etc
	This syntax allows you to create very compact anonymous functions.
	If **character vector**, **numeric vector**, or **list**, it is converted to an extractor function. Character vectors index by name and numeric vectors index by position; use a list to index by position and name at different levels. If a component is not present, the value of `.default` will be returned.
`...`	Additional arguments passed on to the mapped function.
`.size`	An integer for window size. If positive, moving forward from left to right; if negative, moving backward (from right to left).
`.bind`	If `.x` is a list, should `.x` be combined before applying `.f`? If `.x` is a list of data frames, row binding is carried out.
`.id`	Either a string or `NULL`. If a string, the output will contain a variable with that name, storing either the name (if `.x` is named) or the index (if `.x` is unnamed) of the input. If `NULL`, the default, no variable will be created.
	Only applies to `_dfr` variant.

See Also

- [future_tile](#) for tiling window in parallel
- [slide](#) for sliding window with overlapping observations
- [stretch](#) for expanding more observations

Other tiling window functions: [`tile2()`](#)

Examples

```
x <- 1:5
lst <- list(x = x, y = 6:10, z = 11:15)
tile_dbl(x, mean, .size = 2)
tile_lgl(x, ~ mean(.) > 2, .size = 2)
tile(lst, ~., .size = 2)
```

---

tile2 *Tiling window calculation over multiple inputs simultaneously*

Description

Questioning

The rolling window family will be deprecated in the future. Please consider using the [slider](#) package.

Tiling window without overlapping observations:

- `tile2()` and `ptile()` always returns a list.
- `tile2_lgl()`, `tile2_int()`, `tile2_dbl()`, `tile2_chr()` use the same arguments as `tile2()`, but return vectors of the corresponding type.
- `tile2_dfr()` `tile2_dfc()` return data frames using row-binding & column-binding.

Usage

```
tile2(.x, .y, .f, ..., .size = 1, .bind = FALSE)

tile2_dfr(.x, .y, .f, ..., .size = 1, .bind = FALSE, .id = NULL)

tile2_dfc(.x, .y, .f, ..., .size = 1, .bind = FALSE)

ptile(.l, .f, ..., .size = 1, .bind = FALSE)

ptile_dfr(.l, .f, ..., .size = 1, .bind = FALSE, .id = NULL)

ptile_dfc(.l, .f, ..., .size = 1, .bind = FALSE)
```

Arguments

`.x`	Objects to slide over simultaneously.
`.y`	Objects to slide over simultaneously.

.f   A function, formula, or vector (not necessarily atomic).

                            If a **function**, it is used as is.

                            If a **formula**, e.g. ~ .x + 2, it is converted to a function. There are three ways to
                            refer to the arguments:

                               • For a single argument function, use .

                               • For a two argument function, use .x and .y

                               • For more arguments, use ..1, ..2, ..3 etc

                            This syntax allows you to create very compact anonymous functions.

                            If **character vector**, **numeric vector**, or **list**, it is converted to an extractor func-
                            tion. Character vectors index by name and numeric vectors index by position;
                            use a list to index by position and name at different levels. If a component is not
                            present, the value of .default will be returned.

...   Additional arguments passed on to the mapped function.

.size   An integer for window size. If positive, moving forward from left to right; if
                            negative, moving backward (from right to left).

.bind   If .x is a list, should .x be combined before applying .f? If .x is a list of data
                            frames, row binding is carried out.

.id   Either a string or NULL. If a string, the output will contain a variable with that
                            name, storing either the name (if .x is named) or the index (if .x is unnamed)
                            of the input. If NULL, the default, no variable will be created.

                            Only applies to _dfr variant.

.l   A list of vectors, such as a data frame. The length of .l determines the number
                            of arguments that .f will be called with. List names will be used if present.

See Also

   • [slide2](#) for sliding window with overlapping observations

   • [stretch2](#) for expanding more observations

Other tiling window functions: [tile()](#)

Examples

```
x <- 1:5
y <- 6:10
z <- 11:15
lst <- list(x = x, y = y, z = z)
df <- as.data.frame(lst)
tile2(x, y, sum, .size = 2)
tile2(lst, lst, ~., .size = 2)
tile2(df, df, ~., .size = 2)
ptile(lst, sum, .size = 1)
ptile(list(lst, lst), ~., .size = 2)
```

| tiler | *Splits the input to a list according to the tiling window size.* |

Description

Splits the input to a list according to the tiling window size.

Usage

```
tiler(.x, .size = 1, .bind = FALSE)

ptiler(..., .size = 1, .bind = FALSE)
```

Arguments

.x	An objects to be split.
.size	An integer for window size. If positive, moving forward from left to right; if negative, moving backward (from right to left).
.bind	If .x is a list or data frame, the input will be flattened to a list of data frames.
...	Multiple objects to be split in parallel.

Examples

```
x <- 1:5
y <- 6:10
z <- 11:15
lst <- list(x = x, y = y, z = z)
df <- as.data.frame(lst)

tiler(x, .size = 2)
tiler(lst, .size = 2)
ptiler(lst, .size = 2)
ptiler(list(x, y), list(y))
ptiler(df, .size = 2)
ptiler(df, df, .size = 2)
```

| tile_tsibble | *Perform tiling windows on a tsibble by row* |

Description

Questioning

Usage

```
tile_tsibble(.x, .size = 1, .id = ".id")
```

Arguments

`.x`	A tsibble.
`.size`	A positive integer for window size.
`.id`	A character naming the new column `.id` containing the partition.

Rolling tsibble

`slide_tsibble()`, `tile_tsibble()`, and `stretch_tsibble()` provide fast and shorthand for rolling over a tsibble by observations. That said, if the supplied tsibble has time gaps, these rolling helpers will ignore those gaps and proceed.

They are useful for preparing the tsibble for time series cross validation. They all return a tsibble including a new column `.id` as part of the key. The output dimension will increase considerably with `slide_tsibble()` and `stretch_tsibble()`, which is likely to run out of memory when the data is large. Alternatively, you could construct cross validation using `pslide()` and `pstretch()` to avoid the memory issue.

See Also

Other rolling tsibble: [slide_tsibble](), [stretch_tsibble]()

Examples

```
harvest <- tsibble(
  year = rep(2010:2012, 2),
  fruit = rep(c("kiwi", "cherry"), each = 3),
  kilo = sample(1:10, size = 6),
  key = fruit, index = year
)
harvest %>%
  tile_tsibble(.size = 2)
```

---

time_in	*If time falls in the ranges using compact expressions*

Description

This function respects time zone and encourages compact expressions.

Usage

```
time_in(x, ...)
```

Arguments

x	A vector of time index, such as classes `POSIXct`, `Date`, `yearweek`, `yearmonth`, `yearquarter`, `hms`/`difftime`, and `numeric`.
...	Formulas that specify start and end periods (inclusive), or strings.

- `~ end` or `. ~ end`: from the very beginning to a specified ending period.
- `start ~ end`: from specified beginning to ending periods.
- `start ~ .`: from a specified beginning to the very end of the data. Supported index type: `POSIXct` (to seconds), `Date`, `yearweek`, `yearmonth`/`yearmon`, `yearquarter`/`yearqtr`, `hms`/`difftime` & `numeric`.

Value

logical vector

System Time Zone ("Europe/London")

There is a known issue of an extra hour gained for a machine setting time zone to "Europe/London", regardless of the time zone associated with the POSIXct inputs. It relates to *anytime* and *Boost*. Use `Sys.timezone()` to check if the system time zone is "Europe/London". It would be recommended to change the global environment "TZ" to other equivalent names: GB, GB-Eire, Europe/Belfast, Europe/Guernsey, Europe/Isle_of_Man and Europe/Jersey as documented in `?Sys.timezone()`, using `Sys.setenv(TZ = "GB")` for example.

See Also

filter_index for filtering tsibble

Examples

```
x <- unique(pedestrian$Date_Time)
lgl <- time_in(x, ~"2015-02", "2015-08" ~ "2015-09", "2015-12" ~ "2016-02")
lgl[1:10]
# more specific
lgl2 <- time_in(x, "2015-03-23 10" ~ "2015-10-31 12")
lgl2[1:10]

library(dplyr)
pedestrian %>%
  filter(time_in(Date_Time, "2015-03-23 10" ~ "2015-10-31 12"))
pedestrian %>%
  filter(time_in(Date_Time, "2015")) %>%
  mutate(Season = ifelse(
    time_in(Date_Time, "2015-03" ~ "2015-08"),
    "Autumn-Winter", "Spring-Summer"
  ))
```

---

tourism   *Australian domestic overnight trips*

Description

A dataset containing the quarterly overnight trips from 1998 Q1 to 2016 Q4 across Australia.

Usage

```
tourism
```

Format

A tsibble with 23,408 rows and 5 variables:

- **Quarter**: Year quarter (index)

- **Region**: The tourism regions are formed through the aggregation of Statistical Local Areas (SLAs) which are defined by the various State and Territory tourism authorities according to their research and marketing needs
- **State**: States and territories of Australia
- **Purpose**: Stopover purpose of visit:
    - "Holiday"
    - "Visiting friends and relatives"
    - "Business"
    - "Other reason"
- **Trips**: Overnight trips in thousands

References

[Tourism Research Australia](#)

Examples

```
library(dplyr)
data(tourism)
# Total trips over geographical regions
tourism %>%
  group_by(Region, State) %>%
  summarise(Total_Trips = sum(Trips))
```

---

tsibble	*Create a tsibble object*

Description

Stable

Usage

```
tsibble(..., key = NULL, index, regular = TRUE, .drop = TRUE)
```

Arguments

`...`	A set of name-value pairs.
`key`	Unquoted variable(s) that uniquely determine time indices. `NULL` for empty key, and `c()` for multiple variables. It works with tidy selector (e.g. `dplyr::starts_with()`).
`index`	A bare (or unquoted) variable to specify the time index variable.
`regular`	Regular time interval (`TRUE`) or irregular (`FALSE`). The interval is determined by the greatest common divisor of index column, if `TRUE`.
`.drop`	If `TRUE`, empty key groups are dropped.

Details

A tsibble is sorted by its key first and index.

Value

A tsibble object.

Index

An extensive range of indices are supported by tsibble: native time classes in R (such as `Date`, `POSIXct`, and `difftime`) and tsibble's new additions (such as [yearweek](), [yearmonth](), and [year-quarter]()). Some commonly-used classes have built-in support too, including ordered, `hms::hms`, `zoo::yearmon`, `zoo::yearqtr`, and `nanotime`.

For a `tbl_ts` of regular interval, a choice of index representation has to be made. For example, a monthly data should correspond to time index created by [yearmonth]() or `zoo::yearmon`, instead of `Date` or `POSIXct`. Because months in a year ensures the regularity, 12 months every year. However, if using `Date`, a month containing days ranges from 28 to 31 days, which results in irregular time space. This is also applicable to year-week and year-quarter.

Tsibble supports arbitrary index classes, as long as they can be ordered from past to future. To support a custom class, you need to define [index_valid()]() for the class and calculate the interval through [interval_pull()]().

Key

Key variable(s) together with the index uniquely identifies each record:

- Empty: an implicit variable. `NULL` resulting in a univariate time series.
- A single variable: For example, `data(pedestrian)` use the bare `Sensor` as the key.
- Multiple variables: For example, Declare `key = c(Region, State, Purpose)` for `data(tourism)`. Key can be created in conjunction with tidy selectors like `starts_with()`.

Interval

The [interval]() function returns the interval associated with the tsibble.

- Regular: the value and its time unit including "nanosecond", "microsecond", "millisecond", "second", "minute", "hour", "day", "week", "month", "quarter", "year". An unrecognisable time interval is labelled as "unit".
- Irregular: `as_tsibble(regular = FALSE)` gives the irregular tsibble. It is marked with !.
- Unknown: Not determined (?), if it's an empty tsibble, or one entry for each key variable.

An interval is obtained based on the corresponding index representation:

- integerish numerics between 1582 and 2499: "year" (Y). Note the year of 1582 saw the beginning of the Gregorian Calendar switch.
- yearquarter/yearqtr: "quarter" (Q)
- yearmonth/yearmon: "month" (M)
- yearweek: "week" (W)
- `Date`: "day" (D)
- difftime: "week" (W), "day" (D), "hour" (h), "minute" (m), "second" (s)
- POSIXt/hms: "hour" (h), "minute" (m), "second" (s), "millisecond" (us), "microsecond" (ms)
- nanotime: "nanosecond" (ns)
- other numerics `&ordered` (ordered factor): "unit" When the interval cannot be obtained due to the mismatched index format, an error is issued.

The interval is invariant to subsetting, such as filter(), slice(), and [.tbl_ts. But if the result is an empty tsibble, the interval is always unknown. When joining a tsibble with other data sources and aggregating to different time scales, the interval gets re-calculated.

See Also

build_tsibble

Examples

```
# create a tsibble w/o a key
tsibble(
  date = as.Date("2017-01-01") + 0:9,
  value = rnorm(10)
)

# create a tsibble with a single variable for key
tsibble(
  qtr = rep(yearquarter("2010 Q1") + 0:9, 3),
  group = rep(c("x", "y", "z"), each = 10),
  value = rnorm(30),
  key = group
)

# create a tsibble with multiple variables for key
tsibble(
  mth = rep(yearmonth("2010 Jan") + 0:8, each = 3),
  xyz = rep(c("x", "y", "z"), each = 9),
  abc = rep(letters[1:3], times = 9),
  value = rnorm(27),
  key = c(xyz, abc)
)

# create a tsibble containing "key" and "index" as column names
tsibble(!!!list(
  index = rep(yearquarter("2010 Q1") + 0:9, 3),
  key = rep(c("x", "y", "z"), each = 10),
  value = rnorm(30)),
  key = key, index = index
)
```

---

tsibble-tidyverse   *Tidyverse methods for tsibble*

Description

- arrange(): if not arranging key and index in past-to-future order, a warning is likely to be issued.
- slice(): if row numbers are not in ascending order, a warning is likely to be issued.
- select(): keeps the variables you mention as well as the index.
- transmute(): keeps the variable you operate on, as well as the index and key.
- summarise() reduces a sequence of values over time instead of a single summary, as well as dropping empty keys/groups.

Usage

```
## S3 method for class 'tbl_ts'
arrange(.data, ...)

## S3 method for class 'tbl_ts'
filter(.data, ..., .preserve = FALSE)

## S3 method for class 'tbl_ts'
slice(.data, ..., .preserve = FALSE)

## S3 method for class 'tbl_ts'
select(.data, ...)

## S3 method for class 'tbl_ts'
rename(.data, ...)

## S3 method for class 'tbl_ts'
mutate(.data, ...)

## S3 method for class 'tbl_ts'
transmute(.data, ...)

## S3 method for class 'tbl_ts'
summarise(.data, ...)
```

Arguments

`.data`	A `tbl_ts`.
`...`	Same arguments accepted as its tidyverse generic.
`.preserve`	when `FALSE` (the default), the grouping structure is recalculated based on the resulting data, otherwise it is kept as is.
`data`	A tbl. All main verbs are S3 generics and provide methods for `tbl_df()`, `dtplyr::tbl_dt()` and `dbplyr::tbl_dbi()`.

Details

Column-wise verbs, including `select()`, `transmute()`, `summarise()`, `mutate()` & `transmute()`, keep the time context hanging around. That is, the index variable cannot be dropped for a tsibble. If any key variable is changed, it will validate whether it's a tsibble internally. Use `as_tibble()` to leave off the time context.

Examples

```
library(dplyr, warn.conflicts = FALSE)
# Sum over sensors
pedestrian %>%
  index_by() %>%
  summarise(Total = sum(Count))
# shortcut
pedestrian %>%
  summarise(Total = sum(Count))
# Back to tibble
```

```
pedestrian %>%
  as_tibble() %>%
  summarise(Total = sum(Count))
library(tidyr)
# reshaping examples from tidyr
stocks <- tsibble(
  time = as.Date("2009-01-01") + 0:9,
  X = rnorm(10, 0, 1),
  Y = rnorm(10, 0, 2),
  Z = rnorm(10, 0, 4)
)
(stocksm <- stocks %>% gather(stock, price, -time))
stocksm %>% spread(stock, price)
```

---

units_since   *Time units since Unix Epoch*

Description

Questioning

Usage

```
units_since(x)
```

Arguments

x   An object of POSIXct, Date, yearweek, yearmonth, yearquarter.

Details

origin:

- POSIXct: 1970-01-01 00:00:00
- Date: 1970-01-01
- yearweek: 1970 W01 (i.e. 1969-12-29)
- yearmonth: 1970 Jan
- yearquarter: 1970 Qtr1

Examples

```
units_since(x = yearmonth(2012 + (0:11) / 12))
```

---

update_tsibble *Update key and index for a tsibble*

Description

Update key and index for a tsibble

Usage

```
update_tsibble(
  x,
  key,
  index,
  regular = is_regular(x),
  validate = TRUE,
  .drop = key_drop_default(x)
)
```

Arguments

x	A tsibble.
key	Unquoted variable(s) that uniquely determine time indices. NULL for empty key, and c() for multiple variables. It works with tidy selector (e.g. `dplyr::starts_with()`).
index	A bare (or unquoted) variable to specify the time index variable.
regular	Regular time interval (TRUE) or irregular (FALSE). The interval is determined by the greatest common divisor of index column, if TRUE.
validate	TRUE suggests to verify that each key or each combination of key variables leads to unique time indices (i.e. a valid tsibble). If you are sure that it's a valid input, specify FALSE to skip the checks.
.drop	If TRUE, empty key groups are dropped.

Details

Unspecified arguments will inherit the attributes from x.

Examples

```
# update index
library(dplyr)
pedestrian %>%
  group_by_key() %>%
  mutate(Hour_Since = Date_Time - min(Date_Time)) %>%
  update_tsibble(index = Hour_Since)

# update key: drop the variable "State" from the key
tourism %>%
  update_tsibble(key = c(Purpose, Region))
```

---

yearweek   *Represent year-week (ISO) starting on Monday, year-month or year-quarter objects*

Description

Stable

Create or coerce using yearweek(), yearmonth(), or yearquarter()

Usage

```
yearweek(x)

is_53weeks(year)

yearmonth(x)

yearquarter(x)
```

Arguments

x   Other object.

year   A vector of years.

Value

Year-week (yearweek), year-month (yearmonth) or year-quarter (yearquarter) objects.

TRUE/FALSE if the year has 53 ISO weeks.

Index functions

The tsibble yearmonth() and yearquarter() function respects time zones of the input x, contrasting to their zoo counterparts.

See Also

interval_pull

Examples

```
# coerce POSIXct/Dates to yearweek, yearmonth, yearquarter
x <- seq(as.Date("2016-01-01"), as.Date("2016-12-31"), by = "1 month")
yearweek(x)
yearmonth(x)
yearmonth(yearweek(x))
yearquarter(x)

# coerce yearmonths to yearquarter
y <- yearmonth(x)
yearquarter(y)

# parse characters
```

```
yearweek(c("2018 W01", "2018 Wk01", "2018 Week 1"))
yearmonth(c("2018 Jan", "2018-01", "2018 January"))
yearquarter(c("2018 Q1", "2018 Qtr1", "2018 Quarter 1"))

# seq() and binary operaters
wk1 <- yearweek("2017-11-01")
wk2 <- yearweek("2018-04-29")
seq(from = wk1, to = wk2, by = 2) # by two weeks
wk1 + 0:9
mth <- yearmonth("2017-11")
seq(mth, length.out = 5, by = 1) # by 1 month
mth + 0:9
seq(yearquarter(mth), length.out = 5, by = 1) # by 1 quarter

# different formats
format(c(wk1, wk2), format = "%V/%Y")
format(y, format = "%y %m")
format(yearquarter(mth), format = "%y Qtr%q")
is_53weeks(2015:2016)
```

Index