Package 'tibbletime'

September 20, 2019

Type Package

Title Time Aware Tibbles

Version 0.1.3

Maintainer Davis Vaughan <davis@rstudio.com>

Description Built on top of the 'tibble' package, 'tibbletime' is an extension that allows for the creation of time aware tibbles. Some immediate advantages of this include: the ability to perform time-based subsetting on tibbles, quickly summarising and aggregating results by time periods, and creating columns that can be used as 'dplyr' time-based groups.

URL https://github.com/business-science/tibbletime

BugReports https://github.com/business-science/tibbletime/issues

License MIT + file LICENSE

Encoding UTF-8

RoxygenNote 6.1.1

Depends R (>= 3.2.0)

Imports assertthat (>= 0.2.0), dplyr (>= 0.7.4), glue (>= 1.1.1), hms (>= 0.4), lubridate (>= 1.6.0), purrr (>= 0.2.3), Rcpp (>= 0.12.7), rlang (>= 0.1.6), tibble (>= 1.4.1), tidyselect (>= 0.2.5), vctrs (>= 0.2.0), zoo (>= 1.8-0), lifecycle

Suggests broom, covr, gapminder, knitr, testthat, tidyr (>= 1.0.0)

VignetteBuilder knitr

LinkingTo Rcpp

LazyData yes

NeedsCompilation yes

Author Davis Vaughan [aut, cre], Matt Dancho [aut]

Repository CRAN

Date/Publication 2019-09-20 05:00:02 UTC

R topics documented:

`as_period` *Change* `tbl_time` *periodicity*

Description

Convert a `tbl_time` object from daily to monthly, from minute data to hourly, and more. This allows the user to easily aggregate data to a less granular level by taking the value from either the beginning or end of the period.

Usage

```
as_period(.tbl_time, period = "yearly", start_date = NULL,
  side = "start", include_endpoints = FALSE, ...)
```

Arguments

`.tbl_time`	A `tbl_time` object.
`period`	A character specification used for time-based grouping. The general format to use is `"frequency period"` where frequency is a number like 1 or 2, and period is an interval like weekly or yearly. There must be a space between the two. Note that you can pass the specification in a flexible way: • 1 Year: `'1 year'` / `'1 Y'` / `'1 yearly'` / `'yearly'`

This shorthand is available for year, quarter, month, day, hour, minute, second, millisecond and microsecond periodicities.

Additionally, you have the option of passing in a vector of dates to use as custom and more flexible boundaries.

`start_date` Optional argument used to specify the start date for the first group. The default is to start at the closest period boundary below the minimum date in the supplied index.

`side` Whether to return the date at the beginning or the end of the new period. By default, the `"start"` of the period. Use `"end"` to change to the end of the period.

`include_endpoints`
Whether to include the first or last data point in addition to the transformed data.

`...` Not currently used.

Details

This function respects `dplyr::group_by()` groups.

The `side` argument is useful when you want to return data at, say, the end of a quarter, or the end of a month.

`include_endpoints` can be useful when calculating a change over time. In addition to changing to monthly dates, you often need the first data point as a baseline for the first calculation.

Examples

```
# Basic usage ----------------------------------------------------------------

# FB stock prices
data(FB)
FB <- as_tbl_time(FB, date)

# Aggregate FB to yearly data
as_period(FB, "yearly")

# Aggregate FB to every 2 years
as_period(FB, "2 years")

# Aggregate FB to yearly data, but use the last data point available
# in that period
as_period(FB, "yearly", side = "end")

# Aggregate FB to yearly data, end of period, and include the first
# endpoint
as_period(FB, "yearly", side = "end", include_endpoints = TRUE)

# Aggregate to weekly. Notice that it only uses the earliest day available
# in the data set at that periodicity. It will not set the date of the first
# row to 2013-01-01 because that date did not exist in the original data set.
as_period(FB, "weekly")
```

```
# Aggregate to every other week
as_period(FB, "2 weeks")

# FB is daily data, aggregate to minute?
# Not allowed for Date class indices, an error is thrown
# as_period(FB, "minute")

# Grouped usage ----------------------------------------------------------

# FANG contains Facebook, Amazon, Netflix and Google stock prices
data(FANG)
FANG <- as_tbl_time(FANG, date)

FANG <- dplyr::group_by(FANG, symbol)

# Respects groups
as_period(FANG, "yearly")

# Every 6 months, respecting groups
as_period(FANG, "6 months")

# Using start_date -------------------------------------------------------


#### One method using start_date

# FB stock prices
data(FB)
FB <- as_tbl_time(FB, date)

# The Facebook series starts at 2013-01-02 so the 'every 2 day' counter
# starts at that date as well. Groups become (2013-01-02, 2013-01-03),
# (2013-01-04, 2013-01-05) and so on.
as_period(FB, "2 day")

# Specifying the `start_date = "2013-01-01"` might be preferable.
# Groups become (2013-01-01, 2013-01-02), (2013-01-03, 2013-01-04) and so on.
as_period(FB, "2 day", start_date = "2013-01-01")

#### Equivalent method using an index vector

# FB stock prices
data(FB)
FB <- as_tbl_time(FB, date)

custom_period <- create_series(
  time_formula = dplyr::first(FB$date) - 1 ~ dplyr::last(FB$date),
  period       = "2 day",
  class        = "Date",
  as_vector    = TRUE)

FB %>%
  as_tbl_time(date) %>%
```

```
  as_period(period = custom_period)

# Manually calculating returns at different periods -------------------------

data(FB)

# Annual Returns
# Convert to end of year periodicity, but include the endpoints to use as
# a reference for the first return calculation. Then calculate returns.
FB %>%
  as_tbl_time(date) %>%
  as_period("1 y", side = "end", include_endpoints = TRUE) %>%
  dplyr::mutate(yearly_return = adjusted / dplyr::lag(adjusted) - 1)
```

`as_tbl_time` *Create* `tbl_time` *objects*

Description

`tbl_time` objects have a time index that contains information about which column should be used for time-based subsetting and other time-based manipulation. Otherwise, they function as normal tibbles.

Usage

```
as_tbl_time(x, index = NULL, ...)

tbl_time(x, index = NULL)
```

Arguments

`x`	An object to be converted to `tbl_time`. This is generally a `tibble::tibble()`, or an object that can first be coerced to a `tibble`.
`index`	The bare column name of the column to be used as the index.
`...`	Arguments passed to `tibble::as_tibble()` if coercion is necessary first.

Details

The information stored about `tbl_time` objects are the `index_quo` and the `index_time_zone`. These are stored as attributes, with the `index_quo` as a `rlang::quosure()` and the `time_zone` as a string.

Currently, `Date` and `POSIXct` classes are fully supported. `yearmon`, `yearqtr`, and `hms` have experimental support. Due to dplyr's handling of S3 classes like these 3, the classes are lost when you manipulate the index columns directly.

Examples

```
# Converting a data.frame to a `tbl_time`
# Using Date index
ex1 <- data.frame(date = Sys.Date(), value = 1)
ex1_tbl_time <- as_tbl_time(ex1, date)
class(ex1_tbl_time)
attributes(ex1_tbl_time)

# Converting a tibble to a `tbl_time`
# Using POSIXct index
ex2 <- tibble::tibble(
  time  = as.POSIXct(c("2017-01-01 10:12:01", "2017-01-02 12:12:01")),
  value = c(1, 2)
)
as_tbl_time(ex2, time)
```

---

ceiling_index *A simple wrapper of* lubridate::ceiling_date()

Description

This is a thin wrapper around a lubridate::ceiling_date() that works for hms, yearmon, and yearqtr classes as well.

Usage

```
ceiling_index(x, unit = "seconds")
```

Arguments

x	a vector of date-time objects
unit	a character string specifying a time unit or a multiple of a unit to be rounded to. Valid base units are second, minute, hour, day, week, month, bimonth, quarter, season, halfyear and year. Arbitrary unique English abbreviations as in the period() constructor are allowed. Rounding to multiple of units (except weeks) is supported.

See Also

lubridate::ceiling_date()

Examples

```
data(FB)
dplyr::mutate(FB, date2 = ceiling_index(date, "year"))

time_test <- create_series('00:00:00'~'12:00:00',
                           '1 minute', class = "hms")

dplyr::mutate(time_test, date2 = ceiling_index(date, "hour"))
```

`collapse_by` *Collapse a tbl_time object by its index*

Description

Collapse the index of a `tbl_time` object by time period. The index column is altered so that all dates that fall in a specified interval share a common date.

Usage

```
collapse_by(.tbl_time, period = "yearly", start_date = NULL,
  side = "end", clean = FALSE, ...)
```

Arguments

`.tbl_time`	A `tbl_time` object.
`period`	A character specification used for time-based grouping. The general format to use is `"frequency period"` where frequency is a number like 1 or 2, and period is an interval like weekly or yearly. There must be a space between the two. Note that you can pass the specification in a flexible way: • 1 Year: `'1 year'` / `'1 Y'` / `'1 yearly'` / `'yearly'` This shorthand is available for year, quarter, month, day, hour, minute, second, millisecond and microsecond periodicities. Additionally, you have the option of passing in a vector of dates to use as custom and more flexible boundaries.
`start_date`	Optional argument used to specify the start date for the first group. The default is to start at the closest period boundary below the minimum date in the supplied index.
`side`	Whether to return the date at the beginning or the end of the new period. By default, the "end" of the period. Use "start" to change to the start of the period.
`clean`	Whether or not to round the collapsed index up / down to the next period boundary. The decision to round up / down is controlled by the side argument.
`...`	Not currently used.

Details

collapse_by() is a simplification of a call to dplyr::mutate() to collapse an index column using collapse_index().

Examples

```
# Basic functionality ------------------------------------------------------

# Facebook stock prices
data(FB)
FB <- as_tbl_time(FB, date)

# Collapse to weekly dates
collapse_by(FB, "weekly")

# A common workflow is to group on the collapsed date column
# to perform a time based summary
FB %>%
  collapse_by("yearly") %>%
  dplyr::group_by(date) %>%
  dplyr::summarise_if(is.numeric, mean)

# Grouped functionality ----------------------------------------------------

data(FANG)
FANG <- FANG %>%
  as_tbl_time(date) %>%
  dplyr::group_by(symbol)

# Collapse each group to monthly,
# calculate monthly standard deviation for each column
FANG %>%
  collapse_by("monthly") %>%
  dplyr::group_by(date, add = TRUE) %>%
  dplyr::summarise_all(sd)
```

---

collapse_index *Collapse an index vector so that all observations in an interval share the same date*

Description

When collapse_index() is used, the index vector is altered so that all dates that fall in a specified interval share a common date. The most common use case for this is to then group on the collapsed index.

Usage

```
collapse_index(index, period = "yearly", start_date = NULL,
  side = "end", clean = FALSE, ...)
```

Arguments

index	An index vector.
period	A character specification used for time-based grouping. The general format to use is "frequency period" where frequency is a number like 1 or 2, and period is an interval like weekly or yearly. There must be a space between the two. Note that you can pass the specification in a flexible way: • 1 Year: '1 year' / '1 Y' / '1 yearly' / 'yearly' This shorthand is available for year, quarter, month, day, hour, minute, second, millisecond and microsecond periodicities. Additionally, you have the option of passing in a vector of dates to use as custom and more flexible boundaries.
start_date	Optional argument used to specify the start date for the first group. The default is to start at the closest period boundary below the minimum date in the supplied index.
side	Whether to return the date at the beginning or the end of the new period. By default, the "end" of the period. Use "start" to change to the start of the period.
clean	Whether or not to round the collapsed index up / down to the next period boundary. The decision to round up / down is controlled by the side argument.
...	Not currently used.

Details

The collapse_by() function provides a shortcut for the most common use of collapse_index(), calling the function inside a call to mutate() to modify the index directly. For more flexibility, like the nesting example below, use collapse_index().

Because this is often used for end of period summaries, the default is to use side = "end". Note that this is the opposite of as_period() where the default is side = "start".

The clean argument is especially useful if you have an irregular series and want cleaner dates to report for summary values.

Examples

```
# Basic functionality ------------------------------------------------------

# Facebook stock prices
data(FB)
FB <- as_tbl_time(FB, date)

# Collapse to weekly dates
dplyr::mutate(FB, date = collapse_index(date, "weekly"))
```

```
# A common workflow is to group on the new date column
# to perform a time based summary
FB %>%
  dplyr::mutate(date = collapse_index(date, "yearly")) %>%
  dplyr::group_by(date) %>%
  dplyr::summarise_if(is.numeric, mean)

# You can also assign the result to a separate column and use that
# to nest on, allowing for 'period nests' that keep the
# original dates in the nested tibbles.
FB %>%
  dplyr::mutate(nest_date = collapse_index(date, "2 year")) %>%
  dplyr::group_by(nest_date) %>%
  tidyr::nest()

# Grouped functionality ----------------------------------------------------

data(FANG)
FANG <- FANG %>%
  as_tbl_time(date) %>%
  dplyr::group_by(symbol)

# Collapse each group to monthly,
# calculate monthly standard deviation for each column
FANG %>%
  dplyr::mutate(date = collapse_index(date, "monthly")) %>%
  dplyr::group_by(date, add = TRUE) %>%
  dplyr::summarise_all(sd)
```

`create_series` *Create a* `tbl_time` *object with a sequence of regularly spaced dates*

Description

`create_series()` allows the user to quickly create a `tbl_time` object with a `date` column populated with a sequence of dates.

Usage

```
create_series(time_formula, period = "daily", class = "POSIXct",
  include_end = FALSE, tz = "UTC", as_vector = FALSE)
```

Arguments

`time_formula`	A period to create the series over. This is specified as a formula. See the `Details` section of `filter_time()` for more information.

period	A character specification used for time-based grouping. The general format to use is "frequency period" where frequency is a number like 1 or 2, and period is an interval like weekly or yearly. There must be a space between the two. Note that you can pass the specification in a flexible way: • 1 Year: '1 year' / '1 Y' / '1 yearly' / 'yearly' This shorthand is available for year, quarter, month, day, hour, minute, second, millisecond and microsecond periodicities. Additionally, you have the option of passing in a vector of dates to use as custom and more flexible boundaries.
class	One of "Date", "POSIXct", "hms", "yearmon", "yearqtr". The default is "POSIXct".
include_end	Whether to always include the RHS of the time_formula even if it does not match the regularly spaced index.
tz	Time zone of the new series.
as_vector	Should the series be returned as a vector instead of a tibble?

Examples

```
# Every day in 2013
create_series(~'2013', 'daily')

# Every other day in 2013
create_series(~'2013', '2 d')

# Every quarter in 2013
create_series(~'2013', '1 q')

# Daily series for 2013-2015
create_series('2013' ~ '2015', '1 d')

# Minute series for 2 months
create_series('2012-01' ~ '2012-02', 'M')

# Second series for 2 minutes
create_series('2011-01-01 12:10:00' ~ '2011-01-01 12:12:00', 's')

# Date class
create_series(~'2013', 'day', class = "Date")

# yearmon class
create_series(~'2013', 'month', class = "yearmon")

# hms class. time_formula specified as HH:MM:SS here
create_series('00:00:00' ~ '12:00:00', 'second' , class = "hms")

# Subsecond series
create_series('2013' ~ '2013-01-01 00:00:01', period = "10 millisec")
milli <- create_series('2013' ~ '2013-01-01 00:00:01', period = ".1 sec")
```

```
# Check that 'milli' is correct by running:
# options("digits.secs" = 4)
# options("digits" = 18)
# milli$date
# as.numeric(milli$date)
```

FANG *Stock prices for Facebook, Amazon, Netflix and Google from 2013-2016*

Description

A dataset containing the date, open, high, low, close, volume, and adjusted stock prices for Facebook, Amazon, Netflix and Google from 2013-2016.

Usage

```
FANG
```

Format

A tibble with 4,032 rows and 8 variables:

symbol stock ticker symbol

date trade date

open stock price at the open of trading, in USD

high stock price at the highest point during trading, in USD

low stock price at the lowest point during trading, in USD

close stock price at the close of trading, in USD

volume number of shares traded

adjusted stock price at the close of trading adjusted for stock splits, in USD

Source

http://www.investopedia.com/terms/f/fang-stocks-fb-amzn.asp

FB *Stock prices for Facebook from 2013-2016*

Description

A dataset containing the date, open, high, low, close, volume, and adjusted stock prices for Facebook from 2013-2016.

Usage

```
FB
```

Format

A tibble with 1,008 rows and 8 variables:

symbol stock ticker symbol

date trade date

open stock price at the open of trading, in USD

high stock price at the highest point during trading, in USD

low stock price at the lowest point during trading, in USD

close stock price at the close of trading, in USD

volume number of shares traded

adjusted stock price at the close of trading adjusted for stock splits, in USD

Source

http://www.investopedia.com/terms/f/fang-stocks-fb-amzn.asp

filter_time *Succinctly filter a* tbl_time *object by its index*

Description

Use a concise filtering method to filter a tbl_time object by its index.

Usage

```
filter_time(.tbl_time, time_formula)

## S3 method for class 'tbl_time'
x[i, j, drop = FALSE]
```

Arguments

.tbl_time	A tbl_time object.
time_formula	A period to filter over. This is specified as a formula. See Details.
x	Same as .tbl_time but consistent naming with base R.
i	A period to filter over. This is specified the same as time_formula or can use the traditional row extraction method.
j	Optional argument to also specify column index to subset. Works exactly like the normal extraction operator.
drop	Will always be coerced to FALSE by tibble.

Details

The time_formula is specified using the format from ~ to. Each side of the time_formula is specified as the character 'YYYY-MM-DD HH:MM:SS', but powerful shorthand is available. Some examples are:

- **Year:** '2013' ~ '2015'
- **Month:** '2013-01' ~ '2016-06'
- **Day:** '2013-01-05' ~ '2016-06-04'
- **Second:** '2013-01-05 10:22:15' ~ '2018-06-03 12:14:22'
- **Variations:** '2013' ~ '2016-06'

The time_formula can also use a one sided formula.

- **Only dates in 2015:** ~'2015'
- **Only dates March 2015:** ~'2015-03'

The time_formula can also use 'start' and 'end' as keywords for your filter.

- **Start of the series to end of 2015:** 'start' ~ '2015'
- **Start of 2014 to end of series:** '2014' ~ 'end'

All shorthand dates are expanded:

- The from side is expanded to be the first date in that period
- The to side is expanded to be the last date in that period

This means that the following examples are equivalent (assuming your index is a POSIXct):

- '2015' ~ '2016' == '2015-01-01 + 00:00:00' ~ '2016-12-31 + 23:59:59'
- ~'2015' == '2015-01-01 + 00:00:00' ~ '2015-12-31 + 23:59:59'
- '2015-01-04 + 10:12' ~ '2015-01-05' == '2015-01-04 + 10:12:00' ~ '2015-01-05 + 23:59:59'

Special parsing is done for indices of class hms. The from ~ to time formula is specified as only HH:MM:SS.

- **Start to 5th second of the 12th hour:** 'start' ~ '12:00:05'

- **Every second in the 12th hour:** `~'12'`

Subsecond resolution is also supported, however, R has a unique way of handling and printing subsecond dates and the user should be comfortable with this already. Specify subsecond resolution like so: `'2013-01-01 00:00:00.1' ~ '2013-01-01 00:00:00.2'`. Note that one sided expansion does not work with subsecond resolution due to seconds and subseconds being grouped together into 1 number (i.e. 1.2 seconds). This means `~'2013-01-01 00:00:00'` does not expand to something like `'2013-01-01 00:00:00.00' ~ '2013-01-01 00:00:00.99'`, but only expands to include whole seconds.

This function respects `dplyr::group_by()` groups.

Examples

```
# FANG contains Facebook, Amazon, Netflix and Google stock prices
data(FANG)
FANG <- as_tbl_time(FANG, date) %>%
  dplyr::group_by(symbol)

# 2013-01-01 to 2014-12-31
filter_time(FANG, '2013' ~ '2014')

# 2013-05-25 to 2014-06-04
filter_time(FANG, '2013-05-25' ~ '2014-06-04')

# Using the `[` subset operator
FANG['2014'~'2015']

# Using `[` and one sided formula for only dates in 2014
FANG[~'2014']

# Using `[` and column selection
FANG['2013'~'2016', c("date", "adjusted")]

# Variables are unquoted using rlang
lhs_date <- "2013"
rhs_date <- as.Date("2014-01-01")
filter_time(FANG, lhs_date ~ rhs_date)

# Use the keywords 'start' and 'end' to conveniently access ends
filter_time(FANG, 'start' ~ '2014')

# hms (hour, minute, second) classes have special parsing
hms_example <- create_series(~'12:01', 'second', class = 'hms')
filter_time(hms_example, 'start' ~ '12:01:30')
```

floor_index *A simple wrapper of* lubridate::floor_date()

Description

This is a thin wrapper around a lubridate::floor_date() that works for hms, yearmon, and yearqtr classes as well.

Usage

```
floor_index(x, unit = "seconds")
```

Arguments

x	a vector of date-time objects
unit	a character string specifying a time unit or a multiple of a unit to be rounded to. Valid base units are second, minute, hour, day, week, month, bimonth, quarter, season, halfyear and year. Arbitrary unique English abbreviations as in the period() constructor are allowed. Rounding to multiple of units (except weeks) is supported.

See Also

lubridate::floor_date()

Examples

```
data(FB)
dplyr::mutate(FB, date2 = floor_index(date, "year"))

time_test <- create_series('00:00:00'~'12:00:00',
                           '1 minute', class = "hms")

dplyr::mutate(time_test, date2 = floor_index(date, "hour"))
```

getters *Getters*

Description

Accessors to attributes of tbl_time objects.

Usage

```
get_index_quo(.tbl_time)

get_index_char(.tbl_time)

get_index_col(.tbl_time)

get_index_time_zone(.tbl_time)

get_index_class(.tbl_time)
```

Arguments

.tbl_time	A tbl_time object.

---

new_tbl_time *Create a new tbl_time object*

Description

Often used internally by developers extending tibbletime

Usage

```
new_tbl_time(x, index_quo, index_time_zone, ..., subclass = NULL)
```

Arguments

x	A tibble or data.frame
index_quo	The quo that references the index column
index_time_zone	The index time zone
...	Other attributes passed through to new_tibble()
subclass	A subclass to have as a child

parse_period *Parse a character period specification*

Description

The period is parsed into frequency and period and returned as a named list.

Usage

```
parse_period(period)
```

Arguments

period	A character specification used for time-based grouping. The general format to use is "frequency period" where frequency is a number like 1 or 2, and period is an interval like weekly or yearly. There must be a space between the two. Note that you can pass the specification in a flexible way: • 1 Year: '1 year' / '1 Y' / '1 yearly' / 'yearly' This shorthand is available for year, quarter, month, day, hour, minute, second, millisecond and microsecond periodicities. Additionally, you have the option of passing in a vector of dates to use as custom and more flexible boundaries.

Examples

```
parse_period('2 day')
```

partition_index *Partition an index vector into an integer vector representing groups*

Description

partition_index() takes an index vector and returns an integer vector that can be used for grouping by periods. This is the workhorse for many other tibbletime functions.

Usage

```
partition_index(index, period = "yearly", start_date = NULL, ...)
```

Arguments

index	A vector of date indices to create groups for.
period	A character specification used for time-based grouping. The general format to use is "frequency period" where frequency is a number like 1 or 2, and period is an interval like weekly or yearly. There must be a space between the two. Note that you can pass the specification in a flexible way: • 1 Year: '1 year' / '1 Y' / '1 yearly' / 'yearly' This shorthand is available for year, quarter, month, day, hour, minute, second, millisecond and microsecond periodicities. Additionally, you have the option of passing in a vector of dates to use as custom and more flexible boundaries.
start_date	Optional argument used to specify the start date for the first group. The default is to start at the closest period boundary below the minimum date in the supplied index.
...	Not currently used.

Details

This function is used internally, but may provide the user extra flexibility in some cases.

Grouping can only be done on the minimum periodicity of the index and above. This means that a daily series cannot be grouped by minute. An hourly series cannot be grouped by 5 seconds, and so on. If the user attempts this, an error will be thrown.

See Also

as_period(), collapse_index()

Examples

```
data(FB)

partition_index(FB$date, '2 year')

dplyr::mutate(FB, partition_index = partition_index(date, '2 day'))
```

---

posixct_numeric_to_datetime

Converting a posixct numeric time back to a classed datetime

Description

Converting a posixct numeric time back to a classed datetime

Usage

```
posixct_numeric_to_datetime(x, class = "POSIXct", ..., tz = NULL)
```

Arguments

x	A posixct numeric vector
class	The class to convert to
...	Extra arguments passed on the the specific coercion function
tz	The time zone to convert to. The default UTC is used if none is supplied

---

reconstruct *Reconstruct an S3 class from a template*

Description

This is an implementation of `sloop::reconstruct()` that users can ignore. Once `sloop` is on CRAN, this function will be removed and that version will be used. It currently must be exported for use in `tidyquant`.

Usage

```
reconstruct(new, old)
```

Arguments

new	Freshly created object
old	Existing object to use as template

---

rollify *Create a rolling version of any function*

Description

`rollify` returns a rolling version of the input function, with a rolling `window` specified by the user.

Usage

```
rollify(.f, window = 1, unlist = TRUE, na_value = NULL)
```

Arguments

`.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.
`window`	The window size to roll over
`unlist`	If the function returns a single value each time it is called, use `unlist = TRUE`. If the function returns more than one value, or a more complicated object (like a linear model), use `unlist = FALSE` to create a list-column of the rolling results.
`na_value`	A default value for the `NA` values at the beginning of the roll.

Details

The intended use of `rollify` is to turn a function into a rolling version of itself for use inside of a call to `dplyr::mutate()`, however it works equally as well when called from `purrr::map()`.

Because of it's intended use with `dplyr::mutate()`, `rollify` creates a function that always returns output with the same length of the input, aligned right, and filled with `NA` unless otherwise specified by `na_value`.

The form of the `.f` argument is the same as the form that can be passed to `purrr::map()`. Use `.x` or `.` to refer to the first object to roll over, and `.y` to refer to the second object if required. The examples explain this further.

If optional arguments to the function are required, specify them in the call to `rollify`, and not in the call to the rolling version of the function. See the examples for more details.

See Also

purrr::safely, purrr::possibly

Examples

```
# Rolling mean ------------------------------------------------------------

data(FB)

# Turn the normal mean function into a rolling mean with a 5 row window
mean_roll_5 <- rollify(mean, window = 5)
```

```
dplyr::mutate(FB,
       normal_mean  = mean(adjusted),
       rolling_mean = mean_roll_5(adjusted))

# There's nothing stopping you from combining multiple rolling functions with
# different window sizes in the same mutate call
mean_roll_10 <- rollify(mean, window = 10)

dplyr::mutate(FB,
       rolling_mean_5  = mean_roll_5(adjusted),
       rolling_mean_10 = mean_roll_10(adjusted))

# Functions with multiple args and optional args ----------------------------

# With 2 args, use the purrr syntax of ~ and .x, .y
# Rolling correlation example
cor_roll <- rollify(~cor(.x, .y), window = 5)

dplyr::mutate(FB, running_cor = cor_roll(adjusted, open))

# With >2 args, create an anonymous function with >2 args or use
# the purrr convention of ..1, ..2, ..3 to refer to the arguments
avg_of_avgs <- rollify(function(x, y, z) {
                         (mean(x) + mean(y) + mean(z)) / 3
                       },
                       window = 10)

# Or
avg_of_avgs <- rollify(~(mean(..1) + mean(..2) + mean(..3)) / 3,
                       window = 10)

dplyr::mutate(FB, avg_of_avgs = avg_of_avgs(open, high, low))

# Optional arguments MUST be passed at the creation of the rolling function
# Only data arguments that are "rolled over" are allowed when calling the
# rolling version of the function
FB$adjusted[1] <- NA

roll_mean_na_rm <- rollify(~mean(.x, na.rm = TRUE), window = 5)

dplyr::mutate(FB, roll_mean = roll_mean_na_rm(adjusted))

# Returning multiple values -------------------------------------------------

data(FB)

# If the function returns >1 value, set the `unlist = FALSE` argument
# Running 5 number summary
summary_roll <- rollify(summary, window = 5, unlist = FALSE)

FB_summarised <- dplyr::mutate(FB, summary_roll = summary_roll(adjusted))
FB_summarised$summary_roll[[5]]
```

```
# dplyr::bind_rows() is often helpful in these cases to get
# meaningful output

summary_roll <- rollify(~dplyr::bind_rows(summary(.)), window = 5, unlist = FALSE)
FB_summarised <- dplyr::mutate(FB, summary_roll = summary_roll(adjusted))
FB_summarised %>%
  dplyr::filter(!is.na(summary_roll)) %>%
  tidyr::unnest(summary_roll)

# Rolling regressions ------------------------------------------------------

# Extending an example from R 4 Data Science on "Many Models".
# For each country in the gapminder data, calculate a linear regression
# every 5 periods of lifeExp ~ year
library(gapminder)

# Rolling regressions are easy to implement
lm_roll <- rollify(~lm(.x ~ .y), window = 5, unlist = FALSE)

gapminder %>%
  dplyr::group_by(country) %>%
  dplyr::mutate(rolling_lm = lm_roll(lifeExp, year))

# Rolling with groups ------------------------------------------------------

# One of the most powerful things about this is that it works with
# groups since `mutate` is being used
data(FANG)
FANG <- FANG %>%
  dplyr::group_by(symbol)

mean_roll_3 <- rollify(mean, window = 3)

FANG %>%
  dplyr::mutate(mean_roll = mean_roll_3(adjusted)) %>%
  dplyr::slice(1:5)
```

`tibbletime` *tibbletime: time-aware tibbles*

Description

Built on top of the 'tibble' package, 'tibbletime' is an extension that allows for the creation of time aware tibbles. Some immediate advantages of this include: the ability to perform time based subsetting on tibbles, quickly summarising and aggregating results by time periods, and calling functions similar in spirit to the map family from 'purrr' on time based tibbles.

Author(s)

Maintainer: Davis Vaughan <davis@rstudio.com>

Authors:

- Matt Dancho <mdancho@business-science.io>

See Also

Useful links:

- https://github.com/business-science/tibbletime
- Report bugs at https://github.com/business-science/tibbletime/issues

Index