# Package: strand (via r-universe)

September 13, 2024

**Type** Package

**Title** A Framework for Investment Strategy Simulation

**Version** 0.2.0

**Date** 2020-11-18

**Description** Provides a framework for performing discrete (share-level) simulations of investment strategies. Simulated portfolios optimize exposure to an input signal subject to constraints such as position size and factor exposure. For background see L. Chincarini and D. Kim (2010, ISBN:978-0-07-145939-6) ``Quantitative Equity Portfolio Management".

**License** GPL-3

**URL** https://github.com/strand-tech/strand

**BugReports** https://github.com/strand-tech/strand/issues

**Depends** R (>= 3.5.0)

**Imports** R6, Matrix, Rglpk, dplyr, tidyr, arrow, lubridate, rlang, yaml, ggplot2, tibble, methods

**Suggests** testthat, knitr, rmarkdown, shiny, shinyFiles, shinyjs, DT, Rsymphony, officer, flextable, plotly

**Encoding** UTF-8

**LazyData** true

**VignetteBuilder** knitr

**RoxygenNote** 7.1.1

**Repository** https://strand-tech.r-universe.dev

**RemoteUrl** https://github.com/strand-tech/strand

**RemoteRef** HEAD

**RemoteSha** 9cdc27004620166b99a8a4c00f207425e45913b2

# Contents



---

`strand-package` *strand: a framework for investment strategy simulation*

---

## Description

The strand package provides a framework for performing discrete (share-level) simulations of investment strategies. Simulated portfolios optimize exposure to an input signal subject to constraints such as position size and factor exposure.

For an introduction to running simulations using the package, see `vignette("strand")`. For details on available methods see the documentation for the `Simulation` class.

## Author(s)

Jeff Enos <jeffrey.enos@gmail.com> and David Kane <dave.kane@gmail.com>

## Examples

```
# Load up sample data
data(sample_secref)
data(sample_pricing)
data(sample_inputs)

# Load sample configuration
config <- example_strategy_config()

# Override config file end date to run a one-week sim
config$to <- as.Date("2020-06-05")

# Create the Simulation object and run
sim <- Simulation$new(config,
```

```
                      raw_input_data = sample_inputs,
                      raw_pricing_data = sample_pricing,
                      security_reference_data = sample_secref)
sim$run()

# Print overall statistics
sim$overallStatsDf()

# Access tabular result data
head(sim$getSimSummary())
head(sim$getSimDetail())
head(sim$getPositionSummary())
head(sim$getInputStats())
head(sim$getOptimizationSummary())
head(sim$getExposures())

# Plot results
## Not run:
sim$plotPerformance()
sim$plotMarketValue()
sim$plotCategoryExposure("sector")
sim$plotFactorExposure(c("value", "size"))
sim$plotNumPositions()

## End(Not run)
```

---

`example_shiny_app` *Run an example shiny app*

---

## Description

Runs a shiny app that allows interactively configuring and running a simulation. Once the simulation is finished results, such as performance statistics and plots of exposures, are available in a results panel.

## Usage

```
example_shiny_app()
```

## Examples

```
if (interactive()) {
  example_shiny_app()
}
```

## example_strategy_config *Load example strategy configuration*

### Description

Loads an example strategy configuration file for use in examples.

### Usage

```
example_strategy_config()
```

### Value

An object of class `list` that contains the example configuration. The list object is the result of loading the package's example yaml configuration file `application/strategy_config.yaml`.

### Examples

```
config <- example_strategy_config()
names(config$strategies)
show(config$strategies$strategy_1)
```

## make_ft *Make Basic Flextable*

### Description

Make a flextable with preferred formatting

### Usage

```
make_ft(x, title = NULL, col_names = NULL, hlines = "all")
```

### Arguments

| | |
|---|---|
| `x` | The data.frame to use for flextable |
| `title` | The string to use as the table title |
| `col_names` | A character vector of preferred column names for flextable. Length of character vector must be equal to the number of columns. Defaults to NULL, in which case the column names of x are used in the flextable. |
| `hlines` | The row numbers to draw horizontal lines beneath. Defaults to "all", can be "all", "none", or a numeric vector. |

## Value

A flextable object with the argued formatting

---

`PortOpt` *Portfolio optimization class*

---

## Description

The `PortOpt` object is used to set up and solve a portfolio optimization problem.

## Details

A `PortOpt` object is configured in the same way as a `Simulation` object, by supplying configuration in a yaml file or list to the object constructor. Methods are available for adding constraints and retrieving information about the optimization setup and results. See the package vignette for information on configuration file setup.

## Methods

### Public methods:

- `PortOpt$new()`
- `PortOpt$setVerbose()`
- `PortOpt$addConstraints()`
- `PortOpt$getConstraintMatrix()`
- `PortOpt$getConstraintMeta()`
- `PortOpt$solve()`
- `PortOpt$getResultData()`
- `PortOpt$getLoosenedConstraints()`
- `PortOpt$getMaxPosition()`
- `PortOpt$getMaxOrder()`
- `PortOpt$summaryDf()`
- `PortOpt$print()`
- `PortOpt$clone()`

**Method** `new()`**:** Create a new `PortOpt` object.

*Usage:*

```
PortOpt$new(config, input_data)
```

*Arguments:*

`config` An object of class `list` or `character`. If the value passed is a character vector, it should be of length 1 and specify the path to a yaml configuration file that contains the object's configuration info. If the value passed is of class list(), the list should contain the object's configuration info in list form (e.g, the return value of calling `yaml.load_file` on the configuration file).

`input_data` A `data.frame` that contains all necessary input for the optimization.
If the top-level configuration item `price_var` is not set, prices will be expected in the `ref_price` column of `input_data`.

*Returns:* A new `PortOpt` object.

*Examples:*

```
library(dplyr)
data(sample_secref)
data(sample_inputs)
data(sample_pricing)

# Construct optimization input for one day from sample data. The columns
# of the input data must match the input configuration.
optim_input <-
  inner_join(sample_inputs, sample_pricing,
             by = c("id", "date")) %>%
  left_join(sample_secref, by = "id") %>%
  filter(date %in% as.Date("2020-06-01")) %>%
  mutate(ref_price = price_unadj,
                shares_strategy_1 = 0)

opt <-
  PortOpt$new(config = example_strategy_config(),
              input_data = optim_input)

# The problem is not solved until the \code{solve} method is called
# explicitly.
opt$solve()
```

**Method** `setVerbose()`**:** Set the verbose flag to control the amount of informational output.

*Usage:*

```
PortOpt$setVerbose(verbose)
```

*Arguments:*

`verbose` Logical flag indicating whether to be verbose or not.

*Returns:* No return value, called for side effects.

**Method** `addConstraints()`**:** Add optimization constraints.

*Usage:*

```
PortOpt$addConstraints(constraint_matrix, dir, rhs, name)
```

*Arguments:*

`constraint_matrix` Matrix with one row per constraint and $(S+1) \times N$ columns, where S is number of strategies and N is the number of stocks.
The variables in the optimization are

$$x_{1,1}, x_{2,1}, \ldots, x_{N,1},$$

$$x_{1,2}, x_{2,2}, \ldots, x_{N,2},$$

$$\vdots$$

$$x_{1,S}, x_{2,S}, \ldots, x_{N,S},$$

$$y_1, \ldots, y_N$$

The first $N \times S$ variables are the individual strategy trades. Variable $x_{i,s}$ represents the signed trade for stock i in strategy s. The following N auxillary variables $y_1, \ldots, y_N$ represent the absolute value of the net trade in each stock. So for a stock i, we have:

$$y_i = \sum_s |x_{i,s}|$$

`dir` Vector of class character of length `nrow(constraint_matrix)` that specifies the direction of the constraints. All elements must be one of ">=", "==", or "<=".

`rhs` Vector of class numeric of length `nrow(constraint_matrix)` that specifies the bounds of the constraints.

`name` Character vector of length 1 that specifies a name for the set of constraints that are being created.

*Returns:* No return value, called for side effects.

**Method** `getConstraintMatrix()`**:** Constraint matrix access.

*Usage:*

```
PortOpt$getConstraintMatrix()
```

*Returns:* The optimization's constraint matrix.

**Method** `getConstraintMeta()`**:** Provide high-level constraint information.

*Usage:*

```
PortOpt$getConstraintMeta()
```

*Returns:* A data frame that contains constraint metadata, such as current constraint value and whether a constraint is currently within bounds, for all single-row constraints. Explicitly exclude net trade constraints and constraints that involve net trade variables.

**Method** `solve()`**:** Solve the optimization. After running `solve()`, results can be retrieved using `getResultData()`.

*Usage:*

```
PortOpt$solve()
```

*Returns:* No return value, called for side effects.

**Method** `getResultData()`**:** Get optimization result.

*Usage:*

```
PortOpt$getResultData()
```

*Returns:* A data frame that contains the number of shares and the net market value of the trades at the strategy and joint (net) level for each stock in the optimization's input.

**Method** `getLoosenedConstraints()`**:** Provide information about any constraints that were loosened in order to solve the optimization.

*Usage:*

```
PortOpt$getLoosenedConstraints()
```

*Returns:* Object of class `list` where keys are the names of the loosened constraints and values are how much they were loosened toward current values. Values are expressed as (current constraint value - loosened constraint value) / (current constraint value - violated constraint value). A value of 0 means a constraint was loosened 100% and is not binding.

**Method** `getMaxPosition()`**:** Provide information about the maximum position size allowed for long and short positions.

*Usage:*

```
PortOpt$getMaxPosition()
```

*Returns:* An object of class `data.frame` that contains the limits on size for long and short positions for each strategy and security. The columns in the data frame are:

**id** Security identifier.

**strategy** Strategy name.

**max_pos_lmv** Maximum net market value for a long position.

**max_pos_smv** Maximum net market value for a short position.

**Method** `getMaxOrder()`**:** Provide information about the maximum order size allowed for each security and strategy.

*Usage:*

```
PortOpt$getMaxOrder()
```

*Returns:* An object of class `data.frame` that contains the limit on order size for each strategy and security. The columns in the data frame are:

**id** Security identifier.

**strategy** Strategy name.

**max_order_gmv** Maximum gross market value allowed for an order.

**Method** `summaryDf()`**:** Provide aggregate level optimization information if the problem has been solved.

*Usage:*

```
PortOpt$summaryDf()
```

*Returns:* A data frame with one row per strategy, including the joint (net) level, and columns for starting and ending market values and factor expoure values.

**Method** `print()`**:** Print summary information.

*Usage:*

```
PortOpt$print()
```

*Returns:* No return value, called for side effects.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
PortOpt$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

## Examples

```
## ------------------------------------------------
## Method `PortOpt$new`
## ------------------------------------------------

library(dplyr)
data(sample_secref)
data(sample_inputs)
data(sample_pricing)

# Construct optimization input for one day from sample data. The columns
# of the input data must match the input configuration.
optim_input <-
  inner_join(sample_inputs, sample_pricing,
             by = c("id", "date")) %>%
  left_join(sample_secref, by = "id") %>%
  filter(date %in% as.Date("2020-06-01")) %>%
  mutate(ref_price = price_unadj,
                shares_strategy_1 = 0)

opt <-
  PortOpt$new(config = example_strategy_config(),
              input_data = optim_input)

# The problem is not solved until the \code{solve} method is called
# explicitly.
opt$solve()
```

---

`sample_inputs` *Sample security inputs for examples and testing*

---

## Description

A dataset containing sample security input data for 492 securities and 65 weekdays, from 2020-06-01 to 2020-08-31. Data items include average trading dollar volume, market cap, and normalized size and value factors. The pricing data used to construct the dataset was downloaded using the Tiingo Stock API and is used with permission. Fundamental data items were downloaded from EDGAR.

## Usage

```
data(sample_inputs)
```

## Format

A data frame with 31980 rows and 7 variables:

**date** Input date. It is assumed that the input data for day X is known at the beginning of day X (e.g., the data is as-of the previous day's close).

**id** Security identifier.

**rc_vol** Average dollar trading volume for the security over the past 20 trading days.

**market_cap** Market capitalization, in dollars. The shares outstanding value used to calculate market cap is the latest value available at the beginning of the month.

**book_to_price** Ratio of total equity to market cap. The stockholders' equity value used to calculate book to price is the latest value available at the beginning of the month.

**size** Market cap factor normalized to be N(0,1) for each day.

**value** Book to price factor normalized to be N(0,1) for each day.

## Details

Data for most members of the S&P 500 are present. Some securities have been omitted due to data processing complexities. For example, securities for companies with multiple share classes have been omitted in the current version.

Values for shares outstanding and stockholders' equity downloaded from EDGAR may be inaccurate due to XBRL parsing issues.

Full code for reconstructing the dataset can be found in the pystrand repository.

---

`sample_pricing` *Sample security pricing data for examples and testing*

---

## Description

A dataset containing sample security pricing data for 492 securities and 65 weekdays, from 2020-06-01 to 2020-08-31. This data was downloaded using the Tiingo Stock API and is redistributed with permission.

## Usage

```
data(sample_pricing)
```

## Format

A data frame with 31980 rows and 8 variables:

**date** Pricing date.

**id** Security identifier.

**price_unadj** The unadjusted price of the security.

**prior_close_unadj** The unadjusted prior closing price of the security.

**dividend_unadj** The dividend for the security on an unadjusted basis, if any.

**distribution_unadj** The distribution (e.g., spin-off) for the security on an unadjusted basis (note that there is no spin-off information in this dataset, so all values are zero).

**volume** Trading volume for the security, in shares.

**adjustment_ratio** The adjustment ratio for the security. For example, AAPL has an adjustment ratio of 0.25 to account for its 4:1 split on 2020-08-31.

### Details

Full code for reconstructing the dataset can be found in the pystrand repository.

---

## sample_secref *Sample security reference data for examples and testing*

---

### Description

A dataset containing sample reference data for the securities of 492 large companies. All securities in the dataset were in the S&P 500 for most or all of the period June-August 2020.

### Usage

```
data(sample_secref)
```

### Format

A data frame with 492 rows and 4 variables:

**id** Unique security identifier (the security's ticker).

**name** Company name.

**symbol** Human-readable symbol for display and reporting purposes. In the case of this dataset it is the same as the `id` variable.

**sector** GICS sector for the company according to the Wikipedia page List of S&P 500 companies.

---

## show_best_worst *Show Best/Worst Performers*

---

### Description

Build a flextable object showing a Simulation's best and worst performers

### Usage

```
show_best_worst(sim)
```

### Arguments

| | |
|---|---|
| `sim` | A Simulation object to show the best and worst performers for |

## show_config *Show Strategy Configuration*

### Description

Build a flextable object showing a Simulation's configuration

### Usage

```
show_config(sim)
```

### Arguments

| | |
|---|---|
| sim | A Simulation object to show the configuration for |

## show_constraints *Show Strategy Constraints*

### Description

Build a flextable object showing a Simulation's risk constraints

### Usage

```
show_constraints(sim)
```

### Arguments

| | |
|---|---|
| sim | A Simulation object to show the configuration for |

## show_monthly_returns *Show monthly returns*

### Description

Build a flextable object that shows a simulation's return by month by formatting the output of 'Simulation$overallReturnsByMonthDf'.

### Usage

```
show_monthly_returns(sim)
```

### Arguments

| | |
|---|---|
| sim | A Simulation object with results to display |

## show_stats *Show Overall Stats Table*

### Description

Build a flextable object showing a Simulation's overall statistics

### Usage

```
show_stats(sim)
```

### Arguments

| | |
|---|---|
| sim | A Simulation object to show the statistics for |

## Simulation *Simulation class*

### Description

Class for running a simulation and getting results.

### Details

The `Simulation` class is used to set up and run a daily simulation over a particular period. Portfolio construction parameters and other simulator settings can be configured in a yaml file that is passed to the object's constructor. See `vignette("strand")` for information on configuration file setup.

### Methods

#### Public methods:

- `Simulation$new()`
- `Simulation$setVerbose()`
- `Simulation$setShinyCallback()`
- `Simulation$getSecurityReference()`
- `Simulation$run()`
- `Simulation$getSimDates()`
- `Simulation$getSimSummary()`
- `Simulation$getSimDetail()`
- `Simulation$getPositionSummary()`
- `Simulation$getInputStats()`
- `Simulation$getLooseningInfo()`
- `Simulation$getOptimizationSummary()`
- `Simulation$getExposures()`

- Simulation$getDelistings()
- Simulation$getSingleStrategySummaryDf()
- Simulation$plotPerformance()
- Simulation$plotContribution()
- Simulation$plotMarketValue()
- Simulation$plotCategoryExposure()
- Simulation$plotFactorExposure()
- Simulation$plotNumPositions()
- Simulation$plotTurnover()
- Simulation$plotUniverseSize()
- Simulation$plotNonInvestablePct()
- Simulation$overallStatsDf()
- Simulation$overallReturnsByMonthDf()
- Simulation$print()
- Simulation$writeFeather()
- Simulation$readFeather()
- Simulation$getConfig()
- Simulation$writeReport()
- Simulation$clone()

**Method** `new()`**:** Create a new `Simulation` object.

*Usage:*

```
Simulation$new(
  config = NULL,
  raw_input_data = NULL,
  input_dates = NULL,
  raw_pricing_data = NULL,
  security_reference_data = NULL,
  delisting_data = NULL
)
```

*Arguments:*

`config` An object of class `list` or `character`, or `NULL`. If the value passed is a character vector, it should be of length 1 and specify the path to a yaml configuration file that contains the object's configuration info. If the value passed is of class list(), the list should contain the object's configuration info in list form (e.g, the return value of calling `yaml.load_file` on the configuration file). If the value passed is `NULL`, then there will be no configuration information associated with the simulation and it will not possible to call the `run` method. Setting `config = NULL` is useful when creating simulation objects into which results will be loaded with `readFeather`.

`raw_input_data` A data frame that contains all of the input data (for all periods) for the simulation. The data frame must have a `date` column. Data supplied using this parameter will be used if the configuration option `simulator/input_data/type` is set to `object`. Defaults to `NULL`.

`input_dates` Vector of class `Date` that specifies when input data should be updated. If data is being supplied using the `raw_input_data` parameter, then `input_dates` defaults to set of dates present in this data.

`raw_pricing_data` A data frame that contains all of the input data (for all periods) for the simulation. The data frame must have a `date` column. Data supplied using this parameter will only be used if the configuration option `simulator/pricing_data/type` is set to `object`. Defaults to `NULL`.

`security_reference_data` A data frame that contains reference data on the securities in the simulation, including any categories that are used in portfolio construction constraints. Note that the simulator will throw an error if there are input data records for which there is no entry in the security reference. Data supplied using this parameter will only be used if the configuration option `simulator/secref_data/type` is set to `object`. Defaults to `NULL`.

`delisting_data` A data frame that contains delisting dates and associated returns. It must contain three columns: id (character), delisting_date (Date), and delisting_return (numeric). The date in the delisting_date column means the day on which a stock will be removed from the simulation portfolio. It is typically the day after the last day of trading. The delisting_return column reflects what, if any, P&L should be recorded on the delisting date. A delisting_return of -1 means that the shares were deemed worthless. The delisting return is multiplied by the starting net market value of the position to determine P&L for the delisted position on the delisting date. Note that the portfolio optimization does not include stocks that are being removed due to delisting. Data supplied using this parameter will only be used if the configuration option `simulator/delisting_data/type` is set to `object`. Defaults to `NULL`.

*Returns:* A new `Simulation` object.

**Method** `setVerbose()`**:** Set the verbose flag to control info output.

*Usage:*

```
Simulation$setVerbose(verbose)
```

*Arguments:*

`verbose` Logical flag indicating whether to be verbose or not.

*Returns:* No return value, called for side effects.

**Method** `setShinyCallback()`**:** Set the callback function for updating progress when running a simulation in shiny.

*Usage:*

```
Simulation$setShinyCallback(callback)
```

*Arguments:*

`callback` A function suitable for updating a shiny Progress object. It must have two parameters: `value`, indicating the progress amount, and detail, and `detail`, a text string for display on the progress bar.

*Returns:* No return value, called for side effects.

**Method** `getSecurityReference()`**:** Get security reference information.

*Usage:*

```
Simulation$getSecurityReference()
```

*Returns:* An object of class `data.frame` that contains the security reference data for the simulation.

**Method** `run()`**:** Run the simulation.

*Usage:*

```
Simulation$run()
```

*Returns:* No return value, called for side effects.

**Method** `getSimDates()`**:** Get a list of all date for the simulation.

*Usage:*

```
Simulation$getSimDates()
```

*Returns:* A vector of class `Date` over which the simulation currently iterates: all weekdays between the 'from' and 'to' dates in the simulation's config.

**Method** `getSimSummary()`**:** Get summary information.

*Usage:*

```
Simulation$getSimSummary(strategy_name = NULL)
```

*Arguments:*

`strategy_name` Character vector of length 1 that specifies the strategy for which to get detail data. If `NULL` data for all strategies is returned. Defaults to `NULL`.

*Returns:* An object of class `data.frame` that contains summary data for the simulation, by period, at the joint and strategy level. The data frame contains the following columns:

**strategy** Strategy name, or 'joint' for the aggregate strategy.

**sim_date** Date of the summary data.

**market_fill_nmv** Total net market value of fills that do not net down across strategies.

**transfer_fill_nmv** Total net market value of fills that represent "internal transfers", i.e., fills in one strategy that net down with fills in another. Note that at the joint level this column by definition is 0.

**market_order_gmv** Total gross market value of orders that do not net down across strategies.

**market_fill_gmv** Total gross market value of fills that do not net down across strategies.

**transfer_fill_gmv** Total gross market value of fills that represent "internal transfers", i.e., fills in one strategy that net down with fills in another.

**start_nmv** Total net market value of all positions at the start of the period.

**start_lmv** Total net market value of all long positions at the start of the period.

**start_smv** Total net market value of all short positions at the start of the period.

**end_nmv** Total net market value of all positions at the end of the period.

**end_gmv** Total gross market value of all positions at the end of the period.

**end_lmv** Total net market value of all long positions at the end of the period.

**end_smv** Total net market value of all short positions at the end of the period.

**end_num** Total number of positions at the end of the period.

**end_num_long** Total number of long positions at the end of the period.

**end_num_short** Total number of short positions at the end of the period.

**position_pnl** The total difference between the end and start market value of positions.

**trading_pnl** The total difference between the market value of trades at the benchmark price and at the end price. Note: currently assuming benchmark price is the closing price, so trading P&L is zero.

**gross_pnl** Total P&L gross of costs, calculated as position_pnl + trading_pnl.

**trade_costs** Total trade costs (slippage).

**financing_costs** Total financing/borrow costs.

**net_pnl** Total P&L net of costs, calculated as gross_pnl - trade_costs - financing_costs.

**fill_rate_pct** Total fill rate across all market orders, calculated as 100 * market_fill_gmv / market_order_gmv.

**num_investable** Number of investable securities (size of universe).

**Method** `getSimDetail()`**:** Get detail information.

*Usage:*

```
Simulation$getSimDetail(
  sim_date = NULL,
  strategy_name = NULL,
  security_id = NULL,
  columns = NULL
)
```

*Arguments:*

`sim_date` Vector of length 1 of class Date or character that specifies the period for which to get detail information. If `NULL` then data from all periods is returned. Defaults to `NULL`.

`strategy_name` Character vector of length 1 that specifies the strategy for which to get detail data. If `NULL` data for all strategies is returned. Defaults to `NULL`.

`security_id` Character vector of length 1 that specifies the security for which to get detail data. If `NULL` data for all securities is returned. Defaults to `NULL`.

`columns` Vector of class character specifying the columns to return. This parameter can be useful when dealing with very large detail datasets.

*Returns:* An object of class `data.frame` that contains security-level detail data for the simulation for the desired strategies, securities, dates, and columns. Available columns include:

**id** Security identifier.

**strategy** Strategy name, or 'joint' for the aggregate strategy.

**sim_date** Date to which the data pertains.

**shares** Shares at the start of the period.

**int_shares** Shares at the start of the period that net down with positions in other strategies.

**ext_shares** Shares at the start of the period that do not net down with positions in other strategies.

**order_shares** Order, in shares.

**market_order_shares** Order that does not net down with orders in other strategies, in shares.

**transfer_order_shares** Order that nets down with orders in other strategies, in shares.

**fill_shares** Fill, in shares.

**market_fill_shares** Fill that does not net down with fills in other strategies, in shares.

**transfer_fill_shares** Fill that nets down with fills in other strategies, in shares.

**end_shares** Shares at the end of the period.

**end_int_shares** Shares at the end of the period that net down with positions in other strategies.

**end_ext_shares** Shares at the end of the period that do not net down with positions in other strategies.

**start_price** Price for the security at the beginning of the period.

**end_price** Price for the security at the end of the period.

**dividend** Dividend for the security, if any, for the period.

**distribution** Distribution (e.g., spin-off) for the security, if any, for the period.

**investable** Logical indicating whether the security is part of the investable universe. The value of the flag is set to TRUE if the security has not been delisted and satisfies the universe criterion provided (if any) in the `simulator/universe` configuration option.

**delisting** Logical indicating whether a position in the security was removed due to delisting. If delisting is set to TRUE, the gross_pnl and net_pnl columns will contain the P&L due to delisting, if any. P&L due to delisting is calculated as the delisting return times the `start_nmv` of the position.

**position_pnl** Position P&L, calculated as shares * (end_price + dividend + distribution - start_price)

**trading_pnl** The difference between the market value of trades at the benchmark price and at the end price. Note: currently assuming benchmark price is the closing price, so trading P&L is zero.

**trade_costs** Trade costs, calculated as a fixed percentage (set in the simulation configuration) of the notional of the market trade (valued at the close).

**financing_costs** Financing cost for the position, calculated as a fixed percentage (set in the simulation configuration) of the notional of the starting value of the portfolio's external positions. External positions are positions held on the street and are recorded in the ext_shares column.

**gross_pnl** Gross P&L, calculated as position_pnl + trading_pnl.

**net_pnl** Net P&L, calculated as gross_pnl - trade_costs - financing_costs.

**market_order_nmv** Net market value of the order that does not net down with orders in other strategies.

**market_fill_gmv** Gross market value of the order that does not net down with orders in other strategies.

**market_fill_nmv** Net market value of the fill that does not net down with orders in other strategies.

**market_fill_gmv** Gross market value of the fill that does not net down with orders in other strategies.

**transfer_fill_nmv** Net market value of the fill that nets down with fills in other strategies.

**transfer_fill_gmv** Gross market value of the fill that nets down with fills in other strategies.

**start_nmv** Net market value of the position at the start of the period.

**end_nmv** Net market value of the position at the end of the period.

**end_gmv** Gross market value of the position at the end of the period.

**Method** `getPositionSummary()`**:** Get summary information by security. This method can be used, for example, to calculate the biggest winners and losers over the course of the simulation.

*Usage:*

```
Simulation$getPositionSummary(strategy_name = NULL)
```

*Arguments:*

`strategy_name` Character vector of length 1 that specifies the strategy for which to get detail data. If `NULL` data for all strategies is returned. Defaults to `NULL`.

*Returns:* An object of class `data.frame` that contains summary information aggregated by security. The data frame contains the following columns:

**id** Security identifier.

**strategy** Strategy name, or 'joint' for the aggregate strategy.

**gross_pnl** Gross P&L for the position over the entire simulation.

**gross_pnl** Net P&L for the position over the entire simulation.

**average_market_value** Average net market value of the position over days in the simulation where the position was not flat.

**total_trading** Total gross market value of trades for the security.

**trade_costs** Total cost of trades for the security over the entire simulation.

**trade_costs** Total cost of financing for the position over the entire simulation.

**days_in_portfolio** Total number of days there was a position in the security in the portfolio over the entire simulation.

**Method** `getInputStats()`**:** Get input statistics.

*Usage:*

```
Simulation$getInputStats()
```

*Returns:* An object of class `data.frame` that contains statistics on select columns of input data. Statistics are tracked for the columns listed in the configuration variable `simulator/input_data/track_metadata`. The data frame contains the following columns:

**period** Period to which statistics pertain.

**input_rows** Total number of rows of input data, including rows carried forward from the previous period.

**cf_rows** Total number of rows carried forward from the previous period.

**num_na_*column*** Number of NA values in *column*. This measure appears for each element of `track_metadata`.

**cor_*column*** Period-over-period correlation for *column*. This measure appears for each element of `track_metadata`.

**Method** `getLooseningInfo()`**:** Get loosening information.

*Usage:*

```
Simulation$getLooseningInfo()
```

*Returns:* An object of class `data.frame` that contains, for each period, which constraints were loosened in order to solve the portfolio optimization problem, if any. The data frame contains the following columns:

**date** Date for which the constraint was loosened.

**constraint_name** Name of the constraint that was loosened.

**pct_loosened** Percentage by which the constraint was loosened, where 100 means loosened fully (i.e., the constraint is effectively removed).

**Method** `getOptimizationSummary()`**:** Get optimization summary information.

*Usage:*

```
Simulation$getOptimizationSummary()
```

*Returns:* An object of class `data.frame` that contains optimization summary information, such as starting and ending factor constraint values, at the strategy and joint level. The data frame contains the following columns:

**strategy** Strategy name, or 'joint' for the aggregate strategy.

**sim_date** Date to which the data pertains.

**order_gmv** Total gross market value of orders generated by the optimization.

**start_smv** Total net market value of short positions at the start of the optimization.

**start_lmv** Total net market value of long positions at the start of the optimization.

**end_smv** Total net market value of short positions at the end of the optimization.

**end_lmv** Total net market value of long positions at the end of the optimization.

**start_*factor*** Total net exposure to *factor* at the start of the optimization, for each factor constraint.

**end_*factor*** Total net exposure to *factor* at the start of the optimization, for each factor constraint.

**Method** `getExposures()`**:** Get end-of-period exposure information.

*Usage:*

```
Simulation$getExposures(type = "net")
```

*Arguments:*

`type` Vector of length 1 that may be one of `"net"`, `"long"`, `"short"`, and `"gross"`.

*Returns:* An object of class `data.frame` that contains end-of-period exposure information for the simulation portfolio. The units of the exposures are portfolio weight relative to strategy_captial (i.e., net market value of exposure divided by strategy capital). The data frame contains the following columns:

**strategy** Strategy name, or 'joint' for the aggregate strategy.

**sim_date** Date of the exposure data.

***category_level*** Exposure to *level* within *category*, for all levels of all category constraints, at the end of the period.

***factor*** Exposure to *factor*, for all factor constraints, at the end of the period.

**Method** `getDelistings()`**:** Get information on positions removed due to delisting.

*Usage:*

```
Simulation$getDelistings()
```

*Returns:* An object of class `data.frame` that contains a row for each position that is removed from the simulation portfolio due to a delisting. Each row contains the size of the position on the day on which it was removed from the portfolio.

**Method** `getSingleStrategySummaryDf()`**:** Get summary information for a single strategy suitable for plotting input.

*Usage:*

```
Simulation$getSingleStrategySummaryDf(
  strategy_name = "joint",
  include_zero_row = TRUE
)
```

*Arguments:*

`strategy_name` Strategy for which to return summary data.

`include_zero_row` Logical flag indicatiing whether to prepend a row to the summary data with starting values at zero. Defaults to `TRUE`.

*Returns:* A data frame that contains summary information for the desired strategy, as well as columns for cumulative net and gross total return, calculated as pnl divided by ending gross market value.

**Method** `plotPerformance()`**:** Draw a plot of cumulative gross and net return by date.

*Usage:*

```
Simulation$plotPerformance(strategy_name = "joint")
```

*Arguments:*

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `"joint"`.

**Method** `plotContribution()`**:** Draw a plot of contribution to net return on GMV for levels of a specified category.

*Usage:*

```
Simulation$plotContribution(category_var, strategy_name = "joint")
```

*Arguments:*

`category_var` Plot performance contribution for the levels of `category_var`. `category_var` must be present in the simulation's security reference, and detail data must be present in the object's result data.

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `"joint"`.

**Method** `plotMarketValue()`**:** Draw a plot of total gross, long, short, and net market value by date.

*Usage:*

```
Simulation$plotMarketValue(strategy_name = "joint")
```

*Arguments:*

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `"joint"`.

**Method** `plotCategoryExposure()`**:** Draw a plot of exposure to all levels in a category by date.

*Usage:*

```
Simulation$plotCategoryExposure(in_var, strategy_name = "joint")
```

*Arguments:*

`in_var` Category for which exposures are plotted. In order to plot exposures for category `in_var`, we must have run the simulation with `in_var` in the config setting `simulator/calculate_exposures/cate`

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `"joint"`.

**Method** `plotFactorExposure()`**:** Draw a plot of exposure to factors by date.

*Usage:*

```
Simulation$plotFactorExposure(in_var, strategy_name = "joint")
```

*Arguments:*

`in_var` Factors for which exposures are plotted.

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `"joint"`.

**Method** `plotNumPositions()`**:** Draw a plot of number of long and short positions by date.

*Usage:*

```
Simulation$plotNumPositions(strategy_name = "joint")
```

*Arguments:*

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `"joint"`.

**Method** `plotTurnover()`**:** Draw a plot of number of long and short positions by date.

*Usage:*

```
Simulation$plotTurnover(strategy_name = "joint")
```

*Arguments:*

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `"joint"`.

**Method** `plotUniverseSize()`**:** Draw a plot of the universe size, or number of investable stocks, over time.

*Usage:*

```
Simulation$plotUniverseSize(strategy_name = "joint")
```

*Arguments:*

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `joint`.

**Method** `plotNonInvestablePct()`**:** Draw a plot of the percentage of portfolio GMV held in non-investable stocks (e.g., stocks that do not satisfy universe criteria) for a given strategy. Note that this plot requires detail data.

*Usage:*

```
Simulation$plotNonInvestablePct(strategy_name = "joint")
```

*Arguments:*

`strategy_name` Character vector of length 1 specifying the strategy for the plot. Defaults to `"joint"`.

**Method** `overallStatsDf()`**:** Calculate overall simulation summary statistics, such as total P&L, Sharpe, average market values and counts, etc.

*Usage:*

```
Simulation$overallStatsDf()
```

*Returns:* A data frame that contains summary statistics, suitable for reporting.

**Method** `overallReturnsByMonthDf()`**:** Calculate return for each month and summary statistics for each year, such as total return and annualized Sharpe. Return in data frame format suitable for reporting.

*Usage:*

```
Simulation$overallReturnsByMonthDf()
```

*Returns:* The data frame contains one row for each calendar year in the simulation, and up to seventeen columns: one column for year, one column for each calendar month, and columns for the year's total return, annualized return, annualized volatility, and annualized Sharpe. Total return is the sum of daily net returns. Annualized return is the mean net return times 252. Annualized volatility is the standard deviation of net return times the square root of 252. Annualized Sharpe is the ratio of annualized return to annualized volatility. All returns are in percent.

**Method** `print()`**:** Print overall simulation statistics.

*Usage:*

```
Simulation$print()
```

**Method** `writeFeather()`**:** Write the data in the object to feather files.

*Usage:*

```
Simulation$writeFeather(out_loc)
```

*Arguments:*

`out_loc` Directory in which output files should be created.

*Returns:* No return value, called for side effects.

**Method** `readFeather()`**:** Load files created with `writeFeather` into the object. Note that because detail data is not re-split by period, it will not be possible to use the `sim_date` parameter when calling `getSimDetail` on the populated object.

*Usage:*

```
Simulation$readFeather(in_loc)
```

*Arguments:*

`in_loc` Directory that contains files to be loaded.

*Returns:* No return value, called for side effects.

**Method** `getConfig()`**:** Get the object's configuration information.

*Usage:*

```
Simulation$getConfig()
```

*Returns:* Object of class `list` that contains the simulation's configuration information.

**Method** `writeReport()`**:** Write an html document of simulation results.

*Usage:*

```
Simulation$writeReport(
  out_dir,
  out_file,
  out_fmt = "html",
  contrib_vars = NULL
)
```

*Arguments:*

`out_dir` Directory in which output files should be created

`out_file` File name for output

`out_fmt` Format in which output files should be created. The default is html and that is currently the only option.

`contrib_vars` Security reference variables for which to plot return contribution.

`res` The object of class 'Simulation' which we want to write the report about.

**Method** `clone()`**:** The objects of this class are cloneable with this method.

*Usage:*

```
Simulation$clone(deep = FALSE)
```

*Arguments:*

`deep` Whether to make a deep clone.

# Index