

Package ‘rutils’

May 8, 2024

Type Package

Title Utility Functions for Simplifying Financial Data Management and Modeling

Version 0.2

Date 2018-09-12

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 MPL-2.0

Depends xts,
quantmod,
dygraphs

Imports xts,
quantmod,
dygraphs

Suggests knitr,
rmarkdown,
testthat

VignetteBuilder knitr

LazyData true

ByteCompile true

Repository GitHub

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

RoxygenNote 7.2.3

R topics documented:

adjust_ohlc	2
calc_endpoints	3
calc_sharpe	4
calc_skew	5
chart_dygraph	6
chart_dygraph2y	6
chart_xts	7

chart_xts2y	8
diffit	9
diffohl	10
diffxts	10
do_call	11
do_call_assign	12
do_call_rbind	13
etf_data	14
getpoly	14
get_col	16
get_data	17
get_name	18
lagit	19
lagxts	20
nalocf	21
plot_acf	22
roll_max	23
roll_sum	23
sub_set	24
to_period	25

adjust_ohlc	<i>Adjust the first four columns of OHLC data using the "adjusted" price column.</i>
-------------	--

Description

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

Usage

```
adjust_ohlc(ohlc)
```

Arguments

ohlc 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::etfenv$VTI)
```

calc_endpoints	<i>Calculate a vector of equally spaced end points along the elements of a vector, matrix, or time series.</i>
----------------	--

Description

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

Usage

```
calc_endpoints(xtsv, interval, stub_front = TRUE)
```

Arguments

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

Details

The end points are a vector of integers which divide the elements (rows) of xtsv into equally spaced intervals.

If interval is an *integer* then calc_endpoints() calculates the number of whole intervals that fit over the elements (rows) of xtsv. If a whole number of intervals doesn't fit over the elements (rows) of xtsv, then calc_endpoints() adds a stub interval either at the beginning (the default) or at the end.

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

The function calc_endpoints() is a generalization of function endpoints() from package **xts**, since interval can accept both *integer* and *string* values. Similar to xts::endpoints(), the first integer returned by calc_endpoints() is equal to zero.

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, interval=11)
# Calculate end points with a stub interval at the end
rutils::calc_endpoints(rutils::etfenv$VTI, interval=365, stub_front=FALSE)
# Calculate end points at the end of every hour
rutils::calc_endpoints(rutils::etfenv$VTI, interval="hours")
```

calc_sharpe	<i>Calculate the Sharpe and Sortino ratios of a time series of returns.</i>
-------------	---

Description

Calculate the Sharpe and Sortino ratios of a time series of returns.

Usage

```
calc_sharpe(retp, riskf = 0, nperiods = 252)
```

Arguments

retp	A time series of returns, with multiple columns.
riskf	The annual risk-free interest rate (the default is 0).
nperiods	The number of time periods in a year (the default is 252 days).

Details

The function `calc_sharpe()` calculates the Sharpe and Sortino ratios of a time series of returns.

The function `calc_sharpe()` performs an `sapply()` loop over the columns of the `retp` argument. It calculates the Sharpe and Sortino ratios for each column. It subtracts from `retp` the annual risk-free interest rate `riskf` divided by `nperiods`. It multiplies the Sharpe and Sortino ratios by the square root of `nperiods`, in order to obtain the annual ratios.

The Sharpe ratio S_r is defined as:

$$S_r = \sqrt{n} \frac{\bar{r}}{\sigma_r}$$

Where r are the daily excess returns (the returns minus the risk-free rate), \bar{r} are the average excess returns, and σ_r is their daily standard deviation.

The Sortino ratio So_r is defined as:

$$So_r = \sqrt{n} \frac{\bar{r}}{\sigma_d}$$

Where r are the daily excess returns (the returns minus the risk-free rate), \bar{r} are the average excess returns, and σ_d is their daily *downside deviation*. The *downside deviation* σ_d is equal to the standard deviation of the downside returns r_d , the returns that are less than the risk-free rate.

Value

A matrix of the Sharpe and Sortino ratios.

Examples

```
# Calculate the Sharpe and Sortino ratios of VTI and IEF returns
rutils::calc_sharpe(rutils::etfenv$returns[, c("VTI", "IEF")])
```

calc_skew

*Calculate the skewness or kurtosis of a time series of returns.***Description**

Calculate the skewness or kurtosis of a time series of returns.

Usage

```
calc_skew(retp, expn = 3)
```

Arguments

retp A time series of returns, with multiple columns.

expn The power (exponent) to raise the returns (the default is 3 for skewness).

Details

The function `calc_skew()` calculates the skewness or kurtosis of a time series of returns.

The function `calc_skew()` performs an `sapply()` loop over the columns of the `retp` argument. It raise the returns to the power `expn`. If `expn = 3` it calculates the skewness. If `expn = 4` it calculates the kurtosis.

The skewness ς is defined as:

$$\varsigma = \frac{1}{n-1} \sum_{i=1}^n \left(\frac{r - \bar{r}}{\sigma} \right)^3$$

Where r are the daily returns, \bar{r} are the average returns, and σ is their standard deviation.

The kurtosis κ is defined as:

$$\kappa = \frac{1}{n-1} \sum_{i=1}^n \left(\frac{r - \bar{r}}{\sigma} \right)^4$$

Value

A matrix of the skewness or kurtosis values.

Examples

```
# Calculate the kurtosis of VTI and IEF returns
rutils::calc_skew(rutils::etfenv$returns[, c("VTI", "IEF")], expn=4)
```

chart_dygraph	<i>Plot an interactive dygraphs candlestick plot with background shading for an OHLC time series in xts format.</i>
---------------	---

Description

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

Usage

```
chart_dygraph(ohlc, indic = NULL, ...)
```

Arguments

ohlc	An <i>OHLC</i> time series in <i>xts</i> format.
indic	A <i>Boolean</i> time series in <i>xts</i> format for specifying the shading areas, with <code>TRUE</code> indicating "lightgreen" shading, and <code>FALSE</code> indicating "antiquewhite" shading (the default is <code>NULL</code>).
...	Additional arguments to function <code>dygraphs::dygraph()</code> .

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
ohlc <- rutils::etfenv$VTI
vwapv <- TTR::VWAP(price=quantmod::Ad(ohlc), volume=quantmod::Vo(ohlc), n=20)
ohlc <- cbind(ohlc[, c(1:3, 6)], vwapv)["2009-02/2009-04"]
rutils::chart_dygraph(ohlc, indic=(ohlc[, 4] > vwapv))
```

chart_dygraph2y	<i>Plot an interactive dygraphs line plot for two xts time series, with two "y" axes.</i>
-----------------	---

Description

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

Usage

```
chart_dygraph2y(xtsv, ...)
```

Arguments

xtsv 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
prices <- cbind(Ad(rutils::etfenv$VTI), Ad(rutils::etfenv$IEF))
colnames(prices) <- get_name(colnames(prices), posv=2)
rutils::chart_dygraph2y(prices)
```

chart_xts	<i>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.</i>
-----------	--

Description

A wrapper for function `chart_Series()` from package **quantmod**.

Usage

```
chart_xts(xtsv, colors = NULL, ylim = NULL, indic = NULL, x11 = TRUE, ...)
```

Arguments

xtsv An *xts* time series or an *OHLC* time series.

colors A vector of *strings* with the custom line colors.

ylim A *numeric* vector with two elements containing the y-axis range.

indic A *Boolean* vector or *xts* time series for specifying the shading areas, with TRUE indicating "lightgreen" shading, and FALSE indicating "antiquewhite" shading.

x11 A *Boolean* argument: if TRUE then open x11 window for plotting, else plot in standard window (the 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 *xtsv* 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 *indic* argument, using function `add_TA()`. The *indic* argument should have the same length as the *xtsv* 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::etfenv$VTI["2015-11"],
  name="VTI in Nov 2015", ylim=c(102, 108),
  indic=zoo::index(rutils::etfenv$VTI["2015-11"]) > as.Date("2015-11-18"))
# Plot two time series with custom line colors
rutils::chart_xts(na.omit(cbind(rutils::etfenv$XLU[, 4],
  rutils::etfenv$XLP[, 4])), colors=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(xtsv, color = "red", x11 = TRUE, ...)
```

Arguments

<code>xtsv</code>	An <i>xts</i> time series with two columns.
<code>color</code>	A string specifying the color of the second line and axis (the default is "red").
<code>x11</code>	A <i>Boolean</i> argument: if TRUE then open <code>x11</code> window for plotting, else plot in standard window (the default is TRUE).
<code>...</code>	Additional arguments to function <code>plot.zoo()</code> .

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 *xtsv* 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::etfenv$VTI),
                             quantmod::Cl(rutils::etfenv$IEF))["2015"])
```

diffit	<i>Calculate the row differences of a numeric or Boolean vector, matrix, or xts time series.</i>
--------	--

Description

Calculate the row differences of a *numeric* or *Boolean* vector, matrix, or *xts* time series.

Usage

```
diffit(inputv, lagg = 1, ...)
```

Arguments

inputv	A <i>numeric</i> or <i>Boolean</i> vector or matrix, or <i>xts</i> time series.
lagg	An integer equal to the number of time periods of lag (the default is 1).

Details

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

When applied to *xts* time series, the function `diffit()` calls the function `diff.xts()` from package **xts**, but it pads the output with zeros instead of with NAs.

Value

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

Examples

```
# Diff vector by 2 periods
rutils::diffit(1:10, lagg=2)
# Diff matrix by negative 2 periods
rutils::diffit(matrix(1:10, ncol=2), lagg=-2)
# Diff an xts time series
rutils::diffit(rutils::etfenv$VTI, lagg=10)
```

diffohlc	<i>Calculate the reduced form of an OHLC time series, or calculate the standard form from the reduced form of an OHLC time series.</i>
----------	--

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

```
diffohlc(ohlc, reducit = TRUE, ...)
```

Arguments

ohlc	An <i>OHLC</i> time series of prices in <i>xts</i> format.
reducit	A <i>Boolean</i> argument: should the reduced form be calculated or the standard form? (the default is TRUE)
...	Additional arguments to function <code>xts::diff.xts()</code> .

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
diffVTI <- rutils::diffohlc(rutils::etfenv$VTI)
# Calculate standard form of an OHLC time series
VTI <- rutils::diffohlc(diffVTI, reducit=FALSE)
identical(VTI, rutils::etfenv$VTI[, 1:5])
```

diffxts	<i>Calculate the time differences of an xts time series.</i>
---------	--

Description

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

Usage

```
diffxts(xtsv, lagg = 1, ...)
```

Arguments

<code>xtsv</code>	An <i>xts</i> time series.
<code>lagg</code>	An integer equal to the number of time periods of lag (the default is 1).
<code>...</code>	Additional arguments to function <code>xts::diff.xts()</code> .

Details

The function `diffxts()` 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 for a negative `lagg`). The function `diff()` is just a wrapper for `diff.xts()` from package *xts*, but it pads with *zeros* instead of NAs.

The function `diffit()` has incorporated the functionality of `diffxts()`, so that `diffxts()` will be retired in future package versions.

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::diffxts(rutils::etfenv$VTI, lag=10)
```

<code>do_call</code>	<i>Recursively apply a function to a list of objects, such as xts time series.</i>
----------------------	--

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, listv, ...)
```

Arguments

<code>func</code>	The name of function that returns a single object from a list of objects.
<code>listv</code>	A list of objects, such as <i>vectors</i> , <i>matrices</i> , <i>data frames</i> , or <i>time series</i> .
<code>...</code>	Additional arguments to function <code>func()</code> .

Details

The function `do_call()` performs an lapply loop, each time binding neighboring elements and dividing the length of `listv` 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
xtsv <- xts(x=rnorm(1000), order.by=(Sys.time()-3600*(1:1000)))
# Split time series into daily list
list_xts <- split(xtsv, "days")
# rbind the list back into a time series and compare with the original
identical(xtsv, 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,
  symbolv = NULL,
  outv,
  inenv = .GlobalEnv,
  outenv = .GlobalEnv,
  ...
)
```

Arguments

func	The name of a function that returns a single object (<i>vector</i> , <i>xts</i> time series, etc.)
symbolv	A vector of <i>character</i> strings with the names of input objects.
outv	The string with name of output object.
inenv	The environment containing the input symbolv.
outenv	The environment for creating the outv.
...	Additional arguments to function func().

Details

The function `do_call_assign()` performs an `lapply` loop over `symbolv`, applies the function `func()`, merges the outputs into a single object, and creates the object in the environment `outenv`. 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

```
newenv <- new.env()
rutils::do_call_assign(
  func=get_col,
  symbolv=rutils::etfenv$symbolv,
  outv="prices",
  inenv=etfenv, outenv=newenv)
```

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(listv)
```

Arguments

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

etf_data	<i>The etf_data dataset contains a single environment called etfenv, which includes daily OHLC time series data for a portfolio of symbols. All the prices are already adjusted.</i>
----------	--

Description

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

symbolv a vector of strings with the portfolio symbols.

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

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

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

Usage

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

Format

Each xts time series contains the following columns with adjusted prices and trading volume:

Open Open prices

High High prices

Low Low prices

Close Close prices

Volume daily trading volume

Examples

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

getpoly	<i>Download an OHLC time series of prices from Polygon.</i>
---------	---

Description

Download an *OHLC* time series of prices from Polygon.

Usage

```
getpoly(
  symbol = "SPY",
  startd = as.Date("1997-01-01"),
  endd = Sys.Date(),
  tspan = "day",
  apikey
)
```

Arguments

symbol	The stock symbol (ticker).
startd	The start date (the default is "1997-01-01").
endd	The end date (the default is Sys.Date()).
tspan	The data frequency, i.e. the time span for data aggregations (the default is "day" for daily data).
apikey	The API key issued by Polygon.

Details

The function `getpoly()` downloads historical prices from **Polygon**, and returns an *OHLC* time series of class *xts*.

Polygon is a provider of live and historical prices for stocks, options, foreign currencies, and cryptocurrencies.

The function `getpoly()` sends a request for data to the **Polygon** rest API, using the function `read_json()` from package **jsonlite**. The query requires an API key issued by **Polygon**. The API key must be passed to the argument `apikey`.

Polygon returns data in *JSON* format, which is then formatted into an *OHLC* time series of class *xts*.

The argument `tspan` determines the data frequency, i.e. it's the time span for data aggregations. The default is "day" for daily data. Other possible values of `tspan` are "minute", "hour", "week", and "month".

Value

An *OHLC* time series of class *xts*.

Examples

```
## Not run:
# Polygon API key - user must obtain their own key
apikey <- "0Q2f7j8CwAbdY5M8VYt_8pwdP0V4TunxbvRVC_"
# Download SPY prices from Polygon
ohlc <- rutils::getpoly(symbol="SPY", apikey=apikey)
# Plot candlesticks of SPY OHLC prices
library(highcharter)
highcharter::highchart(type="stock") %>% hc_add_series(ohlc, type="candlestick")

## End(Not run)
```

get_col	<i>Extract columns of data from OHLC time series using column field names.</i>
---------	--

Description

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

Usage

```
get_col(ohlc, fieldn = "Close", datenv = NULL)
```

Arguments

ohlc	An <i>OHLC</i> time series in <i>xts</i> format, or a vector of <i>character</i> strings with the names of <i>OHLC</i> time series.
fieldn	A vector of strings with the field names of the columns to be extracted (the default is "Close").
datenv	The environment containing <i>OHLC</i> time series (the 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.fieldn"`, for example `"VTI.Close"`.

In the simplest case when `ohlc` is a single *xts* time series and `fieldn` 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 `fieldn` argument is partially matched, for example "Vol" is matched to Volume (but it's case sensitive).

In the case when `ohlc` is a vector of strings with the names of *OHLC* time series, the function `get_col()` reads the *OHLC* time series from the environment `datenv`, 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::etfenv$VTI)
# get volumes for VTI
rutils::get_col(rutils::etfenv$VTI, fieldn="Vol")
# get close prices and volumes for VTI
rutils::get_col(rutils::etfenv$VTI, fieldn=c("Cl", "Vol"))
# get close prices and volumes for VTI and IEF
rutils::get_col(ohlc=c("VTI", "IEF"), fieldn=c("Cl", "Vol"),
  datenv=rutils::etfenv)
```

get_data	<i>Load OHLC time series data into an environment, either from an external source (download from YAHOO), or from CSV files in a local drive.</i>
----------	--

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(
  symbolv,
  datadir = NULL,
  datenv,
  startd = "2007-01-01",
  endd = Sys.Date(),
  func = match.fun("as.Date"),
  formatv = "%Y-%m-%d",
  header = TRUE,
  echo = TRUE,
  scrub = TRUE
)
```

Arguments

symbolv	A vector of strings representing instrument symbols (tickers).
datadir	The directory containing <i>CSV</i> files (the default is <code>NULL</code>).
datenv	The environment for loading the data into.
startd	The start date of time series data (the default is "2007-01-01").
endd	The end date of time series data (the default is <code>Sys.Date()</code>).
func	The name of the function for formatting the date fields in the <i>CSV</i> files (the default is <code>as.Date()</code>).
formatv	The format of the date fields in the <i>CSV</i> files (the default is <code>%Y-%m-%d</code>).
header	A <i>Boolean</i> argument: if <code>TRUE</code> then read the header in the <i>CSV</i> files (the default is <code>TRUE</code>).
echo	A <i>Boolean</i> argument: if <code>TRUE</code> then print to console information on the progress of <i>CSV</i> file loading (the default is <code>TRUE</code>).
scrub	A <i>Boolean</i> argument: if <code>TRUE</code> then remove NA values using function <code>rutils::nalocf()</code> (the default is <code>TRUE</code>).

Details

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

If the argument `datadir` is specified, then `get_data()` loads from *CSV* files in that directory, and overwrites NA values if `scrub=TRUE`. If the argument `datadir` 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 `startd` and `endd` must be either of class *Date*, or a string in the format "*YYYY-mm-dd*".

Value

A vector of `symbolv` returned invisibly.

Examples

```
## Not run:
newenv <- new.env()
# Load prices from local csv files
rutils::get_data(symbolv=c("SPY", "EEM"),
                  datadir="C:/Develop/data/bbg_records",
                  datenv=newenv)
# Download prices from YAHOO
rutils::get_data(symbolv=c("MSFT", "XOM"),
                  datenv=newenv,
                  startd="2012-12-01",
                  endd="2015-12-01")

## End(Not run)
```

get_name	<i>Extract symbol names (tickers) from a vector of character strings.</i>
----------	---

Description

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

Usage

```
get_name(strng, posv = 1, sep = "[.]")
```

Arguments

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

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 "XLU2017_09_05.csv" (with `sep="_"`).

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("XLU2017_09_05.csv", sep="_")
rutils::get_name("XLU 2017 09 05.csv", sep=" ")
# Extract fields "Open", "High", "Low", "Close" from the column names
rutils::get_name(c("VTI.Open", "VTI.High", "VTI.Low", "VTI.Close"), posv=2)
```

lagit

Apply a lag to a numeric or Boolean vector, matrix, or xts time series.

Description

Apply a lag to a *numeric* or *Boolean* vector, matrix, or *xts* time series.

Usage

```
lagit(inputv, lagg = 1, pad_zeros = TRUE, ...)
```

Arguments

inputv	A <i>numeric</i> or <i>Boolean</i> vector or matrix, or <i>xts</i> time series.
lagg	An integer equal to the number of time periods (rows) of lag (the default is 1).
pad_zeros	A <i>Boolean</i> argument: Should the output be padded with zeros? (The default is pad_zeros = TRUE.)

Details

The function `lagit()` applies a lag to the input object by shifting its rows by the number of time periods equal to the integer argument `lagg`.

For positive `lagg` values, the current row is replaced with the row that is `lagg` rows above it. And vice versa for a negative `lagg` values. This also applies to vectors, since they can be viewed as single-column matrices. If `lagg = 0`, then `lagit()` returns the input object unchanged.

To avoid leading or trailing NA values, the output object is padded with zeroes, or with elements from either the first or the last row.

For the lag of asset returns, they should be padded with zeros, to avoid look-ahead bias. For the lag of prices, they should be padded with the first or last prices, not with zeros.

When applied to *xts* time series, the function `lagit()` calls the function `lag.xts()` from package **xts**, but it pads the output with the first and last rows, instead of with NAs.

Value

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

Examples

```
# Lag vector by 2 periods
rutils::lagit(1:10, lag=2)
# Lag matrix by negative 2 periods
rutils::lagit(matrix(1:10, ncol=2), lag=-2)
# Lag an xts time series
lag_ged <- rutils::lagit(rutils::etfenv$VTI, lag=10)
```

lagxts

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

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

Usage

```
lagxts(xtsv, lagg = 1, pad_zeros = TRUE, ...)
```

Arguments

<code>xtsv</code>	An <i>xts</i> time series.
<code>lagg</code>	An integer equal to the number of time periods of lag (the default is 1).
<code>pad_zeros</code>	A <i>Boolean</i> argument: Should the output be padded with zeros? (The default is <code>pad_zeros = TRUE</code> .)
<code>...</code>	Additional arguments to function <code>xts::lagxts()</code> .

Details

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

A positive lag argument `lagg` means elements from `lagg` periods in the past are moved to the present. A negative lag argument `lagg` moves elements from the future to the present. If `lagg = 0`, then `lagxts()` returns the input time series unchanged.

To avoid leading or trailing NA values, the output *xts* is padded with zeroes, or with elements from either the first or the last row.

For the lag of asset returns, they should be padded with zeros, to avoid look-ahead bias. For the lag of prices, they should be padded with the first or last prices, not with zeros.

The function `lagxts()` is just a wrapper for function `lag.xts()` from package **xts**, but it pads with the first and last elements instead of NAs.

The function `lagit()` has incorporated the functionality of `lagxts()`, so that `lagxts()` will be retired in future package versions.

Value

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

Examples

```
# Lag by 10 periods
rutils::lagxts(rutils::etfenv$VTI, lag=10)
```

nalocf	<i>Replace NA values with the most recent non-NA values prior to them.</i>
--------	--

Description

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

Usage

```
nalocf(inputv, fromLast = FALSE, narm = FALSE, maxgap = NROW(inputv))
```

Arguments

inputv	A <i>numeric</i> or <i>Boolean</i> vector or matrix, or <i>xts</i> time series.
fromLast	A <i>Boolean</i> argument: should non-NA values be carried backward rather than forward? (the default is FALSE)
narm	A <i>Boolean</i> argument: should any remaining (leading or trailing) NA values be removed? (the default is FALSE)
maxgap	The maximum number of neighboring NA values that can be replaced (the default is NROW(inputv)).

Details

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

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

The function `nalocf()` performs the same operations as the function `zoo::na.locf()` from package `zoo`.

The function `nalocf()` calls the function `xts::na.locf.xts()` from package `xts`.

Value

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

Examples

```
# Create vector containing NA values
inputv <- sample(22)
inputv[sample(NROW(inputv), 4)] <- NA
# Replace NA values with the most recent non-NA values
rutils::nalocf(inputv)
# Create matrix containing NA values
inputv <- sample(44)
inputv[sample(NROW(inputv), 8)] <- NA
inputv <- matrix(inputv, nc=2)
# Replace NA values with the most recent non-NA values
rutils::nalocf(inputv)
# Create xts series containing NA values
inputv <- xts::xts(inputv, order.by=seq.Date(from=Sys.Date(),
by=1, length.out=NROW(inputv)))
```

```
# Replace NA values with the most recent non-NA values
rutils::na.locf(inputtv, fromLast=TRUE)
```

plot_acf	<i>Calculate the autocorrelation function (ACF) of a time series of returns, and plot it.</i>
----------	---

Description

Calculate the autocorrelation function (ACF) of a time series of returns, and plot it.

Usage

```
plot_acf(
  xtsv,
  lagg = 10,
  plotobj = TRUE,
  xlab = "Lag",
  ylab = "",
  main = "",
  ...
)
```

Arguments

xtsv	A vector, matrix, or time series of returns.
lagg	The maximum lag at which to calculate the ACF (the default is 10).
plotobj	A <i>Boolean</i> argument: should a plot be made? (the default is TRUE)
xlab	A string with the x-axis label.
ylab	A string with the y-axis label.
main	A string with the plot title.
...	Additional arguments to the function <code>stats::acf()</code> .

Details

The function `plot_acf()` calculates the autocorrelation function (ACF) of a time series of returns, and plots it. The function `plot_acf()` is just a wrapper for the function `stats::acf()`. The function `stats::acf()` calculates the autocorrelation function, including the lag zero autocorrelation, which is by definition equal to 1.

The function `plot_acf()` calls the function `stats::acf()`, removes the spurious lag zero autocorrelation, creates a plot, and returns the ACF data invisibly.

Value

Returns the ACF data invisibly and creates a plot.

Examples

```
# Plot the ACF of random returns
rutils::plot_acf(rnorm(1e4), lag=10, main="ACF of Random Returns")
# Plot the ACF of VTI returns
rutils::plot_acf(na.omit(rutils::etfenv$returns$VTI), lag=10, main="ACF of VTI Returns")
```

roll_max	<i>Calculate the rolling maximum of an xts time series over a sliding window (lookback period).</i>
----------	---

Description

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

Usage

```
roll_max(xtsv, lookb)
```

Arguments

xtsv	An <i>xts</i> time series containing one or more columns of data.
lookb	The size of the lookback window, equal to the number of data points for calculating the rolling sum.

Details

For example, if lookb=3, then the rolling sum at any point is equal to the sum of xtsv 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
xtsv <- xts(x=rnorm(1000), order.by=(Sys.time()-3600*(1:1000)))
rutils::roll_max(xtsv, lookb=3)
```

roll_sum	<i>Calculate the rolling sum of a numeric vector, matrix, or xts time series over a sliding window (lookback period).</i>
----------	---

Description

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

Usage

```
roll_sum(xtsv, lookb)
```

Arguments

xtsv	A vector, matrix, or <i>xts</i> time series containing one or more columns of data.
lookb	The size of the lookback window, equal to the number of data points for calculating the rolling sum.

Details

For example, if lookb=3, then the rolling sum at any point is equal to the sum of xtsv 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
vectorv <- rnorm(1000)
rutils::roll_sum(vectorv, lookb=3)
# Rolling sum of matrix
matrixv <- matrix(rnorm(1000), nc=5)
rutils::roll_sum(matrixv, lookb=3)
# Rolling sum of xts time series
xtsv <- xts(x=rnorm(1000), order.by=(Sys.time()-3600*(1:1000)))
rutils::roll_sum(xtsv, lookb=3)
```

sub_set	<i>Subset an xts time series (extract an xts sub-series corresponding to the input dates).</i>
---------	--

Description

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

Usage

```
sub_set(xtsv, startd, endd, get_rows = TRUE)
```

Arguments

xtsv	An <i>xts</i> time series.
startd	The start date of the extracted time series data.
endd	The end date of the extracted time series data, or the number of data rows to be extracted.
get_rows	A <i>Boolean</i> argument: if TRUE then extract the given number of rows of data, else extract the given number of calendar days (the default is TRUE).

Details

The function `sub_set()` extracts an *xts* sub-series corresponding to the input dates. If `endd` 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 `endd` 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 `endd` 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 `endd` is a number, and either `startd` or `endd` are outside the date range of *xtsv*, then `sub_set()` extracts the maximum available range of *xtsv*.

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::etfenv$VTI, startd="2015-01-01", endd="2015-01-10")
# Extract 6 consecutive rows of data from the past, using a date and a negative number
rutils::sub_set(rutils::etfenv$VTI, startd="2015-01-01", endd=-6)
# Extract 6 calendar days of data
rutils::sub_set(rutils::etfenv$VTI, startd="2015-01-01", endd=6, get_rows=FALSE)
# Extract up to 100 consecutive rows of data
rutils::sub_set(rutils::etfenv$VTI, startd="2016-08-01", endd=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 a lower periodicity (say minutes).

Usage

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

Arguments

<code>timeser</code>	An <i>OHLC</i> time series of prices in <i>xts</i> format.
<code>period</code>	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.)
endpoints	An integer vector of end points.

Details

The function `to_period()` performs a similar aggregation as function `xts::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 endpoints 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 endpoints.

Examples

```
## Not run:
# Define end points at 10-minute intervals (HighFreq::SPY are minutes bars)
endd <- rutils::calc_endpoints(HighFreq::SPY["2009"], interval=10)
# Aggregate over 10-minute endpoints:
ohlc <- rutils::to_period(timeser=HighFreq::SPY["2009"], endpoints=endd)
# Aggregate over days:
ohlc <- rutils::to_period(timeser=HighFreq::SPY["2009"], period="days")
# Equivalent to:
ohlc <- xts::to.period(x=HighFreq::SPY["2009"], period="days", name=rutils::get_name(colnames(HighFreq::SPY))

## End(Not run)
```