Package ‘PMwR’

July 21, 2024

Type Package

Title Portfolio Management with R

Version 0.19-5

Date 2024-07-21

Maintainer Enrico Schumann <es@enricoschumann.net>

Description Tools for the practical management of financial portfolios: backtesting investment and trading strategies, computing profit/loss and returns, analysing trades, handling lists of transactions, reporting, and more. The package provides a small set of reliable, efficient and convenient tools for processing and analysing trade/portfolio data. The Manual provides all the details; it is available from <https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html>. Examples and descriptions of new features are provided at <https://enricoschumann.net/notes/PMwR/>.

Imports NMOF, datetimeutils, fastmatch, orgutils, parallel, textutils, utils, zoo

Suggests crayon, rbenchmark, tinytest

Depends R (>= 3.5)

License GPL-3

LazyLoad yes

LazyData yes

ByteCompile yes

URL https://enricoschumann.net/PMwR/ ,
https://git.sr.ht/~enricoschumann/PMwR ,
https://gitlab.com/enricoschumann/PMwR ,
https://github.com/enricoschumann/PMwR

NeedsCompilation no

Author Enrico Schumann [aut, cre] (<https://orcid.org/0000-0001-7601-6576>)

Repository CRAN

Date/Publication 2024-07-21 13:40:02 UTC

Contents

`PMwR-package` *Tools for the Management of Financial Portfolios*

Description

Tools for the practical management of financial portfolios: backtesting investment and trading strategies, computing profit-and-loss and returns, analysing trades, reporting, and more.

Details

PMwR provides a small set of reliable, efficient and convenient tools for processing and analysing trade/portfolio data. The Manual provides all the details; it is available from `https://enricoschumann.net/PMwR/`. Examples and descriptions of new features are provided at `https://enricoschumann.net/notes/PMwR/`.

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

The **PMwR** Manual, which explains all functionality:
Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/PMwR/

The closely-related **NMOF** package is described in:
Gilli, M., Maringer, D. and Schumann, E. (2019) *Numerical Methods and Optimization in Finance*. 2nd edition. Elsevier. doi:10.1016/C2017001621X

Schumann, E. (2023) Financial Optimisation with R (**NMOF** Manual). https://enricoschumann.net/NMOF.htm#NMOFmanual

---

`Adjust-Series` *Adjust Time Series for Dividends and Splits*

Description

Adjust a time series for dividends and splits.

Usage

```
div_adjust(x, t, div, backward = TRUE, additive = FALSE)

split_adjust(x, t, ratio, backward = TRUE)
```

Arguments

`x`	a numeric vector: the series to be adjusted
`t`	An integer vector, specifying the positions in `x` at which dividends were paid (‘ex-days’) or at which a split occurred. Timestamps may be duplicated, e.g. several payments may occur on a single timestamp.
`div`	A numeric vector, specifying the dividends (or payments, cashflows). If necessary, recycled to the length of `t`.
`ratio`	a numeric vector, specifying the split ratios. The ratio must be ‘American Style’: a 2-for-1 stock split, for example, corresponds to a ratio of 2. (In other countries, for instance Germany, a 2-for-1 stock split would be called a 1-for-1 split: you keep your shares and receive one new share per share that you own.)
`backward`	logical; see Details
`additive`	logical; see Details

Details

The function transforms x into returns, and with those returns specified in t calculated as

$$\frac{x_t + D_t}{x_{t-1}} - 1\,,$$

in which x is the price, D are dividends and t is time. The adjusted x is then reconstructed from those returns.

When `additive` is `TRUE`, dividends are simply added back to the series; see Examples.

With `backward` set to `TRUE`, which is the default, the final prices in the unadjusted series matches the final prices in the adjusted series.

Value

a numeric vector of length equal to `length(x)`

Author(s)

Enrico Schumann

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/PMwR/

Using `div_adjust` for handling generic external cashflows: https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#returns-with-external-cashflows

Examples

```
x <- c(9.777, 10.04, 9.207, 9.406)
div <- 0.7
t <- 3

div_adjust(x, t, div)
div_adjust(x, t, div, FALSE)



## assume there were three splits: adjust shares outstanding
shares <- c(100, 100, 200, 200, 1000, 1500)
t <- c(3, 5, 6)
ratio <- c(2, 5, 1.5)
### => invert ratio
split_adjust(shares, t, 1/ratio)
## [1] 1500 1500 1500 1500 1500 1500

split_adjust(shares, t, 1/ratio, backward = FALSE)
## [1] 100 100 100 100 100 100
```

```
## 'additive' ** FALSE ** (default setting)
x <- c(100, 95, 100, 95, 100)
div <- 5
t <- c(2, 4)
div_adjust(x, t, div)
## 90.25  90.25  95.00  95.00 100.00
returns(div_adjust(x, t, div))
## 0.00000000 0.05263158 0.00000000 0.05263158
## ==> reflect _actual_ returns 100/95 - 1

## 'additive' ** TRUE **
div_adjust(x, t, div, additive = TRUE)
## 90  90  95  95 100
returns(div_adjust(x, t, div, additive = TRUE))
## 0.00000000 0.05555556 0.00000000 0.05263158
## ==> reflect return 95/90 - 1
```

---

btest *Backtesting Investment Strategies*

Description

Testing trading and investment strategies.

Usage

```
btest(prices, signal,
      do.signal = TRUE, do.rebalance = TRUE,
      print.info = NULL, b = 1, fraction = 1,
      initial.position = 0, initial.cash = 0,
      final.position = FALSE,
      cashflow = NULL, tc = 0, ...,
      add = FALSE, lag = 1, convert.weights = FALSE,
      trade.at.open = TRUE, tol = 1e-5, tol.p = NA,
      Globals = list(),
      prices0 = NULL,
      include.data = FALSE, include.timestamp = TRUE,
      timestamp, instrument,
      progressBar = FALSE,
      variations, variations.settings, replications)
```

Arguments

`prices` For a single asset, a matrix of prices with four columns: open, high, low and close. For `n` assets, a list of length four: `prices[[1]]` is then a matrix with `n` columns containing the open prices for the assets; `prices[[2]]` is a matrix with the high prices, and so on. If only close prices are used, then for a single asset either a matrix of one column or a numeric vector; for multiple assets a list of length one, containing the matrix of close prices. For example, with 100 close prices of 5 assets, the prices should be arranged in a matrix `p` of size 100 times 5; and `prices = list(p)`.

The series in `prices` are used both as transaction prices and for valuing open positions. If signals are to be based on other series, such other series should be passed via the ... argument.

Prices must be ordered by time (though the timestamps need not be provided).

`signal`
: A function that evaluates to the position in units of the instruments suggested by the trading rule. If `convert.weights` is `TRUE`, `signal` should return the suggested position as weights (which need not sum to 1). If `signal` returns `NULL`, the current position is kept. See Details.

`do.signal`
: Logical or numeric vector, a function that evaluates to `TRUE` or `FALSE`, or a string.

When a logical vector, its length must match the number of observations in prices: `do.signal` then corresponds to the rows in `prices` at which a signal is computed. Alternatively, these rows may also be specified as integers. If a length-one `TRUE` or `FALSE`, the value is recycled to match the number of observations in prices. Default is `TRUE`: a signal is then computed in every period.

`do.signal` may also be the string "firstofmonth", "lastofmonth", "firstofquarter" or "lastofquarter"; in these cases, `timestamp` needs to specified and must be coercable to `Date`.

If `timestamp` is specified, `do.signal` may also be a vector of the same class as `timestamp` (typically `Date` or `POSIXct`). If the timestamps specified in `do.signal` do not occur in `timestamp`, a signal is computed on the next possible time instance.

`do.rebalance`
: Same as `do.signal`, but it may return a logical vector of length equal to the number of assets, which indicates which assets to rebalance. Can also be the string `"do.signal"`, in which case the value of `do.signal` is copied. `do.rebalance` is called after signal computation, so it can access the suggested position of the current period (via `SuggestedPortfolio(0)`.

`print.info`
: A function, called at the very end of each period, i.e. after rebalancing. Can also be `NULL`, in which case nothing is printed.

`cashflow`
: A function or `NULL` (default).

`b`
: burn-in (an integer). Defaults to 1. This may also be a length-one timestamp of the same class as `timestamp`, in which case the data up to (and including) `b` are skipped.

`fraction`
: amount of rebalancing to be done: a scalar between 0 and 1

`initial.position`
: a numeric vector: initial portfolio in units of instruments. If supplied, this will also be the initial suggested position.

`initial.cash`
: a numeric vector of length 1. Defaults to 0.

`final.position`
: logical

`tc`
: transaction costs as a fraction of turnover (e.g., 0.001 means 0.1%). May also be a function that evaluates to such a fraction. More-complex computations may be specified with argument `cashflow`.

`...` other named arguments. All functions (signal, do.signal, do.rebalance, print.info, cashflow) will have access to these arguments. See Details for reserved argument names.

`add` Default is `FALSE`. `TRUE` is **not implemented** – but would mean that `signal` should evaluate to *changes* in position, i.e. orders.

`lag` default is 1

`convert.weights`
Default is `FALSE`. If `TRUE`, the value of signal will be considered a weight vector and automatically translated into (fractional) position sizes.

`trade.at.open` A logical vector of length one; default is `TRUE`.

`tol` A numeric vector of length one: only rebalance if the maximum absolute suggested change for at least one position is greater than `tol`. Default is 0.00001 (which practically means always rebalance).

`tol.p` A numeric vector of length one: only rebalance those positions for which the relative suggested change is greater than `tol.p`. Default is `NA`: always rebalance.

`Globals` A `list` of named elements. See Details.

`prices0` A numeric vector (default is `NULL`). Only used if `b` is 0 and an initial portfolio (`initial.position`) is specified.

`include.data` logical. If `TRUE`, all passed data are stored in final `btest` object. See Section Value. See also argument `include.timestamp`.

`include.timestamp`
logical. If `TRUE`, `timestamp` is stored in final `btest` object. If `timestamp` is missing, integers 1, 2, ... are used. See Section Value. See also argument `include.data`.

`timestamp` a vector of timestamps, along prices (optional; mainly used for print method and journal)

`instrument` character vector of instrument names (optional; mainly used for print method and journal)

`progressBar` logical: display `txtProgressBar`?

`variations` a list. See Details.

`variations.settings`
a list. See Details.

`replications` an integer. If set, the function returns a list of `btest` objects. Each `btest` has an attribute `replication`, which records the replication number.

Details

The function provides infrastructure for testing trading rules. Essentially, `btest` does accounting: keep track of transactions and positions, value open positions, etc. The ingredients are price timeseries (single series or OHLC bars), which need not be equally spaced; and several functions that map these series and other pieces of information into positions.

How `btest` works:

`btest` runs a loop from `b + 1` to `NROW(prices)`. In iteration `t`, a `signal` can be computed based on information from periods prior to `t`. Trading then takes place at the opening price of `t`.

```
t     time       open   high   low    close
1     HH:MM:SS                               <--\
2     HH:MM:SS                               <-- - use information
3     HH:MM:SS   _________________________   <--/
4     HH:MM:SS     X                         <- trade here
5     HH:MM:SS
```

For slow-to-compute signals this is reasonable if there is a time lag between close and open. For daily prices, for instance, signals could be computed overnight. For higher frequencies, such as every minute, the signal function should be fast to compute. Alternatively, it may be better to use a larger time offset (i.e. use a longer time lag) and to trade at the close of `t` by setting argument `trade.at.open` to `FALSE`.

```
t     time       open   high   low    close
1     HH:MM:SS                               <-- \
2     HH:MM:SS                               <-- - use information
3     HH:MM:SS   _________________________   <-- /
4     HH:MM:SS                         X     <-- trade here
5     HH:MM:SS
```

If no OHLC bars are available, a single series per asset (assumed to be close prices) can be used. `trade.at.open` will automaticall be set to `FALSE`.

The trade logic needs to be coded in the function `signal`. Arguments to that function must be named and need to be passed with `...`. Certain names are reserved and cannot be used as arguments: `Open`, `High`, `Low`, `Close`, `Wealth`, `Cash`, `Time`, `Timestamp`, `Portfolio`, `SuggestedPortfolio`, `Globals`. Further reserved names may be added in the future: **it is suggested to not start an argument name with a capital letter**.

The function `signal` must evaluate to the target position in units of the instruments. To work with weights, set `convert.weights` to `TRUE`, and `btest` will translate the weights into positions, based on the value of the portfolio at `t - 1`.

Accessing data:

Within `signal` (and also other function arguments, such as `do.signal`), you can access data via special functions such as `Close`. These are automatically added as arguments to `signal`. Currently, the following functions are available: `Open`, `High`, `Low`, `Close`, `Wealth`, `Cash`, `Time`, `Timestamp`, `Portfolio`, `SuggestedPortfolio`, `Globals`. `Globals` is special: it is an `environment`, which can be used to persistently store data during the run of `btest`. Use the argument `Globals` to add initial objects. See the Examples below and the manual.

Additional functions may be added to `btest` in the future. The names of those functions will always be in title case. Hence, it is recommended to not use argument names for `signal`, etc. that start with a capital letter.

Replications and variations:

`btest` allows to run backtests in parallel. See the examples at `https://enricoschumann.net/notes/parallel-backtests.html`.

The argument `variations.settings` is a list with the following defaults:

`method` character: supported are `"loop"`, `"parallel"` (or `"snow"`) and `"multicore"` `load.balancing` logical

`cores` numeric

Value

A list with class attribute `btest`. The list comprises:

`position`	actual portfolio holdings
`suggested.position`	suggested holdings (aka target position)
`cash`	cash
`wealth`	time-series of total portfolio value (aka equity curve)
`cum.tc`	transaction costs
`journal`	`journal` of trades. Only includes trades done during the backtest, not initial positions.
`initial.wealth`	initial wealth
`b`	burn-in
`final.position`	final position if `final.position` is `TRUE`; otherwise `NA`
`Globals`	environment `Globals`

When `include.timestamp` is `TRUE`, the timestamp is included. If no `timestamp` was specified, integers `1, 2, ...` are used instead.

When `include.data` is `TRUE`, essentially all information (prices, instrument, the actual `call` and functions `signal` etc.) are stored in the list as well.

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/PMwR/; in particular, see the chapter on backtesting:
https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#backtesting

Schumann, E. (2018) *Backtesting*.
https://papers.ssrn.com/sol3/papers.cfm?abstract_id=3374195

Examples

```
## For more examples, please see the Manual
## https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html

## 1 - a simple rule
timestamp <- structure(c(16679L, 16680L, 16681L, 16682L,
                         16685L, 16686L, 16687L, 16688L,
                         16689L, 16692L, 16693L),
                       class = "Date")
```

```
prices <- c(3182, 3205, 3272, 3185, 3201,
            3236, 3272, 3224, 3194, 3188, 3213)
data.frame(timestamp, prices)


signal <- function()     ## buy when last price is
    if (Close() < 3200)  ## below 3200, else sell
        1 else 0         ## (more precisely: build position of 1
                         ##  when price < 3200, else reduce
                         ##  position to 0)

solution <- btest(prices = prices, signal = signal)
journal(solution)


## with Date timestamps
solution <- btest(prices = prices, signal = signal,
                  timestamp = timestamp)
journal(solution)



## 2 - a simple MA model
## Not run:
library("PMwR")
library("NMOF")

dax <- DAX[[1]]

n <- 5
ma <- MA(dax, n, pad = NA)

ma_strat <-  function(ma) {
    if (Close() > ma[Time()])
        1
    else
        0
}


plot(as.Date(row.names(DAX)), dax, type = "l", xlab = "", ylab = "DAX")
lines(as.Date(row.names(DAX)), ma, type = "l")

res <- btest(prices = dax,
             signal = ma_strat,
             b = n, ma = ma)

par(mfrow = c(3, 1))
plot(as.Date(row.names(DAX)), dax, type = "l",
     xlab = "", ylab = "DAX")
plot(as.Date(row.names(DAX)), res$wealth, type = "l",
     xlab = "", ylab = "Equity")
plot(as.Date(row.names(DAX)), position(res), type = "s",
```

```
    xlab = "", ylab = "Position")

## End(Not run)
```

DAX *Deutscher Aktienindex (DAX)*

Description

Historical Prices of the DAX.

Usage

```
data("DAX")
```

Format

A data frame with 505 observations on the following variable:

`DAX` a numeric vector

Details

The DAX (*Deutscher Aktienindex*) is a stock-price index of the largest companies listed in Germany. The dataset comprises the close prices of the index for the years 2014 and 2015; dates are provided as rownames.

Examples

```
str(DAX)
summary(DAX)
```

---

drawdowns *Compute Drawdowns*

Description

Compute drawdown statistics.

Usage

```
drawdowns(x, ...)
## Default S3 method:
drawdowns(x, ...)
## S3 method for class 'zoo'
drawdowns(x, ...)
```

Arguments

`x`	a `numeric` vector of prices
`...`	additional arguments, to be passed to methods

Details

`drawdowns` is a generic function. It computes drawdown statistics: maximum; and time of peak, trough and recovery.

Value

a `data.frame`:

`peak`	peak before drawdown
`trough`	lowest point
`recover`	new high or `NA` if the drawdown has not been recovered yet
`max`	the max drawdown

Author(s)

Enrico Schumann

References

Gilli, M., Maringer, D. and Schumann, E. (2019) *Numerical Methods and Optimization in Finance*. 2nd edition. Elsevier. doi:10.1016/C2017001621X

Schumann, E. (2023) *Portfolio Management with R*. `https://enricoschumann.net/PMwR/`; in particular,
`https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#drawdowns-streaks`

See Also

The actual computation of the drawdowns is done by function `drawdown` in package **NMOF**.

Series of uninterrupted up and down movements can be computed with `streaks`.

Examples

```
x <- c(100, 98)
drawdowns(x)

x <- c(100, 98, 102, 99)
dd <- drawdowns(x)
dd[order(dd$max, decreasing = TRUE), ]
```

`instrument` *Retrieve or Change Instrument*

Description

Generic function for retrieving and changing instrument information.

Usage

```
instrument(x, ...)
instrument(x, ...) <- value
```

Arguments

`x`	an object
`...`	arguments passed to methods
`value`	an object

Details

Generic function: extract or, if meaningful, replace instrument information

Value

when used to extract instrument, a character vector

Author(s)

Enrico Schumann

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html

See Also

`position`

Examples

```
jnl <- journal(instrument = "A",
               amount = 100,
               price = 1)
instrument(jnl)
instrument(jnl) <- "B"
```

`is_valid_ISIN` *Validate Security Identification Numbers*

Description

Check whether a given ISIN or SEDOL is valid.

Usage

```
is_valid_ISIN(isin, NA.FALSE = FALSE)
is_valid_SEDOL(SEDOL, NA.FALSE = FALSE)
```

Arguments

`isin`	a character vector
`SEDOL`	a character vector
`NA.FALSE`	logical: if `TRUE`, `NA` values evaluate to `FALSE`.

Details

Checks a character vector of ISINs and SEDOLs. The function returns `TRUE` if the ISIN/SEDOL is valid, else `FALSE`. Handling of `NA` is defined by argument `NA.FALSE`.

International Securities Identification Numbers (ISINs): The test procedure in ISO 6166 does not differentiate between cases. Thus, ISINs are transformed to uppercase before being tested.

Value

A named logical vector. For `is_valid_SEDOL`, a character vector is attached as an attribute `note`.

Author(s)

Enrico Schumann

References

https://en.wikipedia.org/wiki/ISO_6166

https://en.wikipedia.org/wiki/SEDOL

https://anna-web.org/identifiers/

Examples

```
isin <- c("US0378331005", "AU0000XVGZA3",
          "DE000A0C3743", "not_an_isin")
is_valid_ISIN(isin)

is_valid_ISIN(c("US0378331005",
                "us0378331005")) ## case is ignored
```

```
SEDOL <- c("0263494", "B1F3M59", "0263491", "A", NA)
is_valid_SEDOL(SEDOL)
## 0263494 B1F3M59 0263491       A    <NA>
##    TRUE    TRUE   FALSE   FALSE      NA

is_valid_SEDOL(SEDOL, NA.FALSE = TRUE)
## 0263494 B1F3M59 0263491       A    <NA>
##    TRUE    TRUE   FALSE   FALSE   FALSE
```

---

journal *Journal*

Description

Create and manipulate a journal of financial transactions.

Usage

```
journal(amount, ...)

as.journal(x, ...)

is.journal(x)

## Default S3 method:
journal(amount, price, timestamp, instrument,
        id = NULL, account = NULL, ...)

## S3 method for class 'journal'
c(..., recursive = FALSE)

## S3 method for class 'journal'
length(x)

## S3 method for class 'journal'
aggregate(x, by, FUN, ...)

## S3 method for class 'journal'
print(x, ...,
      width = getOption("width"), max.print = getOption("max.print"),
      exclude = NULL, include.only = NULL)

## S3 method for class 'journal'
sort(x, decreasing = FALSE, by = "timestamp", ..., na.last = TRUE)

## S3 method for class 'journal'
```

```
summary(object, by = "instrument", drop.zero = TRUE,
      na.rm = FALSE, ...)

## S3 method for class 'journal'
subset(x, ...)

## S3 method for class 'journal'
x[i, match.against = NULL,
                    ignore.case = TRUE, perl = FALSE, fixed = FALSE,
                    useBytes = FALSE, ..., invert = FALSE]

## S3 replacement method for class 'journal'
x[i, match.against = NULL,
                    ignore.case = TRUE, ..., invert = FALSE] <- value

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

## S3 method for class 'journal'
head(x, n = 6L, ..., by = "instrument")

## S3 method for class 'journal'
tail(x, n = 6L, ..., by = "instrument")
```

Arguments

`timestamp`	An atomic vector of mode numeric or character. Timestamps should typically be sortable.
`amount`	numeric
`price`	numeric
`instrument`	character or numeric (though typically character)
`id`	An atomic vector. Default is `NULL`.
`account`	An atomic vector. Default is `NULL`.
`...`	For `journal`: further arguments, which must all be named. For `subset`: an expression that evaluates to a logical vector. The expression may use all fields of the passed journal; see Examples. For `` `[` ``: arguments other than `ignore.case` to be passed to `grep`. For `sort`: arguments passed to `sort`.
`x`	a `journal` or an object to be coerced to class `journal` (for `as.journal`) or to be checked if it inherits from journal (for `is.journal`)
`object`	a `journal`
`width`	integer. See `options`.
`decreasing`	passed to `sort`

`by`	`sort`: sort by field. `head/tail`: by field (default is instrument). `summary`: a vector of keywords (or `NULL`); supported are `"instrument"`, `"year"` and `"month"`.
`na.rm`	logical
`drop.zero`	logical
`na.last`	arguments passed to sort
`max.print`	maximum number of transactions to print
`exclude`	character: fields that should not be printed
`include.only`	character: print only those fields. (Not supported yet.)
`row.names`	see `as.data.frame`
`optional`	see `as.data.frame`
`recursive`	ignored (see `c`)
`i`	integer, logical or character. The latter is interpreted as a regexp (see `grep`)
`n`	integer
`match.against`	character vector of field names. Default is `NULL`, which means to match against all character fields.
`ignore.case`	logical: passed to `grepl`
`perl`	logical: passed to `grepl`
`fixed`	logical: passed to `grepl`
`useBytes`	logical: passed to `grepl`
`invert`	logical. If `TRUE`, invert selection (when `i` is of mode `character`, select journal entries that do not match regular expression)
`FUN`	either a function that takes as input a journal and evaluates to a journal, or a list of named functions
`value`	a replacement value

Details

The `journal` function creates a list of its arguments and attaches a class attribute (‘`journal`’). It is a generic function; the default method creates a journal from atomic vectors. The `btest` method extracts the journal from the results of a backtest; see `btest`.

`as.journal` coerces an object to a journal and is primarily used for creating a journal from a `data.frame`. Calling `as.journal` on an unnamed numeric vector interprets the vector as amounts. If the vector is named, these are interpreted as instruments; see Examples. Calling `as.journal` on a journal returns the journal itself.

`journal` methods are available for several generic functions, for instance:

`all.equal` compare contents of two journals

`aggregate` Splits a journal according to by, applies a function to every sub-journal and recombines the results into a journal.

`as.data.frame` Coerce journal to `data.frame`.

`c` Combine several journals into one. Note that the first argument to `c.journal` must inherit from `journal`, or else the method dispatch will fail. For empty journals, use `journal()` (not `NULL`).

`length` number of transactions in a journal; it uses the length of `amount`

`split` Splits a journal according to `f`, yielding a list of journals. Often used interactively to have information per sub-journal printed.

`subset` evaluates an expression in an environment that can access all fields of the journal. The function is meant for interactive analysis; care is needed when it is used within other functions: see Examples and the Manual.

`summary` provides summary statistics, such as number of trades and average buy/sell prices

`toOrg` converts a journal to an Org table; package **orgutils** must be available

For journals that have a length, missing arguments will be coded as `NA` except for `id` and `account`, which become `NULL`. In zero-length (i.e. 'empty') journals, all fields have length 0. A zero-length journal is created, for instance, by saying `journal()` or when an zero-row `data.frame` is passed to `as.journal`.

Value

An object of class `journal`, which is a list of atomic vectors.

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/R/packages/PMwR/; in particular, see
https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#journals

See Also

`position`, `pl`

Examples

```
j <- journal(timestamp = 1:3,
             amount = c(1,2,3),
             price = 101:103,
             instrument = c("Stock A", "Stock A", "Stock B"))

## *** subset *** in functions
##   this should work as expected ...
t0 <- 2.5
subset(j, timestamp > t0)

##   ... but here?!
tradesAfterT <- function(j, t0)
    subset(j, timestamp > t0)
tradesAfterT(j, 0)

##   if really required
```

```
tradesAfterT <- function(j, t0) {
    e <- substitute(timestamp > t0, list(t0 = t0))
    do.call(subset, list(j, e))
}
tradesAfterT(j, 0)

##   ... or much simpler
tradesAfterT <- function(j, t0)
    j[j$timestamp > t0]
tradesAfterT(j, 0)


## *** aggregate ***
##   several buys and sells on two days
##   aim: find average buy/sell price per day
j <- journal(timestamp = structure(c(15950, 15951, 15950, 15951, 15950,
                                     15950, 15951, 15951, 15951, 15951),
                                   class = "Date"),
             amount = c(-3, -4, -3, -1, 3, -2, 1, 3, 5, 3),
             price = c(104, 102, 102, 110, 106, 104, 104, 106, 108, 107),
             instrument = c("B", "B", "A", "A", "B", "B", "A", "B", "A", "A"))

by <- list(j$instrument, sign(j$amount), as.Date(j$timestamp))
fun <- function(x) {
    journal(timestamp = as.Date(x$timestamp[1]),
            amount = sum(x$amount),
            price = sum(x$amount*x$price)/sum(x$amount),
            instrument = x$instrument[1L])
}
aggregate(j, by = by, FUN = fun)


## *** iterate over transactions in (previously defined) journal ***
for (j in split(j, seq_along(j)))
    print(j)



## as.journal with numeric vector
as.journal(1:3)
##    amount
## 1       1
## 2       2
## 3       3
##
## 3 transactions

## as.journal with *named* numeric vector
x <- 1:3; names(x) <- LETTERS[1:3]
as.journal(x)
##    instrument  amount
## 1           A       1
## 2           B       2
```

```
## 3          C       3
##
## 3 transactions

x <- 1:3; names(x) <- c("A", "B", "A")
as.journal(x)
##    instrument  amount
## 1           A       1
## 2           B       2
## 3           A       3
##
## 3 transactions
```

NAVseries *Net-Asset-Value (NAV) Series*

Description

Create a net-asset-value (NAV) series.

Usage

```
NAVseries(NAV, timestamp,
          instrument = NULL, title = NULL, description = NULL,
          drop.NA = NULL)

as.NAVseries(x, ...)

## S3 method for class 'NAVseries'
print(x, ... )

## S3 method for class 'NAVseries'
summary(object, ..., monthly.vol = TRUE,
        bm = NULL, monthly.te = TRUE,
        na.rm = FALSE, assume.daily = FALSE)

## S3 method for class 'NAVseries'
plot(x, y, ..., xlab = "", ylab = "", type = "l")

## S3 method for class 'NAVseries'
window(x, start = NULL, end = NULL, ...)
```

Arguments

NAV	numeric
timestamp	time stamp (typically Date or POSIXct)
instrument	character

`title`	character
`description`	character
`x`	an `NAVseries` or an object to be coerced to NAVseries
`object`	an `NAVseries`
`...`	further arguments. For `summary`, these can be `NAVseries`.
`drop.NA`	logical
`bm`	an optional NAVseries. If `bm` does not inherit from `NAVseries`, `as.NAVseries` is tried.
`monthly.vol`	if `TRUE` (default), volatility computations are done on monthly returns
`monthly.te`	if `TRUE` (default), tracking error computations are done on monthly returns
`assume.daily`	logical
`na.rm`	logical
`y`	a second NAVseries to be plotted. Not supported yet.
`xlab`	character
`ylab`	character
`type`	character. See `plot`.
`start`	same class as timestamp; `NULL` means the first timestamp
`end`	same class as timestamp; `NULL` means the last timestamp

Details

NAV series:

An NAVseries is a numeric vector (the actual series) and additional information, attached as attributes: timestamp, instrument, title, description. Of these attributes, timestamp is the most useful, as it is used for several computations (e.g. when calling `summary`) or for plotting.

The 'instrument' is typically an internal label used to identify the series, such as a ticker; 'title' is a label, too, but is intended to be human-readable; 'description' finally should be human-readable as well but may be longer.

Summaries:

The `summary` method returns a list of the original NAVseries plus various statistics, such as return per year and volatility. The method may receive several NAV series as input

Value

an `NAVseries`: see Details.

an `NAVseries` summary: a list of lists. If a benchmark series is present, the summary has an attribute `bm`: an integer, specifying the position of the benchmark.

Note

The semantics of handling NAVseries are not stable yet. Currently, objects of class `NAVseries` are univariate: you create a single NAVseries, summarise it, plot it, and so one. In the future, at least some of the methods will support the multi-variate case, i.e. be able to handle several series at once.

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/PMwR/; in particular, see
https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#NAVseries

See Also

btest, journal

For handling external cashflows, see unit_prices, split_adjust and div_adjust.

Examples

```
summary(NAVseries(DAX[[1]], as.Date(row.names(DAX)), title = "DAX"))
```

---

pl   *Profit and Loss*

Description

Compute profit and (or) loss of financial transactions.

Usage

```
pl(amount, ... )

## Default S3 method:
pl(amount, price, timestamp = NULL,
   instrument = NULL, multiplier = 1,
   multiplier.regexp = FALSE,
   along.timestamp = FALSE, approx = FALSE,
   initial.position = NULL, initial.price = NULL,
   vprice = NULL, tol = 1e-10, do.warn = TRUE,
   do.sum = FALSE, pl.only = FALSE,
   footnotes = TRUE, ... )

## S3 method for class 'journal'
pl(amount, multiplier = 1,
   multiplier.regexp = FALSE,
   along.timestamp = FALSE, approx = FALSE,
   initial.position = NULL, initial.price = NULL,
   vprice = NULL, tol = 1e-10, do.warn = TRUE, ... )

## S3 method for class 'pl'
```

```
pl(amount, ... )

## S3 method for class 'pl'
print(x, ..., use.crayon = NULL, na.print = ".",
        footnotes = TRUE)

## S3 method for class 'pl'
as.data.frame(x, ... )

.pl(amount, price, tol = 1e-10, do.warn = TRUE)
.pl_stats(amount, price, tol = sqrt(.Machine$double.eps))
```

Arguments

`amount`	numeric or a `journal`
`price`	numeric
`instrument`	character or numeric (though typically character)
`timestamp`	An atomic vector of mode `numeric` or `character`. Timestamps should typically be sortable.
`along.timestamp`	logical; or a a vector of timestamps. If the latter, `vprice` must be specified as well. See the vignette "Profit/Loss for Open Positions" (`pl_open_positions`) for details. Timestamps must be in ascending order and will be sorted if they are not (and `vprice` will then be sorted as well).
`initial.position`	a `position`
`initial.price`	prices to evaluate initial position
`vprice`	valuation price; a numeric vector. With several instruments, the prices must be named, e.g. `c(stock1 = 100, stock2 = 101)`. See Details.
`multiplier`	numeric vector. When `instrument` is specified and the vector is named, the names will be matched against instruments.
`multiplier.regexp`	logical. If `TRUE`, the names of `multiplier` are interpreted as regular expressions. See Examples.
`approx`	logical
`tol`	numeric: threshold to consider a position zero.
`x`	a `pl` object to be printed or to be coerced to a data.frame
`...`	further argument
`use.crayon`	logical
`na.print`	character: how to print `NA` values
`do.warn`	logical: issue warnings?
`do.sum`	logical: sum profit/loss across instruments?
`pl.only`	logical: if `TRUE`, return only numeric vector of profit/loss
`footnotes`	logical, with default `TRUE`: collect and print notes?

Details

Computes profit and/or loss and returns a list with several statistics (see Section Value, below). To get only the profit/loss numbers as a numeric vector, set argument `pl.only` to `TRUE`.

`pl` is a generic function: The default input is vectors for amount, price, etc. Alternatively (and often more conveniently), the function may also be called with a `journal` or a `data.frame` as its input. For data frames, columns must be named `amount`, `price`, and so on, as in a `journal`.

`pl` may be called in two ways: either to compute *total profit/loss* from a list of trades, possibly broken down by `instrument` and `account`; or to compute *profit/loss over time*. The latter case typically requires setting arguments `along.timestamp` and/or `vprice` (see Examples). Profit/loss over time is always computed with time in ascending order: so if the timestamps in `along.timestamp` are not sorted, the function will sort them (and `vprice` as well).

Using `vprice`: when `along.timestamp` is logical (`FALSE` or `TRUE`), `vprice` can be used to value an open position. For a single asset, it should be a single number; for several assets, it should be named vector, with names indicating the `instrument`. When `along.timestamp` is used to pass a custom timestamp: for a single asset, `vprice` must be a vector with the same length as `along.timestamp`; for several assets, it must be a numeric matrix with dimension `length(along.timestamp)` times number of assets.

`.pl` and `.pl_stats` are helper functions that are called by `pl`. `.pl_stats` requires amount and price to be sorted in time, and to be of length > 0.

To use package **crayon** – which is only sensible in interactive use –, either explicitly set `use.crayon` to `TRUE` or set an option `PMwR.use.crayon` to `TRUE`.

Value

For `pl`, an object of class `pl`, which is a list of lists: one list for each instrument. Each such list contains numeric vectors: `pl`, `realised`, `unrealised`, `buy`, `sell`, `volume`. If `along.timestamp` is not `FALSE`, a vector `timestamp` is also present.

For `.pl`, a numeric vector with four elements: profit/loss in units of the instrument, sum of absolute amounts, average buy price, average sell price. For zero-length vector, the function evaluates to `c(0, 0, NaN, NaN)`.

For `.pl_stats`, a list of two elements: the average entry-price, and the realized profit/loss. profit/loss in units of the instrument, sum of absolute amounts, average buy price, average sell price. For zero-length vector, the function evaluates to `c(0, 0, NaN, NaN)`.

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/PMwR/; in particular https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#profit-and-loss

See Also

`btest`, `returns`

Examples

```
J <- journal(timestamp = c(  1,   2,   3),
             amount    = c(  1,   1,  -2),
             price     = c(100, 102, 101))
pl(J)

pl(amount = c(  1,   1,  -2),
   price  = c(100, 102, 101))  ## without a 'journal'


J <- journal(timestamp  = c(  1,   2,   3,   1,   2,   3),
             amount     = c(  1,   1,  -2,   1,   1,  -2),
             price      = c(100, 102, 101, 100, 102, 105),
             instrument = c(rep("Bond A", 3), rep("Bond B", 3)))

pl(J)
## Bond A
##   P/L total       0
##   average buy   101
##   average sell  101
##   cum. volume     4
##
## Bond B
##   P/L total       8
##   average buy   101
##   average sell  105
##   cum. volume     4
##
## 'P/L total' is in units of instrument;
## 'volume' is sum of /absolute/ amounts.

as.data.frame(pl(J))  ## a single data.frame
##        pl buy sell volume
## Bond A  0 101  101      4
## Bond B  8 101  105      4

lapply(pl(J), as.data.frame)  ## => a list of data.frames
## $`Bond A`
##   pl realised unrealised buy sell volume
## 1  0       NA         NA 101  101      4
##
## $`Bond B`
##   pl realised unrealised buy sell volume
## 1  8       NA         NA 101  105      4

pl(pl(J))  ## P/L as a numeric vector
## Bond A Bond B
##      0      8
```

```
## Example for 'vprice'
instrument  <- c(rep("Bond A", 2), rep("Bond B", 2))
amount <- c(1, -2, 2, -1)
price <- c(100, 101, 100, 105)

## ... no p/l because positions not closed:
pl(amount, price, instrument = instrument, do.warn = FALSE)

## ... but with vprice specified, p/l is computed:
pl(amount, price, instrument = instrument,
   vprice = c("Bond A" = 103, "Bond B" = 100))

### ... and is, except for volume, the same as here:
instrument  <- c(rep("Bond A", 3), rep("Bond B", 3))
amount <- c(1, -2, 1, 2, -1, -1)
price <- c(100, 101, 103, 100, 105, 100)
pl(amount, price, instrument = instrument)



## p/l over time: example for 'along.timestamp' and 'vprice'
j <- journal(amount = c(1, -1),
             price = c(100, 101),
             timestamp  = as.Date(c("2017-07-05", "2017-07-06")))
pl(j)

pl(j,
   along.timestamp = TRUE)

pl(j,
   along.timestamp = seq(from = as.Date("2017-07-04"),
                         to = as.Date("2017-07-07"),
                         by = "1 day"),
   vprice = 101:104)



## Example for 'multiplier'
jnl <- read.table(text =
"instrument, price, amount
 FGBL MAR 16, 165.20,  1
 FGBL MAR 16, 165.37, -1
 FGBL JUN 16, 164.12,  1
 FGBL JUN 16, 164.13, -1
 FESX JUN 16,   2910,  5
 FESX JUN 16,   2905, -5",
header = TRUE, stringsAsFactors = FALSE, sep = ",")


jnl <- as.journal(jnl)
pl(jnl,  multiplier.regexp = TRUE, ## regexp matching is case sensitive
   multiplier = c("FGBL" = 1000, "FESX" = 10))
```

```
## use package 'crayon'
## Not run:
## on Windows, you may also need 'options(crayon.enabled = TRUE)'
options(PMwR.use.crayon = FALSE)
pl(amount = c(1, -1), price = c(1, 2))
options(PMwR.use.crayon = TRUE)
pl(amount = c(1, -1), price = c(1, 2))

## End(Not run)
```

`plot_trading_hours` *Plot Time Series During Trading Hours*

Description

Plot a time series after removing weekends and specific times of the day.

Usage

```
plot_trading_hours(x, t = NULL, interval = "5 min",
                   labels = "hours", label.format = NULL,
                   exclude.weekends = TRUE, holidays = NULL,
                   fromHHMMSS = "000000", toHHMMSS = "240000",
                   do.plot.axis = TRUE,
                   ...,
                   from = NULL, to = NULL,
                   do.plot = TRUE,
                   axis1.par = list())
```

Arguments

`x`	A numeric vector. Can also be of class `zoo`.
`t`	A vector that inherits from class `POSIXt`. If `x` inherits from class `zoo`, then `index(x)` is used (and any supplied value for `t` is ignored).
`interval`	A character string like "num units", in which `num` is a number, and `units` is "sec", "min", "hour" or "day". The space between num and units is mandatory.
`labels`	A character vector of length one, determining the grid for `plot_trading_hours`: can be "hour", "day", "dayhour" or "month".
`label.format`	See `strftime`.
`exclude.weekends`	logical: default is `TRUE`
`holidays`	a vector of class `Date` or a character vector in a format that is understood by `as.Date`.

`fromHHMMSS`	a character vector of length one in format “HHMMSS”
`toHHMMSS`	a character vector of length one in format “HHMMSS”
`do.plot.axis`	logical. Should `axis(1)` be plotted? Default is `TRUE`.
`...`	parameters passed to `plot` (and typically `par`)
`from`	POSIXct: start plot at (if not specified, plot starts at first data point)
`to`	POSIXct: end plot at (if not specified, plot ends at last data point)
`do.plot`	logical. Should anything be plotted? Default is `TRUE`. If `FALSE`, the function returns a list of points.
`axis1.par`	a list of named elements

Details

Plot a timeseries during specific times of day.

Value

A list (invisibly if `do.plot` is `TRUE`):

`list(t, x, axis.pos = pos, axis.labels, timegrid)`

`t`	positions
`x`	values
`axis.pos`	positions of x-tickmarks
`axis.labels`	labels at x-ticks
`timegrid`	a POSIXct vector
`map`	a function. See the manual (a link is under References).

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

B.D. Ripley and K. Hornik. *Date-Time Classes*. R-News, **1**(2):8–12, 2001.

E. Schumann (2023) *Portfolio Management with R*. `https://enricoschumann.net/PMwR/`; in particular, see
`https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#plot-trading-hours`

See Also

`DateTimeClasses`

Examples

```
t <- as.POSIXct("2012-08-31 08:00:00") + 0:32400
x <- runif(length(t))

par(tck = 0.001, mgp = c(3,1,0.5), bty = "n")
p <- plot_trading_hours(x, t,
                        interval = "5 min", labels = "hours",
                        xlab = "time", ylab = "random points",
                        col = "blue")


## with ?lines
t <- as.POSIXct("2012-08-31 10:00:00") + 0:9000
x <- seq(0, 1, length.out = 9001)
lines(p$map(t)$t, x[p$map(t)$ix], pch = 19)
```

---

position *Aggregate Transactions to Positions*

Description

Use information on single trades to compute a position at a specific point in time.

Usage

```
position(amount, ...)

## Default S3 method:
position(amount, timestamp, instrument, when,
                 drop.zero = FALSE, account = NULL,
                 use.names = NULL, ...)

## S3 method for class 'journal'
position(amount, when, drop.zero = FALSE,
                          use.account = FALSE, ...)

## S3 method for class 'position'
print(x, ..., sep = ":")
```

Arguments

when	a timestamp or a vector of timestamps; alternatively, several keywords are supported. See Details.
amount	numeric or an object of class journal
timestamp	numeric or character: timestamps, must be sortable

instrument	character: symbols to identify different instruments
account	character: description of account. Ignored if NULL.
use.account	logical. If TRUE, positions are computed by account and instrument; otherwise by instrument only.
use.names	logical or NULL. The argument handles whether names of amount are used as instruments. If NULL: if amount is named and instrument is not specified, names of amount are interpreted as instruments. If use.names is FALSE, names of amount are ignored. (Ignoring names was the default behaviour prior to PMwR version 0.11.)
drop.zero	If logical, drop instruments that have a zero position; default is FALSE. If numeric, it is used as a tolerance; e.g., a value of 1-e12 will drop any position whose absolute amount is smaller than 1-e12.
x	An object of type position.
...	arguments passed to print
sep	A regular expression. Split instruments accordingly. **Not implemented yet.**

Details

position computes positions for lists of trades. position is a generic function; most useful is the method for journals.

The function checks if timestamp is sorted (see is.unsorted) and sorts the journal by timestamp, if required. If there are (some) NA values in timestamp, but timestamp is sorted otherwise, the function will proceed (with a warning, though).

The argument when can also be specified as one of several keywords: last (or newest or latest) provides the position at the latest timestamp; first (or oldest) provides the position at the earliest timestamp; all provides the positions at all timestamps in the journal. endofday, endofmonth and endofyear provide positions at the end of all calendar days, months and years within the timestamp range of the journal. The latter keywords can only work if timestamp can be coerced to Date.

Value

An object of class position, which is a numeric matrix with instrument and timestamp attributes. Note that position will never drop the result's dim attribute: it will always be a matrix of size length(when) times length(unique(instrument)), which may not be obvious from the printed output. The rows of the matrix correspond to timestamps; the columns correspond to instruments.

To extract the numeric position matrix, say as.matrix(p).

Author(s)

Enrico Schumann

References

Schumann, E. (2023) *Portfolio Management with* R. https://enricoschumann.net/R/packages/PMwR/; in particular, see
https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#computing-balances

See Also

journal; internal computations are handled by cumsum and findInterval

Examples

```
position(amount = c(1, 1, -1, 3, -4),
         timestamp = 1:5, when = 4.9)

## using a journal
J <- journal(timestamp = 1:5, amount = c(1, 1, -1, 3, -4))
position(J, when = 4.9)


## 'declaring' a position, using named amounts
amount <- c(1, 1, 1)
instrument <- c("A", "A", "B")
position(amount = amount, instrument = instrument)
## .... or equivalently
amount <- c(A = 2, B = 1)
position(amount)

## ignore names of amount
position(amount, use.names = FALSE)
```

---

pricetable *Price Table*

Description

Create price table

Usage

```
pricetable(price, ...)
```

Arguments

price	a matrix
...	further arguments, passed to methods

Details

`pricetable` is a helper function for extracting prices of particular instrument at specified dates. For this, it first creates a table that merges series passed via ... and appends a class attribute. A `[` method then allows to extract prices. Importantly, if you ask for a subset of m rows and n columns, the result will be a matrix of size m times n, even if times or instruments are missing.

`pricetable` is a generic function, currently with methods for numeric vectors (including vectors with a dim, aka matrices) and for zoo objects.

Value

a numeric matrix with class attribute `pricetable`

Author(s)

Enrico Schumann

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/PMwR/

See Also

`match`

Examples

```
## quickly creating a pricetable
pricetable(1:3)
pricetable(1:3, instrument = c("A", "B", "C"))
### ... and the same
pricetable(c(A = 1, B = 2, C = 3))


## subsetting examples
m <- 3
n <- 2
price <- array(c(1:m, 1:m + 100), dim = c(m,n))
colnames(price) <- LETTERS[1:n]
pt <- pricetable(price, timestamp = 1:m)
##   A   B
## 1 1 101
## 2 2 102
## 3 3 103

pt[ , "A"]
##   A
## 1 1
## 2 2
## 3 3

pt[ , c("X", "A", "X")]
##    X A  X
## 1 NA 1 NA
## 2 NA 2 NA
## 3 NA 3 NA

pt[ , c("X", "A", "X"), missing = 0]
##   X A X
## 1 0 1 0
## 2 0 2 0
## 3 0 3 0
```

```
pt[c(0, 1.5, 4), , missing = "locf"]
##      A   B
## 0   NA  NA
## 1.5  2 102
## 4    3 103
```

---

quote32 *Treasury Quotes with 1/32nds of Point*

Description

Print treasury quotes with 1/32nds of points.

Usage

```
quote32(price, sep = "(-|'|:)", warn = TRUE)
q32(price, sep = "(-|'|:)", warn = TRUE)
```

Arguments

`price`	numeric or character. See Details.
`sep`	character: a regular expression
`warn`	logical. Warn about rounding errors?

Details

The function is meant for pretty-printing of US treasury bond quotes; it provides no other functionality.

If `price` is `numeric`, it is interpreted as a quote in decimal notation and 'translated' into a price quoted in fractions of a point.

If `price` is `character`, it is interpreted as a quote in fractional notation.

`q32` is a short-hand for `quote32`.

Value

A numeric vector of class `quote32`.

Author(s)

Enrico Schumann

References

CME Group (2015). *Treasury Futures Price Rounding Conventions.* https://www.cmegroup.com/education/articles-and-reports/treasury-futures-price-rounding-conventions.html

Examples

```
quote32(100 + 17/32 + 0.75/32)
q32("100-172")

q32("100-272") - q32("100-270")
as.numeric(q32("100-272") - q32("100-270"))
```

`rc` *Return Contribution*

Description

Return contribution of portfolio segments.

Usage

```
rc(R, weights, timestamp, segments = NULL,
   R.bm = NULL, weights.bm = NULL,
   method = "contribution",
   linking.method = NULL,
   allocation.minus.bm = TRUE,
   tol = sqrt(.Machine$double.eps))
```

Arguments

`R`	returns: a numeric matrix
`weights`	the segment weights: a numeric matrix. `weights[i, j]` must correspond to `R[i, j]`
`timestamp`	character or numeric
`segments`	character. If missing, column names of `R` or of `weights` are used (if they are not `NULL`).
`method`	a string; default is `contribution`
`linking.method`	`NULL` or a string. Currently supported are `0-cumulative`, `1-cumulative`, `0.5-cumulative` (`geometric{0,1,0.5}`) and `logarithmic`. See Examples.
`allocation.minus.bm`	logical
`tol`	numeric: weights whose absolute value is below `tol` are considered zero and not used for computations. Ignored if `NA`.

If portfolio returns are to be compared against benchmark returns, benchmark returns and weights must be supplied:

`R.bm`	benchmark returns: a numeric matrix
`weights.bm`	the benchmark weights of segments: a numeric matrix. `weights.bm[i, j]` must correspond to `R.bm[i, j]`

Details

The function computes segment contribution, potentially over time. Returns and weights must be arranged in matrices, with rows corresponding to time periods and columns to portfolio segments. If `weights` and `R` are atomic vectors, then they are interpreted as cross-sectional weights/returns for a single period, i.e. they are handled like row vectors.

Weights can be missing, in which case `R` is assumed to already comprise segment returns.

Note that the segment contributions need not come from asset classes; the computation works for any additive single-period decomposition of portfolio returns.

Value

A list of two components:

`period_contributions`
: a data.frame of single-period contributions, sorted in time

`total_contributions`
: a numeric vector

Author(s)

Enrico Schumann

References

David R. Cariño (1999). Combining Attribution Effects Over Time. Journal of Performance Measurement. **3** (4), 5–14.

Jon A. Christopherson and David R. Cariño and Wayne E. Ferson (2009), *Portfolio Performance Measurement and Benchmarking*, McGraw-Hill.

Feibel, Bruce (2003), *Investment Performance Measurement*, Wiley.

Erik Valtonen (2002). Incremental Attribution with and without Notional Portfolios. Journal of Performance Measurement. **7** (1), 68–83.

https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#return-contribution

See Also

`returns`

Examples

```
weights <- rbind(c( 0.25, 0.75),
                 c( 0.40, 0.60),
                 c( 0.25, 0.75))

R <- rbind(c( 1  ,    0),
           c( 2.5, -1.0),
           c(-2  ,  0.5))/100

rc(R, weights, segment = c("equities", "bonds"))
```

```
## EXAMPLE of Christopherson et al., ch 19
weights <- cbind(stocks = c(0.5,  0.55),
                 bonds  = c(0.5, 0.45))
##      stocks bonds
## [1,]   0.50  0.50
## [2,]   0.55  0.45

R <- cbind(stocks = c(.4, 0.1),
           bonds  = c(.1, 0.2))
##      stocks bonds
## [1,]    0.4   0.1
## [2,]    0.1   0.2

## ==> contributions grow at portfolio rate-of-return
rc(R, weights, linking.method =  "geometric1")

## ==> contributions are made on top of current portfolio-value
rc(R, weights, linking.method =  "geometric0")

## ==> mixture
rc(R, weights, linking.method =  "geometric0.5")



## EXAMPLE from
## https://quant.stackexchange.com/questions/36520/
##         how-to-calculate-the-annual-contribution-of-a-fund-to-a-portfolio-of-funds/
##         36530#36530
## (unbreak the URL)

weights <- rbind(c( 0.5, 0.5),
                 c( 0.5, 0.5))

R <- rbind(c( 10,   0),
           c( 0 ,  10))/100


rc(R, weights, segment = c("F1", "F2"), timestamp = 1:2,
   linking.method = "geometric1")
## ==> F1 contributed first, and so gets a higher total
##     contribution

rc(R, weights, segment = c("F1", "F2"), timestamp = 1:2,
   linking.method = "geometric0")
## ==> F2 contributed later, and so gets a higher total
##     contribution because it started off a higher base
##     value
```

```
## contribution for btest:
##   run a portfolio 10% equities, 90% bonds
P <- as.matrix(merge(DAX, REXP, by = "row.names")[, -1])
(bt <- btest(prices = list(P),
             signal = function() c(0.1, 0.9),
             convert.weights = TRUE,
             initial.cash = 100))

W <- bt$position*P/bt$wealth
rc(returns(P)*W[-nrow(W), ])$total_contributions
```

`rebalance` *Rebalance Portfolio*

Description

Compute the differences between two portfolios.

Usage

```
rebalance(current, target, price,
          notional = NULL, multiplier = 1,
          truncate = TRUE, match.names = TRUE,
          fraction = 1, drop.zero = FALSE,
          current.weights = FALSE,
          target.weights = TRUE)

## S3 method for class 'rebalance'
print(x, ..., drop.zero = TRUE)

replace_weight(weights, ..., prefix = TRUE, sep = "::")
```

Arguments

`current`	the current holdings: a (typically named) vector of position sizes; can also be a `position`
`target`	the target holdings: a (typically named) vector of weights; can also be a `position`
`price`	a numeric vector: the current prices; may be named
`notional`	a single number: the value of the portfolio; if missing, replaced by `sum(current*prices)`
`multiplier`	numeric vector, possibly named
`truncate`	truncate computed positions? Default is `TRUE`.
`match.names`	logical
`fraction`	numeric
`x`	an object of class `rebalance`.

...	rebalance: arguments passed to print; replace_weight: numeric vectors
drop.zero	logical: should instruments with no difference between current and target be included? Note the different defaults for computing and printing.
current.weights	logical. If TRUE (the default), the values in current are interpreted as weights. If FALSE, current is interpreted as a position (i.e. notional/number of contracts).
target.weights	logical. If TRUE (the default), the values in target are interpreted as weights. If FALSE, target is interpreted as a position (i.e. notional/number of contracts).
weights	a numeric vector with named components
sep	character
prefix	logical

Details

The function computes the necessary trades to move from the current portfolio to a target portfolio.

replace_weight is a helper function to split baskets into their components. All arguments passed via ... should be named vectors. If names are not syntactically valid (see make.names), quote them. The passed vectors themselves should be passed as named arguments: see examples.

Value

An object of class rebalance, which is a data.frame:

instrument	character, or NA when match.names is FALSE
price	prices
current	current portfolio, in units of instrument
target	new portfolio, in units of instrument
difference	the difference between current and target portfolio

Attached to the data.frame are several attributes:

notional	a single number
match.names	logical
multiplier	a numeric vector with as many elements as the resulting data.frame has rows

Author(s)

Enrico Schumann

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/R/packages/PMwR/; in particular, see
https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#rebalance

See Also

journal

Examples

```
r <- rebalance(current = c(a = 100, b = 20),
               target  = c(a = 0.2, c = 0.3),
               price   = c(a = 1, b = 2, c = 3))
as.journal(r)

## replace_weight: the passed vectors must be named;
##                 'basket_3' is ignored because not
##                 component of weights is named
##                 'basket_3'

replace_weight(c(basket_1 = 0.3,
                 basket_2 = 0.7),
               basket_1 = c(a = 0.1, b = 0.4, c = .5),
               basket_2 = c(x = 0.1, y = 0.4, z = .5),
               basket_3 = c(X = 0.5, Z = 0.5),
               sep = "|")
```

---

returns *Compute Returns*

Description

Convert prices into returns.

Usage

```
returns(x, ...)

## Default S3 method:
returns(x, t = NULL, period = NULL, complete.first = TRUE,
        pad = NULL, position = NULL,
        weights = NULL, rebalance.when = NULL,
        lag = 1, na.rm = TRUE, ...)

## S3 method for class 'zoo'
returns(x, period = NULL, complete.first = TRUE,
        pad = NULL, position = NULL,
        weights = NULL, rebalance.when = NULL, lag = 1, na.rm = TRUE, ...)

## S3 method for class 'p_returns'
print(x, ..., year.rows = TRUE, month.names = NULL,
      zero.print = "0", plus = FALSE, digits = 1,
      na.print = NULL)
```

```
## S3 method for class 'p_returns'
toLatex(object, ...,
        year.rows = TRUE, ytd = "YTD", month.names = NULL,
        eol = "\\\\", stand.alone = FALSE)

## S3 method for class 'p_returns'
toHTML(x, ...,
       year.rows = TRUE, ytd = "YTD", month.names = NULL,
       stand.alone = TRUE, table.style = NULL, table.class = NULL,
       th.style = NULL, th.class = NULL,
       td.style = "text-align:right; padding:0.5em;",
       td.class = NULL, tr.style = NULL, tr.class = NULL,
       browse = FALSE)

.returns(x, pad = NULL, lag)
```

Arguments

x
: for the default method, a numeric vector (possibly with a dim attribute; i.e. a matrix) of prices. returns also supports x of other classes, such as zoo or NAVseries. For time-series classes, argument t should be NULL.

  For .returns, x must be numeric (for other classes, .returns may not work properly).

t
: timestamps. See arguments period and rebalance.when.

period
: Typically a string. Supported are "hour", "day", "month", "quarter", "year", "ann" (annualised), "ytd" (year-to-date), "mtd" (month-to-date), "itd" (inception-to-date) or a single year, such as "2012". Instead of "itd", "total" may also be used. The value of 'period' is used only when timestamp information is available: for instance, when t is not NULL or with zoo/xts objects. The exception is "itd", which can be computed without timestamp information. Holding period "ytd" produces a warning if the current year (as obtained from Sys.Date) differs from the latest timestamp of the series. Specifying period as "ytd!" suppresses the warning.

  All returns are computed as simple returns. They will only be annualised with option "ann"; they will not be annualised when the length of the time series is less than one year. To force annualising in such a case, use "ann!". Annualisation can only work when the timestamp t can be coerced to class Date. The result will have an attribute is.annualised, which is a logical vector of length one.

complete.first
: logical. For holding-period returns such an monthly or yearly, should the first period (if incomplete) be used.

pad
: either NULL (no padding of initial lost observation) or a value used for padding (reasonable values might be NA or 0)

na.rm
: logical; see Details

position
: either a numeric vector of the same length as the number of assets (i.e. ncol(x)), or a numeric matrix whose dimensions match those of prices (i.e. dim(x) must

	equal dim(weights)), or a matrix with as many rows as rebalance.when has elements
weights	either a numeric vector of the same length as the number of assets (i.e. ncol(x)), or a numeric matrix whose dimensions match those of prices (i.e. dim(x) must equal dim(weights)), or a matrix with as many rows as rebalance.when has elements
rebalance.when	logical or numeric. If x is a time-series class (such as zoo), it may also be of the same class as the time index of x.
...	further arguments to be passed to methods
year.rows	logical. If TRUE (the default), print monthly returns with one row per year.
zero.print	character. How to print zero values.
na.print	character. How to print NA values. (Not supported yet.)
plus	logical. Add a '+' before positive numbers? Default is FALSE.
lag	The lag for computing returns. A positive integer, defaults to one; ignored for time-weighted returns or if t is supplied.
object	an object of class p_returns ('period returns')
month.names	character: names of months. Default is an abbreviated month name as provided by the locale. That may cause trouble, notably with toLatex, if such names contain non-ASCII characters: a safe choice is either the numbers 1 to 12, or the character vector month.abb, which lives in the base package.
digits	number of digits in table
ytd	header for YTD
eol	character
stand.alone	logical or character
table.class	character
table.style	character
th.class	character
th.style	character
td.class	character
td.style	character
tr.class	character
tr.style	character
browse	logical: open table in browser?

Details

returns is a generic function. It computes simple returns: current values divided by prior values minus one. The default method works for numeric vectors/matrices. The function .returns does the actual computations and may be used when a 'raw' return computation is needed.

Holding-Period Returns:

When a timestamp is available, returns can compute returns for specific calendar periods. See argument period.

Portfolio Returns:

returns may compute returns for a portfolio specified in weights or position. The portfolio is rebalanced at rebalance.when; the default is every period. Weights need not sum to one. A zero-weight portfolio, or a portfolio that never rebalances (e.g. with rebalance.when set to FALSE), will result in a zero return.

rebalance.when may either be logical, integers or of the same class as a timestamp (e.g. Date).

Handling missing values:

Removing missing values (i.e. setting na.rm to TRUE) only has effects when period is specified.

Value

If called as returns(x): a numeric vector or matrix, possibly with a class attribute (e.g. for a zoo series).

If called with a period argument: an object of class "p_returns" (period returns), which is a numeric vector of returns with attributes t (timestamp) and period. Main use is to have methods that pretty-print such period returns; currently, there are methods for toLatex and toHTML.

In some cases, additional attributes may be attached: when portfolio returns were computed (i.e. argument weights was specified), there are attributes holdings and contributions. For holding-period returns, there may be a logical attribute is.annualised, and an attribute from.to, which tells the start and end date of the holding period.

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/R/packages/PMwR/; in particular, see
https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#computing-returns

See Also

btest, pl

Examples

```
x <- 101:105
returns(x)
returns(x, pad = NA)
returns(x, pad = NA, lag = 2)


## monthly returns
t <- seq(as.Date("2012-06-15"), as.Date("2012-12-31"), by = "1 day")
x <- seq_along(t) + 1000
returns(x, t = t, period = "month")
returns(x, t = t, period = "month", complete.first = FALSE)
```

```
### formatting
print(returns(x, t = t, period = "month"), plus = TRUE, digits = 0)

## returns per year (annualised returns)
returns(x, t = t, period = "ann")  ## less than one year, not annualised
returns(x, t = t, period = "ann!") ## less than one year, *but* annualised

is.ann <- function(x)
    attr(x, "is.annualised")

is.ann(returns(x, t = t, period = "ann"))   ## FALSE
is.ann(returns(x, t = t, period = "ann!"))  ## TRUE


## with weights and fixed rebalancing times
prices <- cbind(p1 = 101:105,
                p2 = rep(100, 5))
R <- returns(prices, weights = c(0.5, 0.5), rebalance.when = 1)
## ... => resulting weights
h <- attr(R, "holdings")
h*prices / rowSums(h*prices)
##             p1        p2
## [1,] 0.5000000 0.5000000  ## <== only initial weights are .5/.5
## [2,] 0.5024631 0.4975369
## [3,] 0.5049020 0.4950980
## [4,] 0.5073171 0.4926829
## [5,] 0.5097087 0.4902913
```

REXP *REXP*

Description

Historical Prices of the REXP.

Usage

```
data("REXP")
```

Format

A data frame with 502 observations on the following variable:

`REXP` a numeric vector

Details

Daily prices.

Examples

```
str(REXP)
```

---

scale1 *Scale Time Series*

Description

Scale time series so that they can be better compared.

Usage

```
scale1(x, ...)

## Default S3 method:
scale1(x, ..., when = "first.complete", level = 1,
       centre = FALSE, scale = FALSE, geometric = TRUE,
       total.g = NULL)

## S3 method for class 'zoo'
scale1(x, ..., when = "first.complete", level = 1,
       centre = FALSE, scale = FALSE, geometric = TRUE,
       inflate = NULL, total.g = NULL)
```

Arguments

x	a time series
when	origin: for the default method, either a string or numeric (integer). Allowed strings are "first.complete" (the default), "first", and "last". For the zoo method, a value that matches the class of the index of x; for instance, with an index of class Date, when should inherit from Date.
level	numeric
centre	logical
scale	logical or numeric
geometric	logical: if TRUE (the default), the geometric mean is deducted with centre is TRUE; if FALSE, the arithmetic mean is used
inflate	numeric: an annual rate at which the series is inflated (or deflated if negative)
total.g	numeric: to total growth rate (or total return) of a series
...	other arguments passed to methods

Details

This is a generic function, with methods for numeric vectors and matrices, and zoo objects.

Value

An object of the same type as x.

Author(s)

Enrico Schumann

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/PMwR/; in particular, see
https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#scaling-series

See Also

scale

Examples

```
scale1(cumprod(1 + c(0, rnorm(20, sd = 0.02))), level = 100)
```

---

streaks *Up and Down Streaks*

Description

Compute up and down streaks for time-series.

Usage

```
streaks(x, ...)

## Default S3 method:
streaks(x, up =  0.2, down = -up,
        initial.state = NA, y = NULL, relative = TRUE, ...)
## S3 method for class 'zoo'
streaks(x, up =  0.2, down = -up,
        initial.state = NA, y = NULL, relative = TRUE, ...)
## S3 method for class 'NAVseries'
streaks(x, up =  0.2, down = -up,
        initial.state = NA, bm = NULL, relative = TRUE, ...)
```

Arguments

`x`	a price series
`initial.state`	`NA`, `"up"` or `"down"`
`up`	a number, such as 0.1 (i.e. 10%)
`down`	a negative number, such as -0.1 (i.e. -10%)
`y`	another price series
`bm`	another price series. Mapped to ‘`y`’ in the default method.
`relative`	logical
`...`	other arguments passed to methods

Details

`streaks` is a generic function. It computes series of uninterrupted up and down movements (‘streaks’) in a price series. Uninterrupted is meant in the sense that no countermovement of `down` (`up`) percent or more occurs in up (down) movements.

There are methods for numeric vectors, and `NAVseries` and `zoo` objects.

The turning points (extreme points) are computed with the benefit of hindsight: the starting point (the low) of an up streak can only be determined once the streak is triggered, i.e. the up streak has already run its minimum amount. Vice versa for down streaks.

When ‘`up`’ and ‘`down`’ are not equal, results may be inconsistent: in the current implementation, `streaks` alternates between up and down streaks. Suppose `up` is large compared with `down`, i.e. it takes long to trigger up streaks, but they are easily broken. Down streaks, on the other hand, are quickly triggered but rarely broken. Now suppose that a down streak is broken by an up streak: it may then well be that the up streak would never have been counted as such, because it was actually broken itself by another down streak. The implementation for differing values of ‘`up`’ and ‘`down`’ may change in the future.

Value

A `data.frame`:

`start`	beginning of streak
`end`	end of streak
`state`	`up`, `down` or `NA`
`return`, `change`	the return over the streak. If `y` was specified, geometric excess return is computed (see Examples). If `relative` is `FALSE`, the column is named `change`.

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

Schumann, E. (2023) *Portfolio Management with R*. https://enricoschumann.net/PMwR/; in particular, see
https://enricoschumann.net/R/packages/PMwR/manual/PMwR.html#drawdowns-streaks

See Also

drawdowns

Examples

```
streaks(DAX[[1]], t = as.Date(row.names(DAX)))

## results <- streaks(x = <...>, y = <...>)
##
## ===> *arithmetic* excess returns
##       x[results$end]/x[results$start] -
##       y[results$end]/y[results$start]
## ===> *geometric* excess returns
##       x[results$end]/x[results$start] /
##      (y[results$end]/y[results$start]) - 1
```

---

toHTML *Import from package* **textutils**

Description

The `toHTML` function is imported from package **textutils**. Help is available at `textutils::toHTML`. Say `library("textutils")` in your code to use the function.

Trade-Analysis *Analysing Trades: Compute Profit/Loss, Resize and more*

Description

Functions to help analyse trades (as opposed to profit-and-loss series)

Usage

```
scale_trades(amount, price, timestamp, aggregate = FALSE,
             fun = NULL, ...)
split_trades(amount, price, timestamp, aggregate = FALSE,
              drop.zero = FALSE)

limit(amount, price, timestamp, lim, tol = 1e-8)
scale_to_unity(amount)
close_on_first(amount)

tw_exposure(amount, timestamp, start, end, abs.value = TRUE)
```

Arguments

amount	notionals
price	a vector of prices
timestamp	a vector.
aggregate	TRUE or FALSE
fun	a function
lim	a maximum absolute position size
start	optional time
end	optional time
drop.zero	logical. If TRUE, trades with zero amounts are removed. See Examples.
abs.value	logical. If TRUE, the absolute exposure is computed.
...	passed on to fun
tol	numeric

Details

scale_trades takes a vector of notionals, prices and scales all trades along the paths so that the maximum exposure is 1.

The default fun divides every element of a vector n by max(abs(cumsum(n))). If user-specified, the function fun needs to take a vector of notionals (changes in position.)

split_trades decomposes a trade list into single trades, where a single trade comprises those trades from a zero position to the next zero position. Note that the trades must be sorted chronologically.

Value

Either a list or a list-of-lists.

Author(s)

Enrico Schumann

See Also

btest

Examples

```
n <- c(1,1,-3,-1,2)
p <- 100 + 1:length(n)
timestamp <- 1:length(n)

scale_trades(n, p, timestamp)
scale_trades(n, p, timestamp, TRUE)  ## each _trade_ gets scaled

split_trades(n, p, timestamp)
```

```
split_trades(n, p, timestamp, TRUE)  ## almost like the original series



## effect of 'drop.zero'
P <- c(100, 99, 104, 103, 102, 105, 104)  ## price series
S <- c(  0,  1,   1,   0,   0,   1,   0)  ## position to be held
dS <- c(0, diff(S)) ## change in position ==> trades
t <- seq_along(P)

#### ==> 1) with all zero amounts
split_trades(amount = dS, price = P, timestamp = t)

#### ==> 2) without zero-amount trades
split_trades(amount = dS, price = P, timestamp = t, drop.zero = TRUE)

#### ==> 3) without all zero-amounts
zero <- dS == 0
split_trades(amount = dS[!zero], price = P[!zero], timestamp = t[!zero])
```

`unit_prices` *Compute Prices for Portfolio Based on Units*

Description

Compute prices for a portfolio based on outstanding shares.

Usage

```
unit_prices(NAV,
            cashflows,
            initial.price, initial.shares = 0,
            cf.included = TRUE)
```

Arguments

`NAV`	a dataframe of two columns: timestamp and net asset value
`cashflows`	a data.frame of two or three columns: timestamp, cashflow and (optionally) an id
`initial.price`	initial price
`initial.shares`	number of outstanding shares for first NAV
`cf.included`	logical

Details

This function is experimental, and its interface is not stable yet.

The function may be used to compute the returns for a portfolio with external cashflows, i.e. what is usually called time-weighted returns.

Valuation (i.e. the computation of the NAV) must take place before external cashflows. Fairness suggests that: what price would you give an external investor if you had not valued the positions? And even if fairness mattered not: suppose we traded on a specific day, had a positive PL, and ended the day in cash. We could then not differentiate any more between a cash increase because of an external inflow and a cash increase because of a profitable trade.

Value

A data.frame

`timestamp`	the timestamp
`NAV`	total NAV
`price`	NAV per share
`units`	outstanding units (i.e. shares) after cashflows

Attached as an attribute is a `data.frame` `transactions`.

Author(s)

Enrico Schumann

References

Schumann, E. (2023) *Portfolio Management with* R. https://enricoschumann.net/PMwR/

See Also

`returns`, `pl`

Examples

```
NAV <- data.frame(timestamp = seq(as.Date("2017-01-01"),
                                  as.Date("2017-01-10"),
                                  by = "1 day"),
                  NAV = c(100:104, 205:209))

cf <- data.frame(timestamp = c(as.Date("2017-01-01"),
                               as.Date("2017-01-06")),
                 cashflow = c(100, 100))

unit_prices(NAV, cf, cf.included = TRUE)
##     timestamp NAV    price    units
## 1  2017-01-01 100 100.0000 1.000000
## 2  2017-01-02 101 101.0000 1.000000
## 3  2017-01-03 102 102.0000 1.000000
```

```
## 4  2017-01-04 103 103.0000 1.000000
## 5  2017-01-05 104 104.0000 1.000000
## 6  2017-01-06 205 105.0000 1.952381
## 7  2017-01-07 206 105.5122 1.952381
## 8  2017-01-08 207 106.0244 1.952381
## 9  2017-01-09 208 106.5366 1.952381
## 10 2017-01-10 209 107.0488 1.952381
```

`valuation` *Valuation*

Description

Valuation of financial objects: map an object into a quantity that is measured in a concrete (typically currency) unit.

Usage

```
valuation(x, ...)

## S3 method for class 'journal'
valuation(x, multiplier = 1,
          cashflow = function(x, ...) x$amount * x$price,
          instrument = function(x, ...) "cash",
          flip.sign = TRUE, ...)

## S3 method for class 'position'
valuation(x, vprice, multiplier = 1,
          do.sum = FALSE,
          price.unit,
          use.names = FALSE,
          verbose = TRUE, do.warn = TRUE, ...)
```

Arguments

`x`	an object
`multiplier`	a numeric vector, typically with named elements
`cashflow`	either a numeric vector or a function that takes on argument (a journal) and transforms it into a numeric vector
`instrument`	either a character vector or a function that takes on argument (a journal) and transforms it into a character vector
`flip.sign`	logical. If `TRUE` (the default), a positive amount gets mapped into a negative cashflow.

vprice	numeric: a matrix whose elements correspond to those in x. If only a single timestamp is used and the position is named, this may also be a named numeric vector; see Examples. The argument behaves like vprice in pl; but for valuation those prices need not be sorted in time.
do.sum	logical: sum over positions
use.names	logical: use names of vprice?
price.unit	a named character vector. Not implemented.
verbose	logical
do.warn	logical
...	other arguments passed to methods

Details

This function is experimental, and the methods' interfaces are not stable yet.

valuation is a generic function. Its semantics suggest that an object (e.g. a financial instrument or a position) is mapped into a concrete quantity (such as an amount of some currency).

The journal method transforms the transactions in a journal into amounts of currency (e.g, a sale of 100 shares of a company is transformed into the value of these 100 shares).

The position method takes a position and returns the value (in currency units) of the position.

Value

depends on the object: for journals, a journal

Author(s)

Enrico Schumann <es@enricoschumann.net>

References

Schumann, E. (2020) *Portfolio Management with R*. https://enricoschumann.net/R/packages/PMwR/

See Also

journal

Examples

```
## valuing a JOURNAL

j <- journal(amount = 10, price = 2)
##    amount  price
## 1      10      2
##
## 1 transaction

valuation(j, instrument = NA)
```

```
##    amount  price
## 1     -20      1
##
## 1 transaction



## valuing a POSITION
pos <- position(c(AMZN = -10, MSFT = 200))

### contructing a price table:
### ==> P[i, j] must correspond to pos[i, j]
P <- array(c(2200, 170), dim = c(1, 2))
colnames(P) <- instrument(pos)

valuation(pos, vprice = P)
##        AMZN  MSFT
## [1,] -22000 34000

### contructing a price table, alternative:
### a named vector
### ==> only works when there is only a single timestamp
valuation(pos, vprice = c(MSFT = 170, AMZN = 2200))

all.equal(valuation(pos, vprice = P),
          valuation(pos, vprice = c(MSFT = 170, AMZN = 2200)))
```

Index