Package 'rutils'

October 3, 2017

Type Package

Title Utility Functions for Simplifying Financial Data Management and Modeling

Version 0.2

Date 2017-09-04

Author Jerzy Pawlowski (algoquant)

Maintainer Jerzy Pawlowski <jp3900@nyu.edu>

Description Functions for managing object names and attributes, applying functions over lists, managing objects in environments.

License GPL (>= 2)

Depends xts, quantmod, RcppRoll,

Suggests knitr, rmarkdown, testthat

VignetteBuilder knitr

LazyData true

Repository GitHub

URL https://github.com/algoquant/rutils

RoxygenNote 5.0.1

R topics documented:

`adjust_ohlc` *Adjust the first four columns of* OHLC *data using the "adjusted" price column.*

Description

Adjust the first four columns of *OHLC* data using the "adjusted" price column.

Usage

```
adjust_ohlc(oh_lc)
```

Arguments

`oh_lc`	An *OHLC* time series of prices in *xts* format.

Details

Adjusts the first four *OHLC* price columns by multiplying them by the ratio of the "adjusted" (sixth) price column, divided by the *Close* (fourth) price column.

Value

An *OHLC* time series with the same dimensions as the input series.

Examples

```
# adjust VTI prices
VTI <- rutils::adjust_ohlc(rutils::env_etf$VTI)
```

`calc_endpoints` *Calculate a vector of equally spaced end points along the elements of a vector, matrix, or time series.*

Description

Calculate a vector of equally spaced end points along the elements of a vector, matrix, or time series.

Usage

```
calc_endpoints(x_ts, inter_val, stub_front = TRUE)
```

Arguments

`x_ts`	A vector, matrix, or time series.
`inter_val`	The number of elements between neighboring end points. or a *string* representing a time period (minutes, hours, days, etc.)
`stub_front`	*Boolean* argument: if `TRUE` then add a stub interval at the beginning, else add a stub interval at the end. (default is `TRUE`)

Details

The end points are a vector of integers which divide the elements (rows) of `x_ts` into equally spaced intervals. If `inter_val` is an *integer* then `calc_endpoints()` calculates the number of whole intervals that fit over the elements (rows) of `x_ts`. If a whole number of intervals doesn't fit over the elements (rows) of `x_ts`, then `calc_endpoints()` adds a stub interval either at the beginning (the default) or at the end.

The function `calc_endpoints()` is a generalization of function `endpoints()` from package xts, since `inter_val` can accept both *integer* and *string* values.

If `inter_val` is a *string* representing a time period (minutes, hours, days, etc.), then `calc_endpoints()` simply calls the function `endpoints()` from package xts.

Value

An *integer* vector of equally spaced end points (vector of integers).

Examples

```
# calculate end points with initial stub interval
rutils::calc_endpoints(1:100, inter_val=11)
# calculate end points with a stub interval at the end
rutils::calc_endpoints(rutils::env_etf$VTI, inter_val=365, stub_front=FALSE)
# calculate end points at the end of every hour
rutils::calc_endpoints(rutils::env_etf$VTI, inter_val="hours")
```

---

chart_dygraph *Plot an interactive* dygraphs *candlestick plot with background shading for an* OHLC *time series in* xts *format.*

Description

Plot an interactive *dygraphs* candlestick plot with background shading for an *OHLC* time series in *xts* format.

Usage

```
chart_dygraph(oh_lc, in_dic = NULL, ...)
```

Arguments

oh_lc	An *OHLC* time series in *xts* format.
in_dic	A *Boolean* time series in *xts* format for specifying the shading areas, with TRUE indicating "lightgreen" shading, and FALSE indicating "antiquewhite" shading (default is NULL).
...	additional arguments to function dygraphs::dygraph().

Details

The function chart_dygraph() creates an interactive dygraphs candlestick plot with background shading for an *OHLC* time series. The function chart_dygraph() uses plotting functions from the package dygraphs.

Value

A dygraphs plot object, and a dygraphs plot produced as a side effect.

Examples

```
# plot an interactive dygraphs candlestick plot with background shading
oh_lc <- rutils::env_etf$VTI
v_wap <- TTR::VWAP(price=quantmod::Ad(oh_lc), volume=quantmod::Vo(oh_lc), n=20)
oh_lc <- cbind(oh_lc[, c(1:3, 6)], v_wap)["2009-02/2009-04"]
rutils::chart_dygraph(oh_lc, in_dic=(oh_lc[, 4] > v_wap))
```

---

chart_dygraph2y *Plot an interactive* dygraphs *line plot for two* xts *time series, with two* "y" *axes.*

Description

Plot an interactive *dygraphs* line plot for two *xts* time series, with two *"y"* axes.

Usage

```
chart_dygraph2y(x_ts, ...)
```

Arguments

x_ts	An *xts* time series with two columns.
...	additional arguments to function dygraphs::dygraph().

Details

The function chart_dygraph2y() creates an interactive dygraphs line plot with two *"y"* axes. The function chart_dygraph2y() uses plotting functions from the package dygraphs.

Value

A dygraphs plot object.

Examples

```
# plot an interactive dygraphs line plot with two "y" axes
price_s <- cbind(Ad(rutils::env_etf$VTI), Ad(rutils::env_etf$IEF))
colnames(price_s) <- get_name(colnames(price_s), field=2)
rutils::chart_dygraph2y(price_s)
```

---

chart_xts — *Plot either a line plot or a candlestick plot of an* xts *time series, with custom line colors, y-axis range, and with vertical background shading.*

Description

A wrapper for function chart_Series() from package quantmod.

Usage

```
chart_xts(x_ts, col_ors = NULL, ylim = NULL, in_dic = NULL, x_11 = TRUE,
  ...)
```

Arguments

x_ts	An *xts* time series or an *OHLC* time series.
col_ors	A vector of *strings* with the custom line colors.
ylim	A *numeric* vector with two elements containing the y-axis range.
in_dic	A *Boolean* vector or *xts* time series for specifying the shading areas, with TRUE indicating "lightgreen" shading, and FALSE indicating "antiquewhite" shading.
x_11	*Boolean* argument: if TRUE then open x11 window for plotting, else plot in standard window (default is TRUE).
...	additional arguments to function chart_Series().

Details

The function `chart_xts()` plots a line plot of a *xts* time series, or a candlestick plot if *x_ts* is a *OHLC* time series. The function `chart_xts()` plots with custom line colors and vertical background shading, using the function `chart_Series()` from package quantmod. By default `chart_xts()` opens and plots in an x11 window.

The function `chart_xts()` extracts the `chart_Series()` chart object and modifies its *ylim* parameter using accessor and setter functions. It also adds background shading specified by the `in_dic` argument, using function `add_TA()`. The `in_dic` argument should have the same length as the `x_ts` time series. Finally the function `chart_xts()` plots the chart object and returns it invisibly.

Value

A `chart_Series()` object returned invisibly.

Examples

```
# plot candlestick chart with shading
rutils::chart_xts(rutils::env_etf$VTI["2015-11"],
  name="VTI in Nov 2015", ylim=c(102, 108),
  in_dic=zoo::index(rutils::env_etf$VTI["2015-11"]) > as.Date("2015-11-18"))
# plot two time series with custom line colors
rutils::chart_xts(na.omit(cbind(rutils::env_etf$XLU[, 4],
  rutils::env_etf$XLP[, 4])), col_ors=c("blue", "green"))
```

---

chart_xts2y *Plot two* xts *time series with two y-axes in an x11 window.*

Description

Plot two *xts* time series with two y-axes in an x11 window.

Usage

```
chart_xts2y(x_ts, col_or = "red", x_11 = TRUE, ...)
```

Arguments

`col_or`	A string specifying the color of the second line and axis (default is `"red"`).
`x_11`	*Boolean* argument: if `TRUE` then open x11 window for plotting, else plot in standard window (default is `TRUE`).
`...`	additional arguments to function `plot.zoo()`.
`An`	x_ts *xts* time series with two columns.

Details

The function `chart_xts2y()` creates a plot of two *xts* time series with two y-axes. By default `chart_xts2y()` opens and plots in an x11 window. The function `chart_xts2y()` uses the standard plotting functions from base *R*, and the function `plot.zoo()` from package zoo.

Value

The *x_ts* column names returned invisibly, and a plot in an x11 window produced as a side effect.

Examples

```
# plot two time series
rutils::chart_xts2y(cbind(quantmod::Cl(rutils::env_etf$VTI),
                          quantmod::Cl(rutils::env_etf$IEF))["2015"])
```

`diff_it` *Calculate the row differences of a* numeric *vector or matrix.*

Description

Calculate the row differences of a *numeric* vector or matrix.

Usage

```
diff_it(in_put, lagg = 1)
```

Arguments

`in_put`	A *numeric* vector or matrix.
`lagg`	An integer equal to the number of time periods of lag (default is 1).

Details

The function `diff_it()` calculates the row differences between rows that are `lagg` rows apart. The leading or trailing stub periods are padded with *zeros*. Positive `lagg` means that the difference is calculated as the current row minus the row that is `lagg` rows above. (vice versa negative `lagg`). This also applies to vectors, since they can be viewed as single-column matrices.

Value

A vector or matrix with the same dimensions as the input object.

Examples

```
# diff vector by 2 periods
rutils::diff_it(1:10, lag=2)
# diff matrix by negative 2 periods
rutils::diff_it(matrix(1:10, ncol=2), lag=-2)
```

diff_ohlc — *Calculate the reduced form of an* OHLC *time series, or calculate the standard form from the reduced form of an* OHLC *time series.*

Description

Calculate the reduced form of an *OHLC* time series, or calculate the standard form from the reduced form of an *OHLC* time series.

Usage

```
diff_ohlc(oh_lc, re_duce = TRUE, ...)
```

Arguments

oh_lc	An *OHLC* time series of prices in *xts* format.
re_duce	*Boolean* argument: should the reduced form be calculated or the standard form? (default is TRUE)
...	additional arguments to function xts::diff.xts().

Details

The reduced form of an *OHLC* time series is obtained by calculating the time differences of its *Close* prices, and by calculating the differences between its *Open*, *High*, and *Low* prices minus the *Close* prices. The standard form is the original *OHLC* time series, and can be calculated from its reduced form by reversing those operations.

Value

An *OHLC* time series with five columns for the *Open*, *High*, *Low*, *Close* prices, and the *Volume*, and with the same time index as the input series.

Examples

```
# calculate reduced form of an OHLC time series
diff_VTI <- rutils::diff_ohlc(rutils::env_etf$VTI)
# calculate standard form of an OHLC time series
VTI <- rutils::diff_ohlc(diff_VTI, re_duce=FALSE)
identical(VTI, rutils::env_etf$VTI[, 1:5])
```

diff_xts — *Calculate the time differences of an* xts *time series.*

Description

Calculate the time differences of an *xts* time series.

Usage

```
diff_xts(x_ts, lagg = 1, ...)
```

Arguments

x_ts	An *xts* time series.
lagg	An integer equal to the number of time periods of lag (default is 1).
...	additional arguments to function xts::diff.xts().

Details

The function diff_xts() calculates the time differences of an *xts* time series and pads with *zeros* instead of *NAs*. Positive lagg means differences are calculated with values from lagg periods in the past (vice versa negative lagg). The function diff() is just a wrapper for diff.xts() from package xts, but it pads with *zeros* instead of *NAs*.

Value

An *xts* time series with the same dimensions and the same time index as the input series.

Examples

```
# calculate time differences over lag by 10 periods
rutils::diff_xts(rutils::env_etf$VTI, lag=10)
```

---

do_call *Recursively apply a function to a list of objects, such as* xts *time series.*

Description

Performs a similar operation as do.call(), but using recursion, which is much faster and uses less memory. The function do_call() is a generalization of function do_call_rbind().

Usage

```
do_call(func_tion, li_st, ...)
```

Arguments

func_tion	The name of function that returns a single object from a list of objects.
li_st	A list of objects, such as vectors, matrices, data frames, or time series.
...	additional arguments to function func_tion().

Details

The function do_call() performs an lapply loop, each time binding neighboring elements and dividing the length of li_st by half. The result of performing do_call(rbind, list_xts) on a list of *xts* time series is identical to performing do.call(rbind, list_xts). But do.call(rbind, list_xts) is very slow, and often causes an 'out of memory' error.

Value

A single vector, matrix, data frame, or time series.

Examples

```
# create xts time series
x_ts <- xts(x=rnorm(1000), order.by=(Sys.time()-3600*(1:1000)))
# split time series into daily list
list_xts <- split(x_ts, "days")
# rbind the list back into a time series and compare with the original
identical(x_ts, rutils::do_call(rbind, list_xts))
```

`do_call_assign` *Apply a function to a list of objects, merge the outputs into a single object, and assign the object to the output environment.*

Description

Apply a function to a list of objects, merge the outputs into a single object, and assign the object to the output environment.

Usage

```
do_call_assign(func_tion, sym_bols = NULL, out_put, env_in = .GlobalEnv,
  env_out = .GlobalEnv, ...)
```

Arguments

`func_tion`	The name of a function that returns a single object (`vector`, *xts* time series, etc.)
`sym_bols`	A vector of *character* strings with the names of input objects.
`out_put`	The string with name of output object.
`env_in`	The environment containing the input `sym_bols`.
`env_out`	The environment for creating the `out_put`.
`...`	additional arguments to function `func_tion()`.

Details

The function `do_call_assign()` performs an lapply loop over `sym_bols`, applies the function `func_tion()`, merges the outputs into a single object, and creates the object in the environment `env_out`. The output object is created as a side effect, while its name is returned invisibly.

Value

A single object (`matrix`, *xts* time series, etc.)

Examples

```
new_env <- new.env()
rutils::do_call_assign(
   func_tion=get_col,
   sym_bols=rutils::env_etf$sym_bols,
   out_put="price_s",
   env_in=env_etf, env_out=new_env)
```

do_call_rbind — *Recursively 'rbind' a list of objects, such as xts time series.*

Description

Recursively 'rbind' a list of objects, such as *xts* time series.

Usage

```
do_call_rbind(li_st)
```

Arguments

li_st	A list of objects, such as vectors, matrices, data frames, or time series.

Details

Performs lapply loop, each time binding neighboring elements and dividing the length of li_st by half. The result of performing do_call_rbind(list_xts) on a list of *xts* time series is identical to performing do.call(rbind, list_xts). But do.call(rbind, list_xts) is very slow, and often causes an 'out of memory' error.

The function do_call_rbind() performs the same operation as do.call(rbind, li_st), but using recursion, which is much faster and uses less memory. This is the same function as 'do.call.rbind' from package 'qmao'.

Value

A single vector, matrix, data frame, or time series.

Examples

```
# create xts time series
x_ts <- xts(x=rnorm(1000), order.by=(Sys.time()-3600*(1:1000)))
# split time series into daily list
list_xts <- split(x_ts, "days")
# rbind the list back into a time series and compare with the original
identical(x_ts, rutils::do_call_rbind(list_xts))
```

etf_data — *The etf_data dataset contains a single environment called env_etf, which includes daily OHLC time series data for a portfolio of symbols.*

Description

The env_etf environment includes daily OHLC time series data for a portfolio of symbols, and reference data:

sym_bols a vector of strings with the portfolio symbols.

price_s a single xts time series containing daily closing prices for all the sym_bols.

re_turns a single xts time series containing daily returns for all the sym_bols.

Individual time series "VTI", "VEU", etc., containing daily OHLC prices for the sym_bols.

Usage

```
data(etf_data)  # not required - data is lazy load
```

Format

Each `xts` time series contains the columns:

Open Open prices

High High prices

Low Low prices

Close Close prices

Volume daily trading volume

Adjusted Adjusted closing prices

Examples

```
# data(etf_data)  # not needed - data is lazy load
# get first six rows of OHLC prices
head(env_etf$VTI)
chart_Series(x=env_etf$VTI["2009-11"])
```

---

get_col — *Extract columns of data from* OHLC *time series using column field names.*

Description

Extract columns of data from *OHLC* time series using column field names.

Usage

```
get_col(oh_lc, field_name = "Close", data_env = NULL)
```

Arguments

`oh_lc`	An *OHLC* time series in *xts* format, or a vector of *character* strings with the names of *OHLC* time series.
`field_name`	A vector of strings with the field names of the columns to be be extracted (default is *"Close"*).
`data_env`	The environment containing *OHLC* time series (default is *NULL*).

Details

The function `get_col()` extracts columns from *OHLC* time series and binds them into a single *xts* time series. `get_col()` can extract columns from a single *xts* time series, or from multiple time series.

The function `get_col()` extracts columns by partially matching field names with column names. The *OHLC* column names are assumed to be in the format `"symbol.field_name"`, for example `"VTI.Close"`.

In the simplest case when oh_lc is a single *xts* time series and *field_name* is a single string, the function get_col() performs a similar operation to the extractor functions Op(), Hi(), Lo(), Cl(), and Vo(), from package quantmod. But get_col() is able to handle symbols like *LOW*, which the function Lo() can't handle. The field_name argument is partially matched, for example "Vol" is matched to Volume (but it's case sensitive).

In the case when oh_lc is a vector of strings with the names of *OHLC* time series, the function get_col() reads the *OHLC* time series from the environment data_env, extracts the specified columns, and binds them into a single *xts* time series.

Value

The specified columns of the *OHLC* time series bound into a single *xts* time series, with the same number of rows as the input time series.

Examples

```
# get close prices for VTI
rutils::get_col(rutils::env_etf$VTI)
# get volumes for VTI
rutils::get_col(rutils::env_etf$VTI, field_name="Vol")
# get close prices and volumes for VTI
rutils::get_col(rutils::env_etf$VTI, field_name=c("Cl", "Vol"))
# get close prices and volumes for VTI and IEF
rutils::get_col(oh_lc=c("VTI", "IEF"), field_name=c("Cl", "Vol"),
  data_env=rutils::env_etf)
```

---

get_data — *Load* OHLC *time series data into an environment, either from an external source (download from* YAHOO*), or from* CSV *files in a local drive.*

Description

Load *OHLC* time series data into an environment, either from an external source (download from *YAHOO*), or from *CSV* files in a local drive.

Usage

```
get_data(sym_bols, data_dir = NULL, data_env, start_date = "2007-01-01",
  end_date = Sys.Date(), date_fun = match.fun("as.Date"),
  for_mat = "%Y-%m-%d", header = TRUE, e_cho = TRUE, scrub = TRUE)
```

Arguments

sym_bols	A vector of strings representing instrument symbols (tickers).
data_dir	The directory containing *CSV* files (default is NULL).
data_env	The environment for loading the data into.
start_date	The start date of time series data (default is "2007-01-01").
end_date	The end date of time series data (default is Sys.Date()).
date_fun	The name of the function for formatting the date fields in the *CSV* files (default is as.Date()).

for_mat	The format of the date fields in the *CSV* files (default is %Y-%m-%d).
header	*Boolean* argument: if TRUE then read the header in the *CSV* files (default is TRUE).
e_cho	*Boolean* argument: if TRUE then print to console information on the progress of *CSV* file loading (default is TRUE).
scrub	*Boolean* argument: if TRUE then remove NA values using function rutils::na_locf() (default is TRUE).

Details

The function get_data() loads *OHLC* time series data into an environment (as a side-effect), and returns invisibly the vector of sym_bols.

If the argument data_dir is specified, then get_data() loads from *CSV* files in that directory, and overwrites NA values if scrub=TRUE. If the argument data_dir is *not* specified, then get_data() downloads adjusted *OHLC* prices from *YAHOO*.

The function get_data() calls the function getSymbols.yahoo() for downloading data from *YAHOO*, and performs a similar operation to the function getSymbols() from package quantmod. But get_data() is faster because it performs less overhead operations, and it's able to handle symbols like *LOW*, which getSymbols() can't handle because the function Lo() can't handle them. The start_date and end_date must be either of class *Date*, or a string in the format *"YYYY-mm-dd"*.

Value

A vector of sym_bols returned invisibly.

Examples

```
## Not run:
new_env <- new.env()
# load prices from local csv files
rutils::get_data(sym_bols=c("SPY", "EEM"),
            data_dir="C:/Develop/data_bbg_records",
            data_env=new_env)
# download prices from YAHOO
rutils::get_data(sym_bols=c("MSFT", "XOM"),
            data_env=new_env,
            start_date="2012-12-01",
            end_date="2015-12-01")

## End(Not run)
```

---

get_name *Extract symbol names (tickers) from a vector of* character *strings.*

Description

Extract symbol names (tickers) from a vector of *character* strings.

Usage

```
get_name(str_ing, sepa_rator = "[.]", field = 1)
```

Arguments

str_ing	A vector of *character* strings containing symbol names.
sepa_rator	The name separator, i.e. the single *character* that separates the symbol name from the rest of the string (default is "[.]").
field	The position of the name in the string, i.e. the integer index of the field to be extracted (default is 1, i.e. the name is at the beginning of the string,)

Details

The function get_name() extracts the symbol names (tickers) from a vector of *character* strings. If the input is a vector of strings, then get_name() returns a vector of names.

The input string is assumed to be in the format "*name*.csv", with the name at the beginning of the string, but get_name() can also parse the name from other string formats as well. For example, it extracts the name "VTI" from the string "VTI.Close", or it extracts the name "XLU" from the string "XLU_2017_09_05.csv" (with sepa_rator="_").

Value

A vector of *character strings* containing symbol names.

Examples

```
# extract symbols "XLU" and"XLP" from file names
rutils::get_name(c("XLU.csv", "XLP.csv"))
# extract symbols from file names
rutils::get_name("XLU_2017_09_05.csv", sep="_")
rutils::get_name("XLU 2017 09 05.csv", sep=" ")
# extract fields "Open", "High", "Low", "Close" from column names
rutils::get_name(colnames(rutils::env_etf$VTI), field=2)
```

---

lag_it *Apply a lag to a* numeric *vector or matrix.*

Description

Apply a lag to a *numeric* vector or matrix.

Usage

```
lag_it(in_put, lagg = 1)
```

Arguments

in_put	A *numeric* vector or matrix.
lagg	An integer equal to the number of time periods of lag (default is 1).

Details

Applies a lag to a vector or matrix, by shifting its values by a certain number of rows, equal to the integer lagg, and pads the leading or trailing stub periods with *zeros*. Positive lagg means that values in the current row are replaced with values from the row that are lagg rows above. (vice versa negative lagg). This also applies to vectors, since they can be viewed as single-column matrices.

Value

A vector or matrix with the same dimensions as the input object.

Examples

```
# lag vector by 2 periods
rutils::lag_it(1:10, lag=2)
# lag matrix by negative 2 periods
rutils::lag_it(matrix(1:10, ncol=2), lag=-2)
```

---

lag_xts *Apply a time lag to an* xts *time series.*

Description

Apply a time lag to an *xts* time series.

Usage

```
lag_xts(x_ts, lagg = 1, ...)
```

Arguments

x_ts	An *xts* time series.
lagg	An integer equal to the number of time periods of lag (default is 1).
...	additional arguments to function xts::lag_xts().

Details

Applies a time lag to an *xts* time series and pads with the first and last values instead of *NAs*.

A positive lag argument lagg means values from lagg periods in the past are moved to the present. A negative lag argument lagg moves values from the future to the present. The function lag_xts() is just a wrapper for function lag.xts() from package xts, but it pads with the first and last values instead of *NAs*.

Value

An *xts* time series with the same dimensions and the same time index as the input x_ts time series.

Examples

```
# lag by 10 periods
rutils::lag_xts(rutils::env_etf$VTI, lag=10)
```

na_locf *Replace* NA *values with the most recent* non-NA *values prior to them.*

Description

Replace *NA* values with the most recent *non-NA* values prior to them.

Usage

```
na_locf(in_put, from_last = FALSE, na_rm = FALSE, max_gap = NROW(in_put))
```

Arguments

in_put	A *numeric* or *Boolean* vector or matrix, or *xts* time series.
from_last	*Boolean* argument: should *non-NA* values be carried backward rather than forward? (default is FALSE)
na_rm	*Boolean* argument: should any remaining (leading or trailing) *NA* values be removed? (default is FALSE)
max_gap	The maximum number of neighboring *NA* values that can be replaced (default is NROW(in_put)).

Details

The function na_locf() replaces *NA* values with the most recent *non-NA* values prior to them.

If the from_last argument is FALSE (the default), then the previous or past *non-NA* values are carried forward to replace the *NA* values. If the from_last argument is TRUE, then the following or future *non-NA* values are carried backward to replace the *NA* values.

The function na_locf() performs the same operation as function na.locf() from package zoo, but it also accepts vectors as input.

The function na_locf() calls the compiled function na_locf() from package xts, which allows it to perform its calculations about three times faster than na.locf().

Value

A vector, matrix, or *xts* time series with the same dimensions and data type as the argument in_put.

Examples

```
# create vector containing NA values
in_put <- sample(22)
in_put[sample(NROW(in_put), 4)] <- NA
# replace NA values with the most recent non-NA values
rutils::na_locf(in_put)
# create matrix containing NA values
in_put <- sample(44)
in_put[sample(NROW(in_put), 8)] <- NA
in_put <- matrix(in_put, nc=2)
# replace NA values with the most recent non-NA values
rutils::na_locf(in_put)
# create xts series containing NA values
in_put <- xts::xts(in_put, order.by=seq.Date(from=Sys.Date(),
```

```
  by=1, length.out=NROW(in_put)))
# replace NA values with the most recent non-NA values
rutils::na_locf(in_put)
```

`roll_max` *Calculate the rolling maximum of an* xts *time series over a sliding window (lookback period).*

Description

Calculate the rolling maximum of an *xts* time series over a sliding window (lookback period).

Usage

```
roll_max(x_ts, look_back)
```

Arguments

`x_ts`	An *xts* time series containing one or more columns of data.
`look_back`	The size of the lookback window, equal to the number of data points for calculating the rolling sum.

Details

For example, if look_back=3, then the rolling sum at any point is equal to the sum of `x_ts` values for that point plus two preceding points.

The initial values of roll_max() are equal to cumsum() values, so that roll_max() doesn't return any NA values.

The function `roll_max()` performs the same operation as function `runMax()` from package TTR, but using vectorized functions, so it's a little faster.

Value

An *xts* time series with the same dimensions as the input series.

Examples

```
# create xts time series
x_ts <- xts(x=rnorm(1000), order.by=(Sys.time()-3600*(1:1000)))
rutils::roll_max(x_ts, look_back=3)
```

roll_sum *Calculate the rolling sum of a* numeric *vector, matrix, or* xts *time series over a sliding window (lookback period).*

Description

Calculate the rolling sum of a *numeric* vector, matrix, or *xts* time series over a sliding window (lookback period).

Usage

```
roll_sum(x_ts, look_back)
```

Arguments

x_ts	A vector, matrix, or *xts* time series containing one or more columns of data.
look_back	The size of the lookback window, equal to the number of data points for calculating the rolling sum.

Details

For example, if look_back=3, then the rolling sum at any point is equal to the sum of x_ts values for that point plus two preceding points. The initial values of roll_sum() are equal to cumsum() values, so that roll_sum() doesn't return any NA values.

The function roll_sum() performs the same operation as function runSum() from package TTR, but using vectorized functions, so it's a little faster.

Value

A vector, matrix, or *xts* time series with the same dimensions as the input series.

Examples

```
# rolling sum of vector
vec_tor <- rnorm(1000)
rutils::roll_sum(vec_tor, look_back=3)
# rolling sum of matrix
mat_rix <- matrix(rnorm(1000), nc=5)
rutils::roll_sum(mat_rix, look_back=3)
# rolling sum of xts time series
x_ts <- xts(x=rnorm(1000), order.by=(Sys.time()-3600*(1:1000)))
rutils::roll_sum(x_ts, look_back=3)
```

sub_set *Subset an* xts *time series (extract an* xts *sub-series corresponding to the input dates).*

Description

Subset an *xts* time series (extract an *xts* sub-series corresponding to the input dates).

Usage

```
sub_set(x_ts, start_date, end_date, get_rows = TRUE)
```

Arguments

x_ts	An *xts* time series.
start_date	The start date of the extracted time series data.
end_date	The end date of the extracted time series data, or the number of data rows to be extracted.
get_rows	*Boolean* argument: if TRUE then extract the given number of rows of data, else extract the given number of calendar days (default is TRUE).

Details

The function sub_set() extracts an *xts* sub-series corresponding to the input dates. If end_date is a date object or a character string representing a date, then sub_set() performs standard bracket subsetting using the package xts.

The rows of data don't necessarily correspond to consecutive calendar days because of weekends and holidays. For example, 10 consecutive rows of data may correspond to 12 calendar days. So if end_date is a number, then we must choose to extract either the given number of rows of data (get_rows=TRUE) or the given number of calendar days (get_rows=FALSE).

If end_date is a positive number then sub_set() returns the specified number of data rows from the future, and if it's negative then it returns the data rows from the past.

If end_date is a number, and either start_date or end_date are outside the date range of x_ts, then sub_set() extracts the maximum available range of x_ts.

Value

An *xts* time series with the same number of columns as the input time series.

Examples

```
# subset an xts time series using two dates
rutils::sub_set(rutils::env_etf$VTI, start_date="2015-01-01", end_date="2015-01-10")
# extract 6 consecutive rows of data from the past, using a date and a negative number
rutils::sub_set(rutils::env_etf$VTI, start_date="2015-01-01", end_date=-6)
# extract 6 calendar days of data
rutils::sub_set(rutils::env_etf$VTI, start_date="2015-01-01", end_date=6, get_rows=FALSE)
# extract up to 100 consecutive rows of data
rutils::sub_set(rutils::env_etf$VTI, start_date="2016-08-01", end_date=100)
```

`to_period` *Aggregate an* OHLC *time series to a lower periodicity.*

Description

Given an *OHLC* time series at high periodicity (say seconds), calculates the *OHLC* prices at lower periodicity (say minutes).

Usage

```
to_period(oh_lc, period = "minutes", k = 1,
  end_points = xts::endpoints(oh_lc, period, k))
```

Arguments

`oh_lc`	An *OHLC* time series of prices in *xts* format.
`period`	aggregation interval ("seconds", "minutes", "hours", "days", "weeks", "months", "quarters", and "years").
`k`	The number of periods to aggregate over (for example if period="minutes" and k=2, then aggregate over two minute intervals.)
`end_points`	An integer vector of end points.

Details

The function `to_period()` performs a similar aggregation as function `to.period()` from package xts, but has the flexibility to aggregate to a user-specified vector of end points. The function `to_period()` simply calls the compiled function `toPeriod()` (from package xts), to perform the actual aggregations. If `end_points` are passed in explicitly, then the `period` argument is ignored.

Value

A *OHLC* time series of prices in *xts* format, with a lower periodicity defined by the end_points.

Examples

```
## Not run:
# define end points at 10-minute intervals (HighFreq::SPY is minutely bars)
end_points <- rutils::calc_endpoints(HighFreq::SPY["2009"], inter_val=10)
# aggregate over 10-minute end_points:
rutils::to_period(oh_lc=HighFreq::SPY["2009"], end_points=end_points)
# aggregate over days:
rutils::to_period(oh_lc=HighFreq::SPY["2009"], period="days")
# equivalent to:
to.period(x=HighFreq::SPY["2009"], period="days", name=rutils::get_name(colnames(HighFreq::SPY)[1])

## End(Not run)
```

Index