Package 'tsibble'

February 18, 2019

Type Package

Title Tidy Temporal Data Frames and Tools

Version 0.7.0

Description Provides a 'tbl_ts' class (the 'tsibble') to store and manage temporal data in a data-centric format, which is built on top of the 'tibble'. The 'tsibble' aims at easily manipulating and analysing temporal data, including counting and filling in time gaps, aggregate over calendar periods, performing rolling window calculations, and etc.

License GPL-3

URL https://tsibble.tidyverts.org

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

Depends R (>= 3.1.3)

Imports anytime (>= 0.3.1), dplyr (>= 0.8.0), lubridate, purrr (>= 0.2.3), Rcpp (>= 0.12.3), rlang (>= 0.2.0), tibble (>= 2.0.1), tidyr, tidyselect

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

LinkingTo Rcpp (>= 0.12.0)

VignetteBuilder knitr

ByteCompile true

Encoding UTF-8

Language en-GB

LazyData true

RoxygenNote 6.1.1

Collate 'RcppExports.R' 'as-tsibble.R' 'bind.R' 'data.R' 'gaps.R' 'deprecated.R' 'dplyr-groups.R' 'dplyr-verbs.R' 'dplyr-join.R' 'error.R' 'filter-index.R' 'format.R' 'holiday.R' 'index-by.R' 'index-valid.R' 'interval.R' 'key-group.R' 'new-data.R'

'period.R' 'reexports.R' 'slide.R' 'stretch.R' 'subset.R'
'tidyr-verbs.R' 'tile.R' 'time-wise.R' 'ts2tsibble.R'
'tsibble-pkg.R' 'tsibble2ts.R' 'type-sum.R' 'update.R'
'utils-format.R' 'utils.R' 'zzz.R'

NeedsCompilation yes

Author Earo Wang [aut, cre] (<https://orcid.org/0000-0001-6448-5260>),
Di Cook [aut, ths] (<https://orcid.org/0000-0002-3813-7155>),
Rob Hyndman [aut, ths] (<https://orcid.org/0000-0002-2140-5352>),
Mitchell O'Hara-Wild [ctb]

Maintainer Earo Wang <earo.wang@gmail.com>

Repository CRAN

Date/Publication 2019-02-18 10:00:02 UTC

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

The time indices are preserved as the essential data component of the tsibble, instead of implicit attribute (for example, the `tsp` attribute in a `ts` object). A few index classes, such as `Date`, `POSIXct`, and `difftime`, forms the basis of the tsibble, with new additions yearweek, yearmonth, and yearquarter representing year-week, year-month, and year-quarter respectively. Any arbitrary index class are also supported, including `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 contains days ranging from 28 to 31 days, which results in irregular time space. This is also applicable to year-week and year-quarter.

Since the **tibble** that underlies the **tsibble** only accepts a 1d atomic vector or a list, a `tbl_ts` doesn't accept `POSIXlt` and `timeDate` columns.

Key

Key variable(s) together with the index uniquely identifies each record, which can be created via the id function as identifiers:

- Empty: an implicit variable `id()` resulting in a univariate time series.
- One or more variables: explicit variables. For example, `data(pedestrian)` and `data(tourism)` use the `id(Sensor)` & `id(Region, State, Purpose)` as the key respectively.

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: if there is only one entry for each key variable, the interval cannot be determined (`?`).

An interval is obtained based on the corresponding index representation:

- integer/numeric: either "unit" or "year"
- `yearquarter`/`yearqtr`: "quarter"
- `yearmonth`/`yearmon`: "month"
- `yearweek`: "week"
- `Date`: "day"
- `POSIXct`: "hour", "minute", "second", "millisecond", "microsecond"
- `nanotime`: "nanosecond"

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> (0000-0001-6448-5260)

Authors:

- Di Cook (0000-0002-3813-7155) [thesis advisor]
- Rob Hyndman (0000-0002-2140-5352) [thesis advisor]

Other contributors:

- Mitchell O'Hara-Wild [contributor]

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 = id(group)
)
```

---

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

Description

Coerce a tsibble to a time series

Usage

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

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 replaces 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

Coerce to a tsibble object

Usage

```
as_tsibble(x, ...)

## S3 method for class 'tbl_df'
as_tsibble(x, key = id(), index, regular = TRUE,
  validate = TRUE, ...)

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

## S3 method for class 'list'
as_tsibble(x, key = id(), index, regular = TRUE,
  validate = TRUE, ...)

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

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

Arguments

x	Other objects to be coerced to a tsibble (tbl_ts).
...	Other arguments passed on to individual methods.
key	Variable(s) that define unique time indices, used in conjunction with the helper id(). If a univariate time series (without an explicit key), simply call id().
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.
tz	Time zone. May be useful when a ts object is more frequent than daily.
gather	TRUE gives a "long" data form, otherwise as "wide" as x.

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)
# specify the index var
as_tsibble(tbl1, index = date)

# coerce tibble to tsibble with one 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 = id(group))
as_tsibble(tbl2, key = id(group), index = mth)

# 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, gather = 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, index, index2, ordered = NULL, regular = TRUE,
  interval = NULL, validate = TRUE)
```

Arguments

x	A data.frame, tbl_df, tbl_ts, or other tabular objects.
key	Variable(s) that define unique time indices, used in conjunction with the helper id(). If a univariate time series (without an explicit key), simply call id().
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 also skips the ordering but gives a warning instead.
regular	Regular time interval (TRUE) or irregular (FALSE). The interval is determined by the greatest common divisor of index column, if TRUE.
interval	NULL computes the interval. Use the specified interval via new_interval() as is, if an class of interval is supplied.
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.

Examples

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

---

count_gaps *Count implicit gaps*

Description

Count implicit gaps

Usage

```
count_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.
...	Other arguments passed on to individual methods.

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)
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

Lagged differences

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)
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

Turn implicit missing values into explicit missing values

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

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

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

# use fill() to fill `NA` by previous/next entry
full_harvest %>%
  group_by(fruit) %>%
  fill(kilo, .direction = "down")

# 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(fruit) %>%
  fill_gaps(kilo = sum(kilo))

# leaves existing `NA` untouched ----
harvest[2, 3] <- NA
harvest %>%
  group_by(fruit) %>%
```

```
  fill_gaps(kilo = sum(kilo, na.rm = TRUE))

# replace NA ----
pedestrian %>%
  group_by(Sensor) %>%
  fill_gaps(Count = as.integer(median(Count)))
```

`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

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 %>%
  nest(-Sensor) %>%
  mutate(diag = future_map(data, ~ future_pslide_dfr(., my_diag, .size = 48)))

## End(Not run)
```

`future_stretch()` *Stretching window in parallel*

Description

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

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

Group by key variables

Usage

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

Arguments

.data	A tbl_ts object.
...	Ignored.
.drop	When .drop = TRUE, empty groups are dropped.

Examples

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

guess_frequency *Guess a time frequency from other index objects*

Description

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 nearest ceiling time resolution. 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, ...)
```

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.
...	Other arguments passed on to individual methods.

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 = id(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 and collapse by time index*

Description

`index_by()` is the counterpart of `group_by()` in temporal context, but it only groups the time index. It adds a new column and then group it. The following operation is applied to each group 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()` or `index_by()` with no arguments to remove the index grouping vars.

Usage

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

Arguments

.data	A tbl_ts.
...	A single name-value pair of expression: a new index on LHS and the current index on RHS. Or an existing variable to be used as index. 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: sub-daily aggregation • other index functions from other packages

Details

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

Examples

```
# Monthly counts across sensors ----
monthly_ped <- pedestrian %>%
  group_by(Sensor) %>%
  index_by(Year_Month = yearmonth(Date_Time)) %>%
  summarise(
    Max_Count = max(Count),
    Min_Count = min(Count)
  )
monthly_ped
index(monthly_ped)

# Using existing variable ----
pedestrian %>%
  group_by(Sensor) %>%
  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(Sensor) %>%
  index_by(Date_Time4 = lubridate::floor_date(Date_Time, "4 hour")) %>%
  summarise(Total_Count = sum(Count))

# Annual trips by Region and State ----
tourism %>%
  index_by(Year = lubridate::year(Quarter)) %>%
```

```
  group_by(Region, State) %>%
  summarise(Total = sum(Trips))
```

`index_valid` *Add custom index support for a tsibble*

Description

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

pull_interval 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)
```

---

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

Description

- `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 = id(), index)

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

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

Arguments

data	A data frame for creating a tsibble.
key	Variable(s) that define unique time indices, used in conjunction with the helper `id()`. If a univariate time series (without an explicit key), simply call `id()`.
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 = id(fruit), index = year)
are_duplicated(harvest, key = id(fruit), index = year)
are_duplicated(harvest, key = id(fruit), index = year, from_last = TRUE)
duplicates(harvest, key = id(fruit), index = year)
```

---

is_tsibble *If the object is a tsibble*

Description

If the object is a tsibble

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

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 = id(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_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

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

pull_interval *Pull time interval from a vector*

Description

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

Usage

```
pull_interval(x)
```

Arguments

`x`	A vector of `POSIXt`, `Date`, `yearmonth`, `yearquarter`, `difftime`, `hms`, `ordered`, `integer`, `numeric`.

Details

`index_valid()` and `pull_interval()` make a tsibble extensible to support custom time index.

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)
pull_interval(x)
```

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.
...	Other arguments passed on to individual methods.

See Also

Other implicit gaps handling: count_gaps, fill_gaps, has_gaps

Examples

```
scan_gaps(pedestrian)
```

---

slide *Sliding window calculation*

Description

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

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
###

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 %>%
  nest(-group) %>%
  mutate(slope = purrr::map(data, ~ pslide_dbl(., slope, .size = 2))) %>%
  unnest()
## window over 2 months
pedestrian %>%
  filter(Sensor == "Southern Cross Station") %>%
  index_by(yrmth = yearmonth(Date_Time)) %>%
  nest(-yrmth) %>%
  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") %>%
  nest(-Sensor) %>%
  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

Perform sliding windows on a tsibble by row

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 new column .id as part of the key. The output dimension will increase considerably with slide_tsibble() and stretch_tsibble().

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 = id(fruit), index = year
)
harvest %>%
  slide_tsibble(.size = 2)
```

---

stretch *Stretching window calculation*

Description

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)

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.
`.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

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

Perform stretching windows on a tsibble by row

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 new column `.id` as part of the key. The output dimension will increase considerably with `slide_tsibble()` and `stretch_tsibble()`.

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 = id(fruit), index = year
)
harvest %>%
  stretch_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.
- `unnest()` requires argument `key = id()` to get back to a tsibble.

Usage

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

## S3 method for class 'grouped_ts'
arrange(.data, ..., .by_group = FALSE)

## 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 'grouped_ts'
select(.data, ...)

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

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

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

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

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

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

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

## S3 method for class 'tbl_ts'
group_by(.data, ..., add = FALSE, .drop = FALSE)

## S3 method for class 'grouped_ts'
ungroup(x, ...)

## S3 method for class 'tbl_ts'
left_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...)

## S3 method for class 'tbl_ts'
right_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...)

## S3 method for class 'tbl_ts'
inner_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...)

## S3 method for class 'tbl_ts'
full_join(x, y, by = NULL, copy = FALSE,
  suffix = c(".x", ".y"), ...)

## S3 method for class 'tbl_ts'
semi_join(x, y, by = NULL, copy = FALSE, ...)

## S3 method for class 'tbl_ts'
anti_join(x, y, by = NULL, copy = FALSE, ...)

## S3 method for class 'tbl_ts'
gather(data, key = "key", value = "value", ...,
  na.rm = FALSE, convert = FALSE, factor_key = FALSE)

## S3 method for class 'tbl_ts'
spread(data, key, value, fill = NA, convert = FALSE,
  drop = TRUE, sep = NULL)

## S3 method for class 'tbl_ts'
nest(data, ..., .key = "data")
```

```
## S3 method for class 'lst_ts'
unnest(data, ..., key = id(), .drop = NA,
  .id = NULL, .sep = NULL, .preserve = NULL)

## S3 method for class 'tbl_ts'
unnest(data, ..., key = id(), .drop = NA,
  .id = NULL, .sep = NULL, .preserve = NULL)

## S3 method for class 'tbl_ts'
fill(data, ..., .direction = c("down", "up"))

## S3 method for class 'grouped_ts'
fill(data, ..., .direction = c("down", "up"))
```

Arguments

.data	A tbl_ts.
...	same arguments accepted as its dplyr generic.
.by_group	If TRUE, will sort first by grouping variable. Applies to grouped data frames only.
.preserve	Optionally, list-columns to preserve in the output. These will be duplicated in the same way as atomic vectors. This has dplyr::select semantics so you can preserve multiple variables with .preserve = c(x, y) or .preserve = starts_with("list").
add	When add = FALSE, the default, group_by() will override existing groups. To add to the existing groups, use add = TRUE.
.drop	When .drop = TRUE, empty groups are dropped.
x	A tbl()
y	tbls to join
by	a character vector of variables to join by. If NULL, the default, *_join() will do a natural join, using all variables with common names across the two tables. A message lists the variables so that you can check they're right (to suppress the message, simply explicitly list the variables that you want to join). To join by different variables on x and y use a named vector. For example, by = c("a" = "b") will match x.a to y.b.
copy	If x and y are not from the same data source, and copy is TRUE, then y will be copied into the same src as x. This allows you to join tables across srcs, but it is a potentially expensive operation so you must opt into it.
suffix	If there are non-joined duplicate variables in x and y, these suffixes will be added to the output to disambiguate them. Should be a character vector of length 2.
data	A data frame.
key	Unquoted variables to create the key (via id) after unnesting.
value	Names of new key and value columns, as strings or symbols. This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with

	rlang::ensym() (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).
na.rm	If TRUE, will remove rows from output where the value column in NA.
convert	If TRUE will automatically run type.convert() on the key column. This is useful if the column types are actually numeric, integer, or logical.
factor_key	If FALSE, the default, the key values will be stored as a character vector. If TRUE, will be stored as a factor, which preserves the original ordering of the columns.
fill	If set, missing values will be replaced with this value. Note that there are two types of missingness in the input: explicit missing values (i.e. NA), and implicit missings, rows that simply aren't present. Both types of missing value will be replaced by fill.
drop	If FALSE, will keep factor levels that don't appear in the data, filling in missing combinations with fill.
sep	If NULL, the column names will be taken from the values of key variable. If non-NULL, the column names will be given by "<key_name><sep><key_value>".
.key	The name of the new column, as a string or symbol. This argument is passed by expression and supports quasiquotation (you can unquote strings and symbols). The name is captured from the expression with rlang::ensym() (note that this kind of interface where symbols do not represent actual objects is now discouraged in the tidyverse; we support it here for backward compatibility).
.id	Data frame identifier - if supplied, will create a new column with name .id, giving a unique identifier. This is most useful if the list column is named.
.sep	If non-NULL, the names of unnested data frame columns will combine the name of the original list-col with the names from nested data frame, separated by .sep.
.direction	Direction in which to fill missing values. Currently either "down" (the default) or "up".

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

```
# Sum over sensors ----
pedestrian %>%
  summarise(Total = sum(Count))
# Back to tibble
pedestrian %>%
  as_tibble() %>%
  summarise(Total = sum(Count))
# example 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)
nested_stock <- stocksm %>%
  nest(-stock)
stocksm %>%
  group_by(stock) %>%
  nest()
nested_stock %>%
  unnest(key = id(stock))
stock_qtl <- stocksm %>%
  group_by(stock) %>%
  index_by(day3 = lubridate::floor_date(time, unit = "3 day")) %>%
  summarise(
    value = list(quantile(price)),
    qtl = list(c("0%", "25%", "50%", "75%", "100%"))
  )
unnest(stock_qtl, key = id(qtl))
```

`tile` *Tiling window calculation*

Description

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

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 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.
.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

Perform tiling windows on a tsibble by row

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 new column .id as part of the key. The output dimension will increase considerably with slide_tsibble() and stretch_tsibble().

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 = id(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]

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

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

`tsibble` *Create a tsibble object*

Description

Create a tsibble object

Usage

```
tsibble(..., key = id(), index, regular = TRUE)
```

Arguments

`...`	A set of name-value pairs. The names of "key" and "index" should be avoided as they are used as the arguments.
`key`	Variable(s) that define unique time indices, used in conjunction with the helper `id()`. If a univariate time series (without an explicit key), simply call `id()`.
`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`.

Details

A tsibble is sorted by its key first and index.

Value

A tsibble object.

Index

The time indices are preserved as the essential data component of the tsibble, instead of implicit attribute (for example, the `tsp` attribute in a `ts` object). A few index classes, such as `Date`, `POSIXct`, and `difftime`, forms the basis of the tsibble, with new additions yearweek, yearmonth, and yearquarter representing year-week, year-month, and year-quarter respectively. Any arbitrary index class are also supported, including `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 contains days ranging from 28 to 31 days, which results in irregular time space. This is also applicable to year-week and year-quarter.

Since the **tibble** that underlies the **tsibble** only accepts a 1d atomic vector or a list, a `tbl_ts` doesn't accept `POSIXlt` and `timeDate` columns.

Key

Key variable(s) together with the index uniquely identifies each record, which can be created via the id function as identifiers:

- Empty: an implicit variable `id()` resulting in a univariate time series.
- One or more variables: explicit variables. For example, `data(pedestrian)` and `data(tourism)` use the `id(Sensor)` & `id(Region, State, Purpose)` as the key respectively.

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: if there is only one entry for each key variable, the interval cannot be determined (?).

An interval is obtained based on the corresponding index representation:

- integer/numeric: either "unit" or "year"
- `yearquarter`/`yearqtr`: "quarter"
- `yearmonth`/`yearmon`: "month"
- `yearweek`: "week"
- `Date`: "day"
- `POSIXct`: "hour", "minute", "second", "millisecond", "microsecond"
- `nanotime`: "nanosecond"

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 one key ----
tsibble(
  qtr = rep(yearquarter("201001") + 0:9, 3),
  group = rep(c("x", "y", "z"), each = 10),
  value = rnorm(30),
  key = id(group)
)
```

`units_since` *Time units since Unix Epoch*

Description

Time units since Unix Epoch

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 = NULL, index = NULL, regular = NULL,
  validate = TRUE)
```

Arguments

x	A tsibble.
key	Variable(s) that define unique time indices, used in conjunction with the helper id(). If a univariate time series (without an explicit key), simply call id().
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.

Details

Default NULL inherits attributes from x.

Examples

```
pedestrian %>%
  group_by_key() %>%
  mutate(Hour_Since = Date_Time - min(Date_Time)) %>%
  update_tsibble(index = Hour_Since)
```

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

Description

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

pull_interval

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))
yearmonth("2018-07")
yearquarter(x)

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

# 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("2017-11-01")
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