Package 'COINr'

July 21, 2025

Type Package

Title Composite Indicator Construction and Analysis

Version 1.1.14

Maintainer William Becker <william.becker@bluefoxdata.eu>

Description A comprehensive high-level package, for composite indicator construction and analysis. It is a ``development environment''
for composite indicators and scoreboards, which includes utilities for construction (indicator selection, denomination, imputation,
data treatment, normalisation, weighting and aggregation) and analysis (multivariate analysis, correlation plotting, short cuts for
principal component analysis, global sensitivity analysis, and more). A composite indicator is completely encapsulated inside a single
hierarchical list called a ``coin''. This allows a fast and efficient work flow, as well as making quick copies, testing methodological variations and making comparisons.
It also includes many plotting options, both statistical (scatter plots, distribution plots) as well as for presenting results.

License MIT + file LICENSE

Encoding UTF-8

URL https://bluefoxr.github.io/COINr/

BugReports https://github.com/bluefoxr/COINr/issues

LazyData true

RoxygenNote 7.2.3

Imports openxlsx (>= 4.2.3), stats, rlang (>= 0.4.10), ggplot2 (>= 3.3.3), readxl (>= 1.3.1), utils

Depends R (>= 4.0.0)

Suggests rmarkdown, spelling, knitr, testthat (>= 3.0.0), matrixStats, performance, covr

VignetteBuilder knitr

Language en-GB

Config/testthat/edition 3

NeedsCompilation no

Author William Becker [aut, cre, cph] (ORCID:
<https://orcid.org/0000-0002-6467-4472>)

Repository CRAN

Date/Publication 2024-05-21 16:00:02 UTC

Contents

`Aggregate` *Aggregate data*

Description

Methods for aggregating numeric vectors, data frames, coins and purses. See individual method documentation for more details:

Usage

```
Aggregate(x, ...)
```

Arguments

`x`	Object to be aggregated
`...`	Further arguments to be passed to methods.

Details

- `Aggregate.data.frame()`
- `Aggregate.coin()`
- `Aggregate.purse()`

Value

An object similar to the input

Examples

```
# see individual method documentation
```

Aggregate.coin *Aggregate indicators in a coin*

Description

Aggregates a named data set specified by `dset` using aggregation function(s) `f_ag`, weights `w`, and optional function parameters `f_ag_para`. Note that COINr has a number of aggregation functions built in, all of which are of the form `a_*()`, e.g. `a_amean()`, `a_gmean()` and friends.

Usage

```
## S3 method for class 'coin'
Aggregate(
  x,
  dset,
  f_ag = NULL,
  w = NULL,
  f_ag_para = NULL,
  dat_thresh = NULL,
  by_df = FALSE,
  out2 = "coin",
  write_to = NULL,
  ...
)
```

Arguments

`x`	A coin class object.
`dset`	The name of the data set to apply the function to, which should be accessible in `.$Data`.
`f_ag`	The name of an aggregation function, a string. This can either be a single string naming a function to use for all aggregation levels, or else a character vector of function names of length `n-1`, where `n` is the number of levels in the index structure. In this latter case, a different aggregation function may be used for each level in the index: the first in the vector will be used to aggregate from Level 1 to Level 2, the second from Level 2 to Level 3, and so on.
`w`	An optional data frame of weights. If `f_ag` does not require accept weights, set to `"none"`. Alternatively, can be the name of a weight set found in `.$Meta$Weights`. This can also be specified as a list specifying the aggregation weights for each level, in the same way as the previous parameters.
`f_ag_para`	Optional parameters to pass to `f_ag`, other than `x` and `w`. As with `f_ag`, this can specified to have different parameters for each aggregation level by specifying as a nested list of length `n-1`. See details.
`dat_thresh`	An optional data availability threshold, specified as a number between 0 and 1. If a row within an aggregation group has data availability lower than this threshold,

	the aggregated value for that row will be NA. Data availability, for a row x_row is defined as sum(!is.na(x_row))/length(x_row), i.e. the fraction of non-NA values. Can also be specified as a vector of length n-1, where n is the number of levels in the index structure, to specify different data availability thresholds by level.
by_df	Controls whether to send a numeric vector to f_ag (if FALSE, default) or a data frame (if TRUE) - see details. Can also be specified as a logical vector of length n-1, where n is the number of levels in the index structure.
out2	Either "coin" (default) to return updated coin or "df" to output the aggregated data set.
write_to	If specified, writes the aggregated data to .$Data[[write_to]]. Default write_to = "Aggregated".
...	arguments passed to or from other methods.

Details

When by_df = FALSE, aggregation is performed row-wise using the function f_ag, such that for each row x_row, the output is f_ag(x_row, f_ag_para), and for the whole data frame, it outputs a numeric vector. Otherwise if by_df = TRUE, the entire data frame of each indicator group is passed to f_ag.

The function f_ag must be supplied as a string, e.g. "a_amean", and it must take as a minimum an input x which is either a numeric vector (if by_df = FALSE), or a data frame (if by_df = TRUE). In the former case f_ag should return a single numeric value (i.e. the result of aggregating x), or in the latter case a numeric vector (the result of aggregating the whole data frame in one go).

Weights are passed to the function f_ag as an argument named w. This means that the function should have arguments that look like f_ag(x, w, ...), where ... are possibly other input arguments to the function. If the aggregation function doesn't use weights, you can set w = "none", and no weights will be passed to it.

f_ag can optionally have other parameters, apart from x and w, specified as a list in f_ag_para.

The aggregation specifications can be set to be different for each level of aggregation: the arguments f_ag, f_ag_para, dat_thresh, w and by_df can all be optionally specified as vectors or lists of length n-1, where n is the number of levels in the index. In this case, the first value in each vector/list will be used for the first round of aggregation, i.e. from indicators to the aggregates at level 2. The next will be used to aggregate from level 2 to level 3, and so on.

When different functions are used for different levels, it is important to get the list syntax correct. For example, in a case with three aggregations using different functions, say we want to use a_amean() for the first two levels, then a custom function f_cust() for the last. f_cust() has some additional parameters a and b. In this case, we would specify e.g. f_ag_para = list(NULL, NULL, list(a = 2, b = 3)) - this is becauase a_amean() requires no additional parameters, so we pass NULL.

Note that COINr has a number of aggregation functions built in, all of which are of the form a_*(), e.g. a_amean(), a_gmean() and friends. To see a list browse COINr functions alphabetically or type a_ in the R Studio console and press the tab key (after loading COINr), or see the online documentation.

Optionally, a data availability threshold can be assigned below which the aggregated value will return NA (see dat_thresh argument). If by_df = TRUE, this will however be ignored because aggregation is not done on individual rows. Note that more complex constraints could be built into f_ag if needed.

Value

An updated coin with aggregated data set added at .$Data[[write_to]] if out2 = "coin", else if out2 = "df" outputs the aggregated data set as a data frame.

Examples

```
# build example up to normalised data set
coin <- build_example_coin(up_to = "Normalise")

# aggregate normalised data set
coin <- Aggregate(coin, dset = "Normalised")
```

Aggregate.data.frame *Aggregate data frame*

Description

Aggregates a data frame into a single column using a specified function. Note that COINr has a number of aggregation functions built in, all of which are of the form a_*(), e.g. a_amean(), a_gmean() and friends.

Usage

```
## S3 method for class 'data.frame'
Aggregate(
  x,
  f_ag = NULL,
  f_ag_para = NULL,
  dat_thresh = NULL,
  by_df = FALSE,
  ...
)
```

Arguments

x	Data frame to be aggregated
f_ag	The name of an aggregation function, as a string.
f_ag_para	Any additional parameters to pass to f_ag, as a named list.

dat_thresh	An optional data availability threshold, specified as a number between 0 and 1. If a row of x has data availability lower than this threshold, the aggregated value for that row will be NA. Data availability, for a row x_row is defined as sum(!is.na(x_row))/length(x_row), i.e. the fraction of non-NA values.
by_df	Controls whether to send a numeric vector to f_ag (if FALSE, default) or a data frame (if TRUE) - see details.
...	arguments passed to or from other methods.

Details

Aggregation is performed row-wise using the function f_ag, such that for each row x_row, the output is f_ag(x_row, f_ag_para), and for the whole data frame, it outputs a numeric vector. The data frame x must only contain numeric columns.

The function f_ag must be supplied as a string, e.g. "a_amean", and it must take as a minimum an input x which is either a numeric vector (if by_df = FALSE), or a data frame (if by_df = TRUE). In the former case f_ag should return a single numeric value (i.e. the result of aggregating x), or in the latter case a numeric vector (the result of aggregating the whole data frame in one go).

f_ag can optionally have other parameters, e.g. weights, specified as a list in f_ag_para.

Note that COINr has a number of aggregation functions built in, all of which are of the form a_*(), e.g. a_amean(), a_gmean() and friends. To see a list browse COINr functions alphabetically or type a_ in the R Studio console and press the tab key (after loading COINr), or see the online documentation.

Optionally, a data availability threshold can be assigned below which the aggregated value will return NA (see dat_thresh argument). If by_df = TRUE, this will however be ignored because aggregation is not done on individual rows. Note that more complex constraints could be built into f_ag if needed.

Value

A numeric vector

Examples

```
# get some indicator data - take a few columns from built in data set
X <- ASEM_iData[12:15]

# normalise to avoid zeros - min max between 1 and 100
X <- Normalise(X,
               global_specs = list(f_n = "n_minmax",
                                   f_n_para = list(l_u = c(1,100))))

# aggregate using harmonic mean, with some weights
y <- Aggregate(X, f_ag = "a_hmean", f_ag_para = list(w = c(1, 1, 2, 1)))
```

`Aggregate.purse` *Aggregate indicators*

Description

Aggregates indicators following the structure specified in `iMeta`, for each coin inside the purse. See `Aggregate.coin()`, which is applied to each coin, for more information

Usage

```
## S3 method for class 'purse'
Aggregate(
  x,
  dset,
  f_ag = NULL,
  w = NULL,
  f_ag_para = NULL,
  dat_thresh = NULL,
  write_to = NULL,
  by_df = FALSE,
  ...
)
```

Arguments

`x`	A purse-class object
`dset`	The name of the data set to apply the function to, which should be accessible in `.$Data`.
`f_ag`	The name of an aggregation function, a string. This can either be a single string naming a function to use for all aggregation levels, or else a character vector of function names of length `n-1`, where `n` is the number of levels in the index structure. In this latter case, a different aggregation function may be used for each level in the index: the first in the vector will be used to aggregate from Level 1 to Level 2, the second from Level 2 to Level 3, and so on.
`w`	An optional data frame of weights. If `f_ag` does not require accept weights, set to `"none"`. Alternatively, can be the name of a weight set found in `.$Meta$Weights`. This can also be specified as a list specifying the aggregation weights for each level, in the same way as the previous parameters.
`f_ag_para`	Optional parameters to pass to `f_ag`, other than `x` and `w`. As with `f_ag`, this can specified to have different parameters for each aggregation level by specifying as a nested list of length `n-1`. See details.
`dat_thresh`	An optional data availability threshold, specified as a number between 0 and 1. If a row within an aggregation group has data availability lower than this threshold, the aggregated value for that row will be `NA`. Data availability, for a row `x_row` is defined as `sum(!is.na(x_row))/length(x_row)`, i.e. the fraction of non-`NA`

	values. Can also be specified as a vector of length n-1, where n is the number of levels in the index structure, to specify different data availability thresholds by level.
write_to	If specified, writes the aggregated data to .$Data[[write_to]]. Default write_to = "Aggregated".
by_df	Controls whether to send a numeric vector to f_ag (if FALSE, default) or a data frame (if TRUE) - see details. Can also be specified as a logical vector of length n-1, where n is the number of levels in the index structure.
...	arguments passed to or from other methods.

Value

An updated purse with new treated data sets added at .$Data[[write_to]] in each coin.

Examples

```
# build example purse up to normalised data set
purse <- build_example_purse(up_to = "Normalise", quietly = TRUE)

# aggregate using defaults
purse <- Aggregate(purse, dset = "Normalised")
```

---

approx_df *Interpolate time-indexed data frame*

Description

Given a numeric data frame Y with rows indexed by a time vector tt, interpolates at time values specified by the vector tt_est. If tt_est is not in tt, will create new rows in the data frame corresponding to these interpolated points.

Usage

```
approx_df(Y, tt, tt_est = NULL, ...)
```

Arguments

Y	A data frame with all numeric columns
tt	A time vector with length equal to nrow(Y), indexing the rows in Y.
tt_est	A time vector of points to interpolate in Y. If NULL, will attempt to interpolate all points in Y (you may need to adjust the rule argument of stats::approx() here). Note that points not specified in tt_est will not be interpolated. tt_est does not need to be a subset of tt.
...	Further arguments to pass to stats::approx() other than x, y and xout.

Details

This is a wrapper for `stats::approx()`, with some differences. In the first place, `stats::approx()` is applied to each column of `Y`, using `tt` each time as the corresponding time vector indexing `Y`. Interpolated values are generated at points specified in `tt_est` but these are appended to the existing data (whereas `stats::approx()` will only return the interpolated points and nothing else). Further arguments to `stats::approx()` can be passed using the `...` argument.

Value

A list with:

- `.$tt` the vector of time points, including time values of interpolated points
- `.$Y` the corresponding interpolated data frame

Both outputs are sorted by `tt`.

Examples

```
# a time vector
tt <- 2011:2020

# two random vectors with some missing values
y1 <- runif(10)
y2 <- runif(10)
y1[2] <- y1[5] <- NA
y2[3] <- y2[5] <- NA
# make into df
Y <- data.frame(y1, y2)

# interpolate for time = 2012
Y_int <- approx_df(Y, tt, 2012)
Y_int$Y

# notice Y_int$y2 is unchanged since at 2012 it did not have NA value
stopifnot(identical(Y_int$Y$y2, y2))

# interpolate at value not in tt
approx_df(Y, tt, 2015.5)
```

`ASEM_COIN` *ASEM COIN (COINr < v1.0)*

Description

This is an "old format" "COIN" object which is stored for testing purposes. It is generated using the COINr6 package (only available on GitHub) using `COINr6::build_ASEM()`

Usage

```
ASEM_COIN
```

Format

A "COIN" class object

Source

https://github.com/bluefoxr/COINr6

`ASEM_iData` *ASEM raw indicator data*

Description

A data set containing raw values of indicators for 51 countries, groups and denominators. See the ASEM Portal for further information and detailed description of each indicator. See also `vignette("coins")` for the format of this data.

Usage

```
ASEM_iData
```

Format

A data frame with 51 rows and 60 variables.

Details

This data set is in the new v1.0 format.

Source

https://composite-indicators.jrc.ec.europa.eu/asem-sustainable-connectivity/repository

ASEM_iData_p *ASEM raw panel data*

Description

This is an artificially-generated set of panel data (multiple observations of indicators over time) that is included to build the example "purse" class, i.e. to build composite indicators over time. This will eventually be replaced with a better example, i.e. a real data set.

Usage

```
ASEM_iData_p
```

Format

A data frame with 255 rows and 60 variables.

Details

This data set is in the new v1.0 format.

Source

https://composite-indicators.jrc.ec.europa.eu/asem-sustainable-connectivity/repository

ASEM_iMeta *ASEM indicator metadata*

Description

This contains all metadata for ASEM indicators, including names, weights, directions, etc. See the ASEM Portal for further information and detailed description of each indicator. See also `vignette("coins")` for the format of this data.

Usage

```
ASEM_iMeta
```

Format

A data frame with 68 rows and 9 variables

Details

This data set is in the new v1.0 format.

Source

https://bluefoxr.github.io/COINrDoc/coins-the-currency-of-coinr.html#aggregation-metadata

---

a_amean *Weighted arithmetic mean*

Description

The vector of weights `w` is relative since the formula is:

Usage

```
a_amean(x, w)
```

Arguments

`x`	A numeric vector.
`w`	A vector of numeric weights of the same length as `x`.

Details

$$y = \frac{1}{\sum w_i} \sum w_i x_i$$

If `x` contains `NA`s, these `x` values and the corresponding `w` values are removed before applying the formula above.

Value

The weighted mean as a scalar value

Examples

```
x <- c(1:10)
w <- c(10:1)
a_amean(x,w)
```

a_copeland *Copeland scores*

Description

Aggregates a data frame of indicator values into a single column using the Copeland method. This function calls `outrankMatrix()`.

Usage

```
a_copeland(X, w = NULL)
```

Arguments

`X`	A numeric data frame or matrix of indicator data, with observations as rows and indicators as columns. No other columns should be present (e.g. label columns).
`w`	A numeric vector of weights, which should have length equal to `ncol(X)`. Weights are relative and will be re-scaled to sum to 1. If `w` is not specified, defaults to equal weights.

Details

The outranking matrix is transformed as follows:

- values > 0.5 are replaced by 1
- values < 0.5 are replaced by -1
- values == 0.5 are replaced by 0
- the diagonal of the matrix is all zeros

The Copeland scores are calculated as the row sums of this transformed matrix.

This function replaces the now-defunct `copeland()` from COINr < v1.0.

Value

Numeric vector of Copeland scores.

Examples

```
# some example data
ind_data <- COINr::ASEM_iData[12:16]

# aggregate with vector of weights
outlist <- outrankMatrix(ind_data)
```

a_genmean *Weighted generalised mean*

Description

Weighted generalised mean of a vector. NA are skipped by default.

Usage

```
a_genmean(x, w = NULL, p)
```

Arguments

x	A numeric vector of positive values.
w	A vector of weights, which should have length equal to `length(x)`. Weights are relative and will be re-scaled to sum to 1. If w is not specified, defaults to equal weights.
p	Coefficient - see details.

Details

The generalised mean is as follows:

$$y = \left(\frac{1}{\sum w_i} \sum w_i x_i^p \right)^{1/p}$$

where p is a coefficient specified in the function argument here. Note that:

- For negative p, all x values must be positive
- Setting `p = 0` will result in an error due to the negative exponent. This case is equivalent to the geometric mean in the limit, so use `a_gmean()` instead.

Value

Weighted harmonic mean, as a numeric value.

Examples

```
# a vector of values
x <- 1:10
# a vector of weights
w <- runif(10)
# cubic mean
a_genmean(x,w, p = 2)
```

a_gmean *Weighted geometric mean*

Description

Weighted geometric mean of a vector. NA are skipped by default.

Usage

```
a_gmean(x, w = NULL)
```

Arguments

x	A numeric vector of positive values.
w	A vector of weights, which should have length equal to `length(x)`. Weights are relative and will be re-scaled to sum to 1. If w is not specified, defaults to equal weights.

Details

This function replaces the now-defunct `geoMean()` from COINr < v1.0.

Value

The geometric mean, as a numeric value.

Examples

```
# a vector of values
x <- 1:10
# a vector of weights
w <- runif(10)
# weighted geometric mean
a_gmean(x,w)
```

a_hmean *Weighted harmonic mean*

Description

Weighted harmonic mean of a vector. NA are skipped by default.

Usage

```
a_hmean(x, w = NULL)
```

Arguments

x	A numeric vector of positive values.
w	A vector of weights, which should have length equal to `length(x)`. Weights are relative and will be re-scaled to sum to 1. If `w` is not specified, defaults to equal weights.

Details

This function replaces the now-defunct `harMean()` from COINr < v1.0.

Value

Weighted harmonic mean, as a numeric value.

Examples

```
# a vector of values
x <- 1:10
# a vector of weights
w <- runif(10)
# weighted harmonic mean
a_hmean(x,w)
```

---

boxcox *Box Cox transformation*

Description

Simple Box Cox, with no optimisation of lambda.

Usage

```
boxcox(x, lambda, makepos = TRUE, na.rm = FALSE)
```

Arguments

x	A vector or column of data to transform
lambda	The lambda parameter of the Box Cox transform
makepos	If `TRUE` (default) makes all values positive by subtracting the minimum and adding 1.
na.rm	If `TRUE`, `NA`s will be removed: only relevant if `makepos = TRUE` which invokes `min()`.

Details

This function replaces the now-defunct `BoxCox()` from COINr < v1.0.

Value

A vector of length `length(x)` with transformed values.

Examples

```
# example data
x <- runif(30)
# Apply Box Cox
xBox <- boxcox(x, lambda = 2)
# plot one against the other
plot(x, xBox)
```

`build_example_coin` *Build ASEM example coin*

Description

Shortcut function to build the ASEM example coin, using inbuilt example data. This can be useful for testing and also for building reproducible examples. To see the underlying commands run `edit(build_example_coin)`. See also `vignette("coins")`.

Usage

```
build_example_coin(up_to = NULL, quietly = FALSE)
```

Arguments

`up_to`	The point up to which to build the index. If `NULL`, builds full index. Else specify a building function (as a string) - the index will be built up to and including this function. This option is mainly for helping with function examples. Example: `up_to = "Normalise"`.
`quietly`	If `TRUE`, suppresses all messages.

Details

This function replaces the now-defunct `build_ASEM()` from COINr < v1.0.

Value

coin class object

Examples

```
# build example coin up to data treatment step
coin <- build_example_coin(up_to = "Treat")
coin
```

build_example_purse *Build example purse*

Description

Shortcut function to build an example purse. This is currently an "artificial" example, in that it takes the ASEM data set used in `build_example_coin()` and replicates it for five years, adding artificial noise to simulate year-on-year variation. This was done simply to demonstrate the functionality of purses, and will at some point be replaced with a real example. See also `vignette("coins")`.

Usage

```
build_example_purse(up_to = NULL, quietly = FALSE)
```

Arguments

up_to	The point up to which to build the index. If `NULL`, builds full index. Else specify a `build_*` function (as a string) - the index will be built up to and including this function. This option is mainly for helping with function examples. Example: `up_to = "build_normalise"`.
quietly	If `TRUE`, suppresses all messages.

Value

purse class object

Examples

```
# build example purse up to unit screening step
purse <- build_example_purse(up_to = "Screen")
purse
```

CAGR *Compound annual growth rate*

Description

Given a variable `y` indexed by a time vector `x`, calculates the compound annual growth rate. Note that CAGR assumes that the `x` refer to years. Also it is only calculated using the first and latest observed values.

Usage

```
CAGR(y, x)
```

Arguments

y	A numeric vector
x	A numeric vector of the same length as y, indexing y in time. No NA values are allowed in x. This vector is assumed to be years, otherwise the result must be interpreted differently.

Value

A scalar value (CAGR)

Examples

```
# random points over 10 years
x <- 2011:2020
y <- runif(10)

CAGR(y, x)
```

---

change_ind *Add and remove indicators*

Description

A shortcut function to add and remove indicators. This will make the relevant changes and recalculate the index if asked. Adding and removing is done relative to the current set of indicators used in calculating the index results. Any indicators that are added must of course be present in the original iData and iMeta that were input to new_coin().

Usage

```
change_ind(coin, add = NULL, drop = NULL, regen = FALSE)
```

Arguments

coin	coin object
add	A character vector of indicator codes to add (must be present in the original input data)
drop	A character vector of indicator codes to remove (must be present in the original input data)
regen	Logical (default): if TRUE, automatically regenerates the results based on the new specs Otherwise, just updates the .$Log parameters. This latter might be useful if you want to Make other changes before re-running using the Regen() function.

Details

See also `vignette("adjustments")`.

This function replaces the now-defunct `indChange()` from COINr < v1.0.

Value

An updated coin, with regenerated results if `regen = TRUE`.

Examples

```
# build full example coin
coin <- build_example_coin(quietly = TRUE)

# exclude two indicators and regenerate
# remove two indicators and regenerate the coin
coin_remove <- change_ind(coin, drop = c("LPI", "Forest"), regen = TRUE)

coin_remove
```

---

check_iData *Check iData*

Description

Checks the format of `iData` input to `new_coin()`. This check must be passed to successfully build a new coin.

Usage

```
check_iData(iData, quietly = FALSE)
```

Arguments

`iData`	A data frame of indicator data.
`quietly`	Set `TRUE` to suppress message if input is valid.

Details

The restrictions on `iData` are not extensive. It should be a data frame with only one required column `uCode` which gives the code assigned to each unit (alphanumeric, not starting with a number). All other columns are defined by corresponding entries in `iMeta`, with the following special exceptions:

- `Time` is an optional column which allows panel data to be input, consisting of e.g. multiple rows for each `uCode`: one for each `Time` value. This can be used to split a set of panel data into multiple coins (a so-called "purse") which can be input to COINr functions. See `new_coin()` for more details.

- uName is an optional column which specifies a longer name for each unit. If this column is not included, unit codes (uCode) will be used as unit names where required.

No column names should contain blank spaces.

Value

Message if everything ok, else error messages.

Examples

```
check_iData(ASEM_iData)
```

`check_iMeta` *Check iMeta*

Description

Checks the format of iMeta input to new_coin(). This performs a series of thorough checks to make sure that iMeta agrees with the specifications. This also includes checks to make sure the structure makes sense, there are no duplicates, and other things. iMeta must pass this check to build a new coin.

Usage

```
check_iMeta(iMeta, quietly = FALSE)
```

Arguments

iMeta	A data frame of indicator metadata. See details.
quietly	Set TRUE to suppress message if input is valid.

Details

Required columns for iMeta are:

- Level: Level in aggregation, where 1 is indicator level, 2 is the level resulting from aggregating indicators, 3 is the result of aggregating level 2, and so on. Set to NA for entries that are not included in the index (groups, denominators, etc).
- iCode: Indicator code, alphanumeric. Must not start with a number or contain blank spaces.
- Parent: Group (iCode) to which indicator/aggregate belongs in level immediately above. Each entry here should also be found in iCode. Set to NA only for the highest (Index) level (no parent), or for entries that are not included in the index (groups, denominators, etc).
- Direction: Numeric, either -1 or 1
- Weight: Numeric weight, will be rescaled to sum to 1 within aggregation group. Set to NA for entries that are not included in the index (groups, denominators, etc).

- `Type`: The type, corresponding to `iCode`. Can be either `Indicator`, `Aggregate`, `Group`, `Denominator`, or `Other`.

Optional columns that are recognised in certain functions are:

- `iName`: Name of the indicator: a longer name which is used in some plotting functions.
- `Unit`: the unit of the indicator, e.g. USD, thousands, score, etc. Used in some plots if available.
- `Target`: a target for the indicator. Used if normalisation type is distance-to-target.

The `iMeta` data frame essentially gives details about each of the columns found in `iData`, as well as details about additional data columns eventually created by aggregating indicators. This means that the entries in `iMeta` must include *all* columns in `iData`, *except* the three special column names: `uCode`, `uName`, and `Time`. In other words, all column names of `iData` should appear in `iMeta$iCode`, except the three special cases mentioned. The `iName` column optionally can be used to give longer names to each indicator which can be used for display in plots.

`iMeta` also specifies the structure of the index, by specifying the parent of each indicator and aggregate. The `Parent` column must refer to entries that can be found in `iCode`. Try `View(ASEM_iMeta)` for an example of how this works.

`Level` is the "vertical" level in the hierarchy, where 1 is the bottom level (indicators), and each successive level is created by aggregating the level below according to its specified groups.

`Direction` is set to 1 if higher values of the indicator should result in higher values of the index, and -1 in the opposite case.

The `Type` column specifies the type of the entry: `Indicator` should be used for indicators at level 1. `Aggregate` for aggregates created by aggregating indicators or other aggregates. Otherwise set to `Group` if the variable is not used for building the index but instead is for defining groups of units. Set to `Denominator` if the variable is to be used for scaling (denominating) other indicators. Finally, set to `Other` if the variable should be ignored but passed through. Any other entries here will cause an error.

Note: this function requires the columns above as specified, but extra columns can also be added without causing errors.

Value

Message if everything ok, else error messages.

Examples

```
check_iMeta(ASEM_iMeta)
```

check_SkewKurt *Check skew and kurtosis of a vector*

Description

Logical test: if abs(skewness) < skew_thresh OR kurtosis < kurt_thresh, returns TRUE, else FALSE

Usage

```
check_SkewKurt(x, na.rm = FALSE, skew_thresh = 2, kurt_thresh = 3.5)
```

Arguments

x	A numeric vector.
na.rm	Set TRUE to remove NA values, otherwise returns NA.
skew_thresh	A threshold for absolute skewness (positive). Default 2.25.
kurt_thresh	A threshold for kurtosis. Default 3.5.

Value

A list with .$Pass is a Logical, where TRUE is pass, FALSE is fail, and .$Details is a sub-list with skew and kurtosis values.

Examples

```
set.seed(100)
x <- runif(20)
# this passes
check_SkewKurt(x)
# if we add an outlier, doesn't pass
check_SkewKurt(c(x, 1000))
```

COIN_to_coin *Convert a COIN to a coin*

Description

Converts an older COIN class to the newer coin class. Note that there are some limitations to this. First, the function arguments used to create the COIN will not be passed to the coin, since the function arguments are different. This means that any data sets beyond "Raw" cannot be regenerated. The second limitation is that anything from the .$Analysis folder will not be passed on.

Usage

```
COIN_to_coin(COIN, recover_dsets = FALSE, out2 = "coin")
```

Arguments

COIN	A COIN class object, generated by COINr version <= 0.6.1, OR a list containing IndData, IndMeta and AggMeta entries.
recover_dsets	Logical: if TRUE, will recover data sets other than "Raw" which are found in the .$Data list.
out2	If "coin" (default) outputs a coin, else if "list", outputs a list with iData and iMeta entries. This may be useful if you want to make further edits before building the coin.

Details

This function works by building the iData and iMeta arguments to new_coin(), using information from the COIN. It then uses these to build a coin if out2 = "coin" or else outputs both data frames in a list.

If recover_dsets = TRUE, any data sets found in COIN$Data (except "Raw") will also be put in coin$Data, in the correct format. These can be used to inspect the data but not to regenerate.

Note that if you want to exclude any indicators, you will have to set out2 = "list" and build the coin in a separate step with exclude specified. Any exclusions/inclusions from the COIN are not passed on automatically.

Value

A coin class object if out2 = "coin", else a list of data frames if out2 = "list".

Examples

```
# see vignette("other_functions")
```

---

compare_coins *Compare two coins*

Description

Compares two coin class objects using a specified iCode (column of data) from specified data sets.

Usage

```
compare_coins(
  coin1,
  coin2,
  dset,
  iCode,
  also_get = NULL,
  compare_by = "ranks",
  sort_by = NULL,
  decreasing = FALSE
)
```

Arguments

coin1	A coin class object
coin2	A coin class object
dset	A data set that is found in .$Data.
iCode	The name of a column that is found in .$Data[[dset]].
also_get	Optional metadata columns to attach to the table: see get_data().
compare_by	Either "ranks" which produces a comparison using ranks, or else "scores", which instead uses scores. Note that scores may be very different if the methodology is different from one coin to another, e.g. for different normalisation methods.
sort_by	Optionally, a column name of the output data frame to sort rows by. Can be either "coin.1", "coin.2", "Diff", "Abs.diff" or possibly a column name imported using also_get.
decreasing	Argument to pass to order(): how to sort.

Details

This function replaces the now-defunct compTable() from COINr < v1.0.

Value

A data frame of comparison information.

Examples

```
# build full example coin
coin <- build_example_coin(quietly = TRUE)

# copy coin
coin2 <- coin

# change to prank function (percentile ranks)
# we don't need to specify any additional parameters (f_n_para) here
coin2$Log$Normalise$global_specs <- list(f_n = "n_prank")
```

```
# regenerate
coin2 <- Regen(coin2)

# compare index, sort by absolute rank difference
compare_coins(coin, coin2, dset = "Aggregated", iCode = "Index",
              sort_by = "Abs.diff", decreasing = TRUE)
```

`compare_coins_corr` *Compare two coins by correlation*

Description

Given two coins, this function returns the correlation between the two coins, for target datset `dset` and target indicator code(s) `iCodes`. Correlation is calculated as the Pearson correlation coefficient, but if `compare_by = "Ranks"` then this is the correlation coefficient of the ranks, which amounts to the Spearman rank correlation. Set `compare_by = "Scores"` to return the Pearson correlation between scores.

Usage

```
compare_coins_corr(coin1, coin2, dset, iCodes, compare_by = "ranks")
```

Arguments

`coin1`	A coin
`coin2`	A coin, with possibly alternative methodology. This should share at least two units in common with `coin1`.
`dset`	Target data set, must be present in both `coin1` and `coin2`
`iCodes`	Character vector of indicator codes to correlate between the two coins.
`compare_by`	Either `"Ranks"` or `"Scores"`.

Value

A list containing a correlation table and a list of comparison data frames.

Examples

```
# build example
coin <- build_example_coin()

# copy coin
coin2 <- coin

# change to prank function (percentile ranks)
# we don't need to specify any additional parameters (f_n_para) here
coin2$Log$Normalise$global_specs <- list(f_n = "n_prank")
```

```
# regenerate
coin2 <- Regen(coin2)

# iCodes to compare: all at level 3 and 4
iCodes <- coin$Meta$Ind$iCode[which(coin$Meta$Ind$Level > 2)]

# compare index, sort by absolute rank difference
l_comp <- compare_coins_corr(coin, coin2, dset = "Aggregated", iCodes = iCodes)

# see df
l_comp$df_corr
```

`compare_coins_multi` *Compare multiple coins*

Description

Given multiple coins as a list, generates a rank comparison of a single indicator or aggregate which is specified by the `dset` and `iCode` arguments (passed to `get_data()`). The indicator or aggregate targeted must be available in all the coins in `coins`.

Usage

```
compare_coins_multi(
  coins,
  dset,
  iCode,
  also_get = NULL,
  tabtype = "Values",
  ibase = 1,
  sort_table = TRUE,
  compare_by = "ranks"
)
```

Arguments

`coins`	A list of coins. If names are provided, these will be used in the tables returned by this function.
`dset`	The name of a data set found in `.$Data`. See `get_data()`.
`iCode`	A column name of the data set targeted by `dset`. See `get_data()`.
`also_get`	Optional metadata columns to attach to the table: see `get_data()`. If this is not specified, the results of each coin will be merged using the `uCodes` within each coin. If this is specified, results will be merged additionally using the metadata columns. This means that coins must share the same metadata columns that are returned as a result of `also_get`.

tabtype	The type of table to generate. One of: • "Values": returns a data frame of rank values for each coin provided, plus ISO3 column • "Diffs": returns a data frame of rank differences between the base coin and each other coin (see ibase) • "AbsDiffs": as "Diffs" but absolute rank differences are returned • "All": returns all of the three previous rank tables, as a list of data frames
ibase	The index of the coin to use as a base comparison (default first coin in list)
sort_table	If TRUE, sorts by the base COIN (ibase) (default).
compare_by	Either "ranks" which produces a comparison using ranks, or else "scores", which instead uses scores. Note that scores may be very different if the methodology is different from one coin to another, e.g. for different normalisation methods.

Details

By default, the ranks of the target indicator/aggregate of each coin will be merged using the uCodes within each coin. Optionally, specifying also_get (passed to get_data()) will additionally merge using the metadata columns. This means that coins must share the same metadata columns that are returned as a result of also_get.

This function replaces the now-defunct compTableMulti() from COINr < v1.0.

Value

Data frame unless tabtype = "All", in which case a list of three data frames is returned.

Examples

```
# see vignette("adjustments")
```

---

compare_df *Compare two data frames*

Description

A custom function for comparing two data frames of indicator data, to see whether they match up, at a specified number of significant figures. Specifically, this is intended to compare two data frames, without regard to row or column ordering. Rows are matched by the required matchcol argument. Hence, it is different from e.g. all.equal() which requires rows to be ordered. In COINr, typically matchcol is the uCode column, for example.

Usage

```
compare_df(df1, df2, matchcol, sigfigs = 5)
```

Arguments

df1	A data frame
df2	Another data frame
matchcol	A common column name that is used to match row order. E.g. this might be uCode.
sigfigs	The number of significant figures to use for matching numerical columns

Details

This function compares numerical and non-numerical columns to see if they match. Rows and columns can be in any order. The function performs the following checks:

- Checks that the two data frames are the same size
- Checks that column names are the same, and that the matching column has the same entries
- Checks column by column that the elements are the same, after sorting according to the matching column

It then summarises for each column whether there are any differences, and also what the differences are, if any.

This is intended to cross-check results. For example, if you run something in COINr and want to check indicator results against external calculations.

This function replaces the now-defunct `compareDF()` from COINr < v1.0.

Value

A list with comparison results. List contains:

- `.$Same`: overall summary: if `TRUE` the data frames are the same according to the rules specified, otherwise `FALSE`.
- `.$Details`: details of each column as a data frame. Each row summarises a column of the data frame, saying whether the column is the same as its equivalent, and the number of differences, if any. In case the two data frames have differing numbers of columns and rows, or have differing column names or entries in `matchcol`, `.$Details` will simply contain a message to this effect.
- `.$Differences`: a list with one entry for every column which contains different entries. Differences are summarised as a data frame with one row for each difference, reporting the value from `df1` and its equivalent from `df2`.

Examples

```
# take a sample of indicator data (including the uCode column)
data1 <- ASEM_iData[c(2,12:15)]
# copy the data
data2 <- data1
# make a change: replace one value in data2 by NA
data2[1,2] <- NA
# compare data frames
compare_df(data1, data2, matchcol = "uCode")
```

Custom *Custom operation*

Description

Allows a custom data operation on coins or purses.

Usage

```
Custom(x, ...)
```

Arguments

x	Object to be operated on (coin or purse)
...	arguments passed to or from other methods.

Value

Modified object.

Custom.coin *Custom operation*

Description

Custom operation on a coin. This is an experimental new feature so please check the results carefully.

Usage

```
## S3 method for class 'coin'
Custom(
  x,
  dset,
  f_cust,
  f_cust_para = NULL,
  write_to = NULL,
  write2log = TRUE,
  ...
)
```

Arguments

x	A coin
dset	Target data set
f_cust	Function to apply to the data set. See details.
f_cust_para	Optional additional parameters to pass to the function defined by f_cust.
write_to	Name of data set to write to
write2log	Logical: whether or not to write to the log.
...	Arguments to pass to/from other methods.

Details

In this function, the data set named dset is extracted from the coin using get_dset(coin, dset). It is passed to the function f_cust, which is required to return an equivalent but modified data frame, which is then written as a new data set with name write_to. This is intended to allow arbitrary operations on coin data sets while staying within the COINr framework, which means that if Regen() is used, these operations will be re-run, allowing them to be included in things like sensitivity analysis.

The format of f_cust is important. It must be a function whose first argument is called x: this will be the argument that the data is passed to. The data will be in the same format as extracted via get_dset(coin, dset), which means it will have a uCode column. f_cust can have other arguments which are passed to it via f_cust_para. The function should return a data frame similar to the data that was passed to it, it must contain have the same column names (meaning you can't remove indicators), but otherwise is flexible - this means some caution is necessary to ensure that subsequent operations don't fail. Be careful, for example, to ensure that there are no duplicates in uCode, and that indicator columns are numeric.

The function assigned to f_cust is passed to base::do.call(), therefore it can be passed either as a string naming the function, or as the function itself. Depending on the context, the latter option may be preferable because this stores the function within the coin, which makes it portable. Otherwise, if the function is simply named as a string, you must make sure it is available to access in the environment.

Value

A coin

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin")

# create function - replaces suspected unreliable point with NA
f_NA <- function(x){ x[3, 10] <- NA; return(x)}

# call function from Custom()
coin <- Custom(coin, dset = "Raw", f_cust = f_NA)
stopifnot(is.na(coin$Data$Custom[3,10]))
```

Custom.purse *Custom operation*

Description

Custom operation on a purse. This is an experimental new feature.

Usage

```
## S3 method for class 'purse'
Custom(
  x,
  dset,
  f_cust,
  f_cust_para = NULL,
  global = FALSE,
  write_to = NULL,
  ...
)
```

Arguments

x	A purse object
dset	The data set to apply the operation to.
f_cust	Function to apply to the data set. See details.
f_cust_para	Optional additional parameters to pass to the function defined by f_cust.
global	Logical: if TRUE, the entire data set, over all time points, is passed to the function f_cust. This is useful if the custom operation should be different for different time points, for example. Otherwise if FALSE, passes the data set within each coin one at a time to f_cust.
write_to	Name of data set to write to
...	Arguments to pass to/from other methods.

Details

In this function, the data set named dset is extracted from the coin using get_dset(purse, dset). It is passed to the function f_cust, which is required to return an equivalent but modified data frame, which is then written as a new data set with name write_to. This is intended to allow arbitrary operations on coin data sets while staying within the COINr framework, which means that if Regen() is used, these operations will be re-run, allowing them to be included in things like sensitivity analysis.

The format of f_cust is important. It must be a function whose first argument is called x: this will be the argument that the data is passed to. The data will be in the same format as extracted via get_dset(purse, dset), which means it will have uCode and Time columns. f_cust can have other arguments which are passed to it via f_cust_para. The function should return a data frame

similar to the data that was passed to it, it must contain have the same column names (meaning you can't remove indicators), but otherwise is flexible - this means some caution is necessary to ensure that subsequent operations don't fail. Be careful, for example, to ensure that there are no duplicates in `uCode`, and that indicator columns are numeric.

The function assigned to `f_cust` is passed to `base::do.call()`, therefore it can be passed either as a string naming the function, or as the function itself. Depending on the context, the latter option may be preferable because this stores the function within the coin, which makes it portable. Otherwise, if the function is simply named as a string, you must make sure it is available to access in the environment.

Value

An updated purse.

Examples

```
# build example purse
purse <- build_example_purse(up_to = "new_coin")

# custom function - set points before 2020 to NA for BEL in FDI due to a
# break in the series
f_cust <- function(x){x[(x$uCode == "BEL") & (x$Time < 2020), "FDI"] <- NA;
                      return(x)}
```

`Denominate` *Denominate data*

Description

"Denominates" or "scales" variables by other variables. Typically this is done by dividing extensive variables such as GDP by a scaling variable such as population, to give an intensive variable (GDP per capita).

Usage

```
Denominate(x, ...)
```

Arguments

`x`	Object to be denominated
`...`	arguments passed to or from other methods

Details

See documentation for individual methods:

- `Denominate.data.frame()`
- `Denominate.coin()`
- `Denominate.purse()`.

This function replaces the now-defunct `denominate()` from COINr < v1.0.

Value

See individual method documentation

Examples

```
# See individual method documentation
```

`Denominate.coin` *Denominate data set in a coin*

Description

"Denominates" or "scales" indicators by other variables. Typically this is done by dividing extensive variables such as GDP by a scaling variable such as population, to give an intensive variable (GDP per capita).

Usage

```
## S3 method for class 'coin'
Denominate(
  x,
  dset,
  denoms = NULL,
  denomby = NULL,
  denoms_ID = NULL,
  f_denom = NULL,
  write_to = NULL,
  out2 = "coin",
  ...
)
```

Arguments

x	A coin class object
dset	The name of the data set to apply the function to, which should be accessible in .$Data.
denoms	An optional data frame of denominator data. Columns should be denominator data, with column names corresponding to entries in denomby. This must also include an ID column identified by denoms_ID to match rows. If denoms is not specified, will extract any potential denominator columns that were attached to iData when calling new_coin().
denomby	Optional data frame which specifies which denominators to use for each indicator, and any scaling factors to apply. Should have columns iCode, Denominator, ScaleFactor. iCode specifies an indicator code found in dset, Denominator specifies a column name from denoms to use to denominate the corresponding column from x. ScaleFactor allows the possibility to scale denominators if needed, and specifies a factor to multiply the resulting values by. For example, if GDP is a denominator and is measured in dollars, dividing will create very small numbers (order 1e-10 and smaller) which could cause problems with numerical precision. If denomby is not specified, specifications will be taken from the "Denominator" column in iMeta, if it exists.
denoms_ID	An ID column for matching denoms with the data to be denominated. This column should contain uMeta codes to match with the data set extracted from the coin.
f_denom	A function which takes two numeric vector arguments and is used to perform the denomination for each column. By default, this is division, i.e. x[[col]]/denoms[[col]] for given columns, but any function can be passed that takes two numeric vectors as inputs and returns a single numeric vector. See details.
write_to	If specified, writes the aggregated data to .$Data[[write_to]]. Default write_to = "Denominated".
out2	Either "coin" (default) to return updated coin or "df" to output the aggregated data set.
...	arguments passed to or from other methods

Details

This function denominates a data set dset inside the coin. By default, denominating variables are taken from the coin, specifically as variables in iData with Type = "Denominator" in iMeta (input to new_coin()). Specifications to map denominators to indicators are also taken by default from iMeta$Denominator, if it exists.

These specifications can be overridden using the denoms and denomby arguments. The operator for denomination can also be changed using the f_denom argument.

See also documentation for Denominate.data.frame() which is called by this method.

Value

An updated coin if out2 = "coin", else a data frame of denominated data if out2 = "df".

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# denominate (here, we only need to say which dset to use, takes
# specs and denominators from within the coin)
coin <- Denominate(coin, dset = "Raw")
```

`Denominate.data.frame` *Denominate data sets by other variables*

Description

"Denominates" or "scales" variables by other variables. Typically this is done by dividing extensive variables such as GDP by a scaling variable such as population, to give an intensive variable (GDP per capita).

Usage

```
## S3 method for class 'data.frame'
Denominate(
  x,
  denoms,
  denomby,
  x_ID = NULL,
  denoms_ID = NULL,
  f_denom = NULL,
  ...
)
```

Arguments

`x`	A data frame of data to be denominated. Columns to be denominated must be numeric, but any columns not specified in `denomby` will be ignored. `x` must also contain an ID column specified by `x_ID` to match rows with `denoms`.
`denoms`	A data frame of denominator data. Columns should be denominator data, with column names corresponding to entries in `denomby`. This must also include an ID column identified by `denoms_ID` to match rows.
`denomby`	A data frame which specifies which denominators to use for each indicator, and any scaling factors to apply. Should have columns `iCode`, `Denominator`, `ScaleFactor`. `iCode` specifies a column name from `x`, `Denominator` specifies a column name from `denoms` to use to denominate the corresponding column from `x`. `ScaleFactor` allows the possibility to scale denominators if needed, and specifies a factor to multiply the resulting values by. For example, if GDP is a denominator and is measured in dollars, dividing will create very small numbers (order 1e-10 and smaller) which could cause problems with numerical precision.

x_ID	A column name of x to use to match rows with denoms. Default is "uCode".
denoms_ID	A column name of denoms to use to match rows with x. Default is "uCode".
f_denom	A function which takes two numeric vector arguments and is used to perform the denomination for each column. By default, this is division, i.e. x[[col]]/denoms[[col]] for given columns, but any function can be passed that takes two numeric vectors as inputs and returns a single numeric vector. See details.
...	arguments passed to or from other methods.

Details

A data frame x is denominated by variables found in another data frame denoms, according to specifications in denomby. denomby specifies which columns in x are to be denominated, and by which columns in denoms, and any scaling factors to apply to each denomination.

Both x and denomby must contain an ID column which matches the rows of x to denomby. If not specified, this is assumed to be uCode, but can also be specified using the x_ID and denoms_ID arguments. All entries in x[[x_ID]] must be present in denoms[[denoms_ID]], although extra rows are allowed in denoms. This is because the rows of x are matched to the rows of denoms using these ID columns, to ensure that units (rows) are correctly denominated.

By default, columns of x are divided by columns of denoms. This can be generalised by setting f_denom to another function which takes two numeric vector arguments. I.e. setting denoms = ``*`` will multiply columns of x and denoms together.

Value

A data frame of the same size as x, with any specified columns denominated according to specifications.

See Also

- WorldDenoms A data set of some common national-level denominators.

Examples

```
# Get a sample of indicator data (note must be indicators plus a "UnitCode" column)
iData <- ASEM_iData[c("uCode", "Goods", "Flights", "LPI")]
# Also get some denominator data
denoms <- ASEM_iData[c("uCode", "GDP", "Population")]
# specify how to denominate
denomby <- data.frame(iCode = c("Goods", "Flights"),
Denominator = c("GDP", "Population"),
ScaleFactor = c(1, 1000))
# Denominate one by the other
iData_den <- Denominate(iData, denoms, denomby)
```

`Denominate.purse` *Denominate a data set within a purse.*

Description

This works in almost exactly the same way as `Denominate.coin()`. The only point of care is that the `denoms` argument here cannot take time-indexed data, but only a single value for each unit. It is therefore recommended to pass the time-dependent denominator data as part of `iData` when calling `new_coin()`. In this way, denominators can vary with time. See `vignette("denomination")`.

Usage

```
## S3 method for class 'purse'
Denominate(
  x,
  dset,
  denoms = NULL,
  denomby = NULL,
  denoms_ID = NULL,
  f_denom = NULL,
  write_to = NULL,
  ...
)
```

Arguments

`x`	A purse class object
`dset`	The name of the data set to apply the function to, which should be accessible in `.$Data`.
`denoms`	An optional data frame of denominator data. Columns should be denominator data, with column names corresponding to entries in `denomby`. This must also include an ID column identified by `denoms_ID` to match rows. If `denoms` is not specified, will extract any potential denominator columns that were attached to `iData` when calling `new_coin()`.
`denomby`	Optional data frame which specifies which denominators to use for each indicator, and any scaling factors to apply. Should have columns `iCode`, `Denominator`, `ScaleFactor`. `iCode` specifies an indicator code found in `dset`, `Denominator` specifies a column name from `denoms` to use to denominate the corresponding column from `x`. `ScaleFactor` allows the possibility to scale denominators if needed, and specifies a factor to multiply the resulting values by. For example, if GDP is a denominator and is measured in dollars, dividing will create very small numbers (order 1e-10 and smaller) which could cause problems with numerical precision. If `denomby` is not specified, specifications will be taken from the "Denominator" column in `iMeta`, if it exists.

denoms_ID	An ID column for matching denoms with the data to be denominated. This column should contain uMeta codes to match with the data set extracted from the coin.
f_denom	A function which takes two numeric vector arguments and is used to perform the denomination for each column. By default, this is division, i.e. x[[col]]/denoms[[col]] for given columns, but any function can be passed that takes two numeric vectors as inputs and returns a single numeric vector. See details.
write_to	If specified, writes the aggregated data to .$Data[[write_to]]. Default write_to = "Denominated".
...	arguments passed to or from other methods.

Value

An updated purse

Examples

```
# build example purse
purse <- build_example_purse(up_to = "new_coin", quietly = TRUE)

# denominate using data/specs already included in coin
purse <- Denominate(purse, dset = "Raw")
```

---

export_to_excel *Export a coin or purse to Excel*

Description

Writes coins and purses to Excel. See individual method documentation:

Usage

```
export_to_excel(x, fname, ...)
```

Arguments

x	A coin or purse
fname	The file name to write to
...	Arguments passed to/from methods

Details

This function replaces the now-defunct coin2Excel() from COINr < v1.0.

- export_to_excel.coin()
- export_to_excel.purse()

Value

An Excel spreadsheet.

Examples

```
# see individual method documentation
```

---

export_to_excel.coin *Export a coin to Excel*

Description

Exports the contents of the coin to Excel. This writes all data frames inside the coin to Excel, with each data frame on a separate tab. Tabs are named according to the position in the coin object. You can write other data frames by simply attaching them to the coin object somewhere.

Usage

```
## S3 method for class 'coin'
export_to_excel(x, fname = "coin_export.xlsx", include_log = FALSE, ...)
```

Arguments

x	A coin class object
fname	The file name/path to write to, as a character string
include_log	Logical: if TRUE, also writes data frames from the .$Log list inside the coin.
...	arguments passed to or from other methods.

Value

.xlsx file at specified path

Examples

```
## Here we write a COIN to Excel, but this is done to a temporary directory
## to avoid "polluting" the working directory when running automatic tests.
## In a real case, set fname to a directory of your choice.

# build example coin up to data treatment step
coin <- build_example_coin(up_to = "Treat")

# write to Excel in temporary directory
export_to_excel(coin, fname = paste0(tempdir(), "\\ASEM_results.xlsx"))

# spreadsheet is at:
print(paste0(tempdir(), "\\ASEM_results.xlsx"))
```

```
# now delete temporary file to keep things tidy in testing
unlink(paste0(tempdir(),"\\ASEM_results.xlsx"))
```

`export_to_excel.purse` *Export a purse to Excel*

Description

Exports the contents of the purse to Excel. This is similar to the coin method `export_to_excel.coin()`, but combines data sets from various time points. It also selectively writes metadata since this may be spread across multiple coins.

Usage

```
## S3 method for class 'purse'
export_to_excel(x, fname = "coin_export.xlsx", include_log = FALSE, ...)
```

Arguments

`x`	A purse class object
`fname`	The file name/path to write to, as a character string
`include_log`	Logical: if `TRUE`, also writes data frames from the `.$Log` list inside the coin.
`...`	arguments passed to or from other methods.

Value

.xlsx file at specified path

Examples

```
#
```

`get_corr` *Get correlations*

Description

Helper function for getting correlations between indicators and aggregates. This retrieves subsets of correlation matrices between different aggregation levels, in different formats. By default, it will return a long-form data frame, unless `make_long = FALSE`. By default, any correlations with a p-value less than 0.05 are replaced with `NA`. See `pval` argument to adjust this.

Usage

```
get_corr(
  coin,
  dset,
  iCodes = NULL,
  Levels = NULL,
  ...,
  cortype = "pearson",
  pval = 0.05,
  withparent = FALSE,
  grouplev = NULL,
  make_long = TRUE,
  use_directions = FALSE
)
```

Arguments

coin	A coin class coin object
dset	The name of the data set to apply the function to, which should be accessible in .$Data.
iCodes	An optional list of character vectors where the first entry specifies the indicator/aggregate codes to correlate against the second entry (also a specification of indicator/aggregate codes). If this is specified as a character vector it will coerced to the first entry of a list, i.e. list(iCodes).
Levels	The aggregation levels to take the two groups of indicators from. See get_data() for details. Defaults to indicator level.
...	Further arguments to be passed to get_data() (uCodes and use_group).
cortype	The type of correlation to calculate, either "pearson", "spearman", or "kendall".
pval	The significance level for including correlations. Correlations with $p > pval$ will be returned as NA. Default 0.05. Set to 0 to disable this.
withparent	If TRUE, and aglev[1] != aglev[2], will only return correlations of each row with its parent. Alternatively, if withparent = "family", will return correlations with parents, grandparents etc, up to the highest level. In both cases the data set must be aggregated for this to work.
grouplev	The aggregation level to group correlations by if aglev[1] == aglev[2]. Requires that make_long = TRUE.
make_long	Logical: if TRUE, returns correlations in long format (default), else if FALSE returns in wide format. Note that if wide format is requested, features specified by grouplev and withparent are not supported.
use_directions	Logical: if TRUE the extracted data is adjusted using directions found inside the coin (i.e. the "Direction" column input in iMeta: any indicators with negative direction will have their values multiplied by -1 which will reverse the direction of correlation). This should only be set to TRUE if the data set has not yet been normalised. For example, this can be useful to set to TRUE to analyse correlations in the raw data, but would make no sense to analyse correlations in the

normalised data because that already has the direction adjusted! So you would reverse direction twice. In other words, use this at your discretion.

Details

This function allows you to obtain correlations between any subset of indicators or aggregates, from any data set present in a coin. Indicator selection is performed using `get_data()`. Two different indicator sets can be correlated against each other by specifying `iCodes` and `Levels` as vectors.

The correlation type can be specified by the `cortype` argument, which is passed to `stats::cor()`.

The `withparent` argument will optionally only return correlations which correspond to the structure of the index. For example, if `Levels = c(1,2)` (i.e. we wish to correlate indicators from Level 1 with aggregates from Level 2), and we set `withparent = TRUE`, only the correlations between each indicator and its parent group will be returned (not correlations between indicators and other aggregates to which it does not belong). This can be useful to check whether correlations of an indicator/aggregate with any of its parent groups exceeds or falls below thresholds.

Similarly, the `grouplev` argument can be used to restrict correlations to within groups corresponding to the index structure. Setting e.g. `grouplev = 2` will only return correlations within the groups defined at Level 2.

The `grouplev` and `withparent` options are disabled if `make_long = FALSE`.

Note that this function can only call correlations within the same data set (i.e. only one data set in `.$Data`).

This function replaces the now-defunct `getCorr()` from COINr < v1.0.

Value

A data frame of pairwise correlation values in wide or long format (see `make_long`). Correlations with $p > pval$ will be returned as `NA`.

See Also

- `plot_corr()` Plot correlation matrices of indicator subsets

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# get correlations
cmat <- get_corr(coin, dset = "Raw", iCodes = list("Environ"),
                 Levels = 1, make_long = FALSE)
```

`get_corr_flags` *Find highly-correlated indicators within groups*

Description

This returns a data frame of any highly correlated indicators within the same aggregation group. The level of the aggregation grouping can be controlled by the `grouplev` argument.

Usage

```
get_corr_flags(
  coin,
  dset,
  cor_thresh = 0.9,
  thresh_type = "high",
  cortype = "pearson",
  grouplev = NULL,
  roundto = 3,
  use_directions = FALSE
)
```

Arguments

`coin`	A coin class object
`dset`	The name of the data set to apply the function to, which should be accessible in `.$Data`.
`cor_thresh`	A threshold to flag high correlation. Default 0.9.
`thresh_type`	Either `"high"`, which will only flag correlations *above* `cor_thresh`, or `"low"`, which will only flag correlations *below* `cor_thresh`.
`cortype`	The type of correlation, either `"pearson"` (default), `"spearman"` or `"kendall"`. See stats::cor.
`grouplev`	The level to group indicators in. E.g. if `grouplev = 2` it will look for high correlations between indicators that belong to the same group in Level 2.
`roundto`	Number of decimal places to round correlations to. Default 3. Set `NULL` to disable rounding.
`use_directions`	Logical: if `TRUE` the extracted data is adjusted using directions found inside the coin (i.e. the "Direction" column input in `iMeta`. See comments on this argument in `get_corr()`.

Details

This function is motivated by the idea that having very highly-correlated indicators within the same group may amount to double counting, or possibly redundancy in the framework.

This function replaces the now-defunct `hicorrSP()` from COINr < v1.0.

Value

A data frame with one entry for every indicator pair that is highly correlated within the same group, at the specified level. Pairs are only reported once, i.e. only uses the upper triangle of the correlation matrix.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "Normalise", quietly = TRUE)

# get correlations between indicator over 0.75 within level 2 groups
get_corr_flags(coin, dset = "Normalised", cor_thresh = 0.75,
               thresh_type = "high", grouplev = 2)
```

---

get_cronbach *Cronbach's alpha*

Description

Calculates Cronbach's alpha, a measure of statistical reliability. Cronbach's alpha is a simple measure of "consistency" of a data set, where a high value implies higher reliability/consistency. The selection of indicators via `get_data()` allows to calculate the measure on any group of indicators or aggregates.

Usage

```
get_cronbach(coin, dset, iCodes, Level, ..., use = "pairwise.complete.obs")
```

Arguments

`coin`	A coin or a data frame containing only numerical columns of data.
`dset`	The name of the data set to apply the function to, which should be accessible in `.$Data`.
`iCodes`	Indicator codes to retrieve. If `NULL` (default), returns all iCodes found in the selected data set. See `get_data()`.
`Level`	The level in the hierarchy to extract data from. See `get_data()`.
`...`	Further arguments passed to `get_data()`, other than those explicitly specified here.
`use`	Argument to pass to stats::cor to calculate the covariance matrix. Default `"pairwise.complete.obs"`.

Details

This function simply returns Cronbach's alpha. If you want a lot more details on reliability, the 'psych' package has a much more detailed analysis.

This function replaces the now-defunct `getCronbach()` from COINr < v1.0.

Value

Cronbach alpha as a numerical value.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# Cronbach's alpha for the "P2P" group
get_cronbach(coin, dset = "Raw", iCodes = "P2P", Level = 1)
```

---

get_data *Get subsets of indicator data*

Description

A helper function to retrieve a named data set from coin or purse objects. See individual method documentation:

Usage

```
get_data(x, ...)
```

Arguments

x	A coin or purse
...	Arguments passed to methods

Details

- get_data.coin()
- get_data.purse()

This function replaces the now-defunct `getIn()` from COINr < v1.0.

Value

Data frame of indicator data, indexed also by time if input is a purse.

Examples

```
# see individual method documentation
```

`get_data.coin` *Get subsets of indicator data*

Description

A flexible function for retrieving data from a coin, from a specified data set. Subsets of data can be returned based on selection of columns, using the `iCodes` and `Level` arguments, and by filtering rowwise using the `uCodes` and `use_group` arguments. The `also_get` argument also allows unit metadata columns to be attached, such as names, groups, and denominators.

Usage

```
## S3 method for class 'coin'
get_data(
  x,
  dset,
  iCodes = NULL,
  Level = NULL,
  uCodes = NULL,
  use_group = NULL,
  also_get = NULL,
  ...
)
```

Arguments

`x`	A coin class object
`dset`	The name of the data set to apply the function to, which should be accessible in `.$Data`.
`iCodes`	Optional indicator codes to retrieve. If `NULL` (default), returns all iCodes found in the selected data set. Can also refer to indicator groups. See details.
`Level`	Optionally, the level in the hierarchy to extract data from. See details.
`uCodes`	Optional unit codes to filter rows of the resulting data set. Can also be used in conjunction with groups. See details.
`use_group`	Optional group to filter rows of the data set. Specified as `list(Group_Var = Group)`, where `Group_Var` is a Group_ column that must be present in the selected data set, and `Group` is a specified group inside that grouping variable. This filters the selected data to only include rows from the specified group. Can also be used in conjunction with `uCodes` – see details.
`also_get`	A character vector specifying any columns to attach to the data set that are *not* indicators or aggregates. These will be e.g. `uName`, groups, denominators or columns labelled as "Other" in `iMeta`. These columns are stored in `.$Meta$Unit` to avoid repetition. Set `also_get = "all"` to attach all columns, or set `also_get = "none"` to return only numeric columns, i.e. no `uCode` column.
`...`	arguments passed to or from other methods.

Details

The iCodes argument can be used to directly select named indicators, i.e. setting iCodes = c("a", "b") will select indicators "a" and "b", attaching any extra columns specified by also_get. However, using this in conjunction with the Level argument returns named groups of indicators. For example, setting iCodes = "Group1" (for e.g. an aggregation group in Level 2) and Level = 1 will return all indicators in Level 1, belonging to "Group1".

Rows can also be subsetted. The uCodes argument can be used to select specified units in the same way as iCodes. Additionally, the use_group argument filters to specified groups. If uCodes is specified, and use_group refers to a named group column, then it will return all units in the groups that the uCodes belong to. This is useful for putting a unit into context with its peers based on some grouping variable.

Note that if you want to retrieve a whole data set (with no column/row subsetting), use the get_dset() function which should be slightly faster.

Value

A data frame of indicator data according to specifications.

Examples

```
# build full example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# get all indicators in "Political group
x <- get_data(coin, dset = "Raw", iCodes = "Political", Level = 1)
head(x, 5)

# see vignette("data_selection") for more examples
```

---

get_data.purse *Get subsets of indicator data*

Description

This retrieves data from a purse. It functions in a similar way to get_data.coin() but has the additional Time argument to allow selection based on the point(s) in time.

Usage

```
## S3 method for class 'purse'
get_data(
  x,
  dset,
  iCodes = NULL,
  Level = NULL,
  uCodes = NULL,
```

```
  use_group = NULL,
  Time = NULL,
  also_get = NULL,
  ...
)
```

Arguments

x	A purse class object
dset	The name of the data set to apply the function to, which should be accessible in .$Data.
iCodes	Optional indicator codes to retrieve. If NULL (default), returns all iCodes found in the selected data set. Can also refer to indicator groups. See details.
Level	Optionally, the level in the hierarchy to extract data from. See details.
uCodes	Optional unit codes to filter rows of the resulting data set. Can also be used in conjunction with groups. See details.
use_group	Optional group to filter rows of the data set. Specified as list(Group_Var = Group), where Group_Var is a Group_ column that must be present in the selected data set, and Group is a specified group inside that grouping variable. This filters the selected data to only include rows from the specified group. Can also be used in conjunction with uCodes – see details.
Time	Optional time index to extract from a subset of the coins present in the purse. Should be a vector containing one or more entries in x$Time or NULL to return all (default).
also_get	A character vector specifying any columns to attach to the data set that are *not* indicators or aggregates. These will be e.g. uName, groups, denominators or columns labelled as "Other" in iMeta. These columns are stored in .$Meta$Unit to avoid repetition. Set also_get = "all" to attach all columns, or set also_get = "none" to return only numeric columns, i.e. no uCode column.
...	arguments passed to or from other methods.

Details

Note that

Value

A data frame of indicator data indexed by a "Time" column.

Examples

```
# build full example purse
purse <- build_example_purse(up_to = "new_coin", quietly = TRUE)

# get specified indicators for specific years, for specified units
get_data(purse, dset = "Raw",
```

```
            iCodes = c("Lang", "Forest"),
            uCodes = c("AUT", "CHN", "DNK"),
            Time = c(2019, 2020))
```

`get_data_avail` *Get data availability of units*

Description

Generic function for getting the data availability of each unit (row).

Usage

```
get_data_avail(x, ...)
```

Arguments

`x`	Either a coin or a data frame
`...`	Arguments passed to other methods

Details

See method documentation:

- `get_data_avail.data.frame()`
- `get_data_avail.coin()`

See also vignettes: `vignette("analysis")` and `vignette("imputation")`.

`get_data_avail.coin` *Get data availability of units*

Description

Returns a list of data frames: the data availability of each unit (row) in a given data set, as well as percentage of zeros. A second data frame gives data availability by aggregation (indicator) groups.

Usage

```
## S3 method for class 'coin'
get_data_avail(x, dset, out2 = "coin", ...)
```

Arguments

x	A coin
dset	String indicating name of data set in .$Data.
out2	Either "coin" to output an updated coin or "list" to output a list.
...	arguments passed to or from other methods.

Details

This function ignores any non-numeric columns, and returns a data availability table of numeric columns with non-numeric columns appended at the beginning.

See also vignettes: vignette("analysis") and vignette("imputation").

Value

An updated coin with data availability tables written in .$Analysis[[dset]], or a list of data availability tables.

Examples

```
# build example coin
coin <-  build_example_coin(up_to = "new_coin", quietly = TRUE)

# get data availability of Raw dset
l_dat <- get_data_avail(coin, dset = "Raw", out2 = "list")
head(l_dat$Summary, 5)
```

---

get_data_avail.data.frame

Get data availability of units

Description

Returns a data frame of the data availability of each unit (row), as well as percentage of zeros. This function ignores any non-numeric columns, and returns a data availability table with non-numeric columns appended at the beginning.

Usage

```
## S3 method for class 'data.frame'
get_data_avail(x, ...)
```

Arguments

x	A data frame
...	arguments passed to or from other methods.

Details

See also vignettes: `vignette("analysis")` and `vignette("imputation")`.

Value

A data frame of data availability statistics for each column of `x`.

Examples

```
# data availability of "airquality" data set
get_data_avail(airquality)
```

---

get_denom_corr *Correlations between indicators and denominators*

Description

Get a data frame containing any correlations between indicators and denominators that exceed a given threshold. This can be useful when *whether* to denominate an indicator and *by what* may not be obvious. If an indicator is strongly correlated with a denominator, this may suggest to denominate it by that denominator.

Usage

```
get_denom_corr(
  coin,
  dset,
  cor_thresh = 0.6,
  cortype = "pearson",
  nround = 2,
  use_directions = FALSE
)
```

Arguments

`coin`	A coin class object.
`dset`	The name of the data set to apply the function to, which should be accessible in `.$Data`.
`cor_thresh`	A correlation threshold: the absolute value of any correlations between indicator-denominator pairs above this threshold will be flagged.
`cortype`	The type of correlation: to be passed to the `method` argument of `stats::cor`.
`nround`	Optional number of decimal places to round correlation values to. Default 2, set `NULL` to disable.
`use_directions`	Logical: if `TRUE` the extracted data is adjusted using directions found inside the coin (i.e. the "Direction" column input in `iMeta`. See comments on this argument in `get_corr()`.

Value

A data frame of pairwise correlations that exceed the threshold.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# get correlations >0.7 of any indicator with denominators
get_denom_corr(coin, dset = "Raw", cor_thresh = 0.7)
```

---

get_dset *Gets a named data set and performs checks*

Description

A helper function to retrieve a named data set from coin or purse objects. See individual documentation on:

Usage

```
get_dset(x, dset, ...)
```

Arguments

x	A coin or purse
dset	A character string corresponding to a named data set within `.$Data`. E.g. `"Raw"`.
...	arguments passed to or from other methods.

Details

- `get_dset.coin()`
- `get_dset.purse()`

Value

Data frame of indicator data, indexed also by time if input is a purse.

Examples

```
# see examples for methods
```

`get_dset.coin` *Gets a named data set and performs checks*

Description

A helper function to retrieve a named data set from the coin object. Also performs input checks at the same time.

Usage

```
## S3 method for class 'coin'
get_dset(x, dset, also_get = NULL, ...)
```

Arguments

`x`	A coin class object
`dset`	A character string corresponding to a named data set within `.$Data`. E.g. `"Raw"`.
`also_get`	A character vector specifying any columns to attach to the data set that are *not* indicators or aggregates. These will be e.g. `uName`, groups, denominators or columns labelled as "Other" in `iMeta`. These columns are stored in `.$Meta$Unit` to avoid repetition. Set `also_get = "all"` to attach all columns, or set `also_get = "none"` to return only numeric columns, i.e. no `uCode` column.
`...`	arguments passed to or from other methods.

Details

If `also_get` is not specified, this will return the indicator columns with the `uCode` identifiers in the first column. Optionally, `also_get` can be specified to attach other metadata columns, or to only return the numeric (indicator) columns with no identifiers. This latter option might be useful for e.g. examining correlations.

Value

Data frame of indicator data.

Examples

```
# build example coin, just up to raw dset for speed
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# retrieve raw data set with added cols
get_dset(coin, dset = "Raw", also_get = c("uName", "GDP_group"))
```

`get_dset.purse` *Gets a named data set and performs checks*

Description

A helper function to retrieve a named data set from a purse object. Retrieves the specified data set from each coin in the purse and joins them together in a single data frame using `rbind()`, indexed with a `Time` column.

Usage

```
## S3 method for class 'purse'
get_dset(x, dset, Time = NULL, also_get = NULL, ...)
```

Arguments

`x`	A purse class object
`dset`	A character string corresponding to a named data set within each coin `.$Data`. E.g. `"Raw"`.
`Time`	Optional time index to extract from a subset of the coins present in the purse. Should be a vector containing one or more entries in `x$Time` or `NULL` to return all (default).
`also_get`	A character vector specifying any columns to attach to the data set that are *not* indicators or aggregates. These will be e.g. `uName`, groups, denominators or columns labelled as "Other" in `iMeta`. These columns are stored in `.$Meta$Unit` to avoid repetition. Set `also_get = "all"` to attach all columns, or set `also_get = "none"` to return only numeric columns, i.e. no `uCode` column.
`...`	arguments passed to or from other methods.

Value

Data frame of indicator data.

Examples

```
# build example purse
purse <- build_example_purse(up_to = "new_coin", quietly = TRUE)

# get raw data set
df1 <- get_dset(purse, dset = "Raw")
```

`get_eff_weights` *Get effective weights*

Description

Calculates the "effective weight" of each indicator and aggregate at the index level. The effective weight is calculated as the final weight of each component in the index, and this is due to not just to its own weight, but also to the weights of each aggregation that it is involved in, plus the number of indicators/aggregates in each group. The effective weight is one way of understanding the final contribution of each indicator to the index. See also `vignette("weights")`.

Usage

```
get_eff_weights(coin, out2 = "df")
```

Arguments

`coin`	A coin class object
`out2`	Either `"coin"` or `"df"`

Details

This function replaces the now-defunct `effectiveWeight()` from COINr < v1.0.

Value

Either an iMeta data frame with effective weights as an added column, or an updated coin with effective weights added to `.$Meta$Ind`.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# get effective weights as data frame
w_eff <- get_eff_weights(coin, out2 = "df")

head(w_eff)
```

`get_noisy_weights` *Noisy replications of weights*

Description

Given a data frame of weights, this function returns multiple replicates of the weights, with added noise. This is intended for use in uncertainty and sensitivity analysis.

Usage

```
get_noisy_weights(w, noise_specs, Nrep)
```

Arguments

`w`	A data frame of weights, in the format found in `.$Meta$Weights`.
`noise_specs`	a data frame with columns: • `Level`: The aggregation level to apply noise to • `NoiseFactor`: The size of the perturbation: setting e.g. 0.2 perturbs by +/- 20% of nominal values.
`Nrep`	The number of weight replications to generate.

Details

Weights are expected to be in a data frame format with columns `Level`, `iCode` and `Weight`, as used in `iMeta`. Note that no `NA`s are allowed anywhere in the data frame.

Noise is added using the `noise_specs` argument, which is specified by a data frame with columns `Level` and `NoiseFactor`. The aggregation level refers to number of the aggregation level to target while the `NoiseFactor` refers to the size of the perturbation. If e.g. a row is `Level = 1` and `NoiseFactor = 0.2`, this will allow the weights in aggregation level 1 to deviate by +/- 20% of their nominal values (the values in `w`).

This function replaces the now-defunct `noisyWeights()` from COINr < v1.0.

Value

A list of `Nrep` sets of weights (data frames).

See Also

- `get_sensitivity()` Perform global sensitivity or uncertainty analysis on a COIN

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# get nominal weights
w_nom <- coin$Meta$Weights$Original

# build data frame specifying the levels to apply the noise at
# here we vary at levels 2 and 3
noise_specs = data.frame(Level = c(2,3),
                         NoiseFactor = c(0.25, 0.25))

# get 100 replications
noisy_wts <- get_noisy_weights(w = w_nom, noise_specs = noise_specs, Nrep = 100)

# examine one of the noisy weight sets, last few rows
tail(noisy_wts[[1]])
```

---

get_opt_weights *Weight optimisation*

Description

This function provides optimised weights to agree with a pre-specified vector of "target importances".

Usage

```
get_opt_weights(
  coin,
  itarg = NULL,
  dset,
  Level,
  cortype = "pearson",
  optype = "balance",
  toler = NULL,
  maxiter = NULL,
  weights_to = NULL,
  out2 = "list"
)
```

Arguments

coin	coin object
itarg	a vector of (relative) target importances. For example, `c(1,2,1)` would specify that the second indicator should be twice as "important" as the other two.

dset	Name of the aggregated data set found in coin$Data which results from calling Aggregate().
Level	The aggregation level to apply the weight adjustment to. This can only be one level.
cortype	The type of correlation to use - can be either "pearson", "spearman" or "kendall". See stats::cor.
optype	The optimisation type. Either "balance", which aims to balance correlations according to a vector of "importances" specified by itarg (default), or "infomax" which aims to maximise overall correlations.
toler	Tolerance for convergence. Defaults to 0.1 (decrease for more accuracy, increase if convergence problems).
maxiter	Maximum number of iterations. Default 500.
weights_to	Name to write the optimised weight set to, if out2 = "coin".
out2	Where to output the results. If "coin" (default for coin input), appends to updated coin, creating a new list of weights in .$Parameters$Weights. Otherwise if "list" outputs to a list (default).

Details

This is a linear version of the weight optimisation proposed in this paper: doi:10.1016/j.ecolind.2017.03.056. Weights are optimised to agree with a pre-specified vector of "importances". The optimised weights are returned back to the coin.

See vignette("weights") for more details on the usage of this function and an explanation of the underlying method. Note that this function calculates correlations without considering statistical significance.

This function replaces the now-defunct weightOpt() from COINr < v1.0.

Value

If out2 = "coin" returns an updated coin object with a new set of weights in .$Meta$Weights, plus details of the optimisation in .$Analysis. Else if out2 = "list" the same outputs (new weights plus details of optimisation) are wrapped in a list.

Examples

```
# build example coin
coin <- build_example_coin(quietly = TRUE)

# check correlations between level 3 and index
get_corr(coin, dset = "Aggregated", Levels = c(3, 4))

# optimise weights at level 3
l_opt <- get_opt_weights(coin, itarg = "equal", dset = "Aggregated",
                         Level = 3, weights_to = "OptLev3", out2 = "list")

# view results
tail(l_opt$WeightsOpt)
```

```
l_opt$CorrResultsNorm
```

`get_PCA` *Perform PCA on a coin*

Description

Performs Principle Component Analysis (PCA) on a specified data set and subset of indicators or aggregation groups. This function has two main outputs: the output(s) of `stats::prcomp()`, and optionally the weights resulting from the PCA. Therefore it can be used as an analysis tool and/or a weighting tool. For the weighting aspect, please see the details below.

Usage

```
get_PCA(
  coin,
  dset = "Raw",
  iCodes = NULL,
  Level = NULL,
  by_groups = TRUE,
  nowarnings = FALSE,
  weights_to = NULL,
  out2 = "list"
)
```

Arguments

`coin`	A coin
`dset`	The name of the data set in `.$Data` to use.
`iCodes`	An optional character vector of indicator codes to subset the indicator data, passed to `get_data()`
`Level`	The aggregation level to take indicator data from. Integer from 1 (indicator level) to N (top aggregation level, typically the index).
`by_groups`	If `TRUE` (default), performs PCA inside each aggregation group inside the specified level. If `FALSE`, performs a single PCA over all indicators/aggregates in the specified level.
`nowarnings`	If `FALSE` (default), will give warnings where missing data are found. Set to `TRUE` to suppress these warnings.
`weights_to`	A string to name the resulting set of weights. If this is specified, and `out2 = "coin"`, will write a new set of "PCA weights" to the `.$Meta$Weights` list. This is experimental - see details. If `NULL`, does not write any weights (default).
`out2`	If the input is a coin object, this controls where to send the output. If `"coin"`, it sends the results to the coin object, otherwise if `"list"`, outputs to a separate list (default).

Details

PCA must be approached with care and an understanding of what is going on. First, let's consider the PCA excluding the weighting component. PCA takes a set of data consisting of variables (indicators) and observations. It then rotates the coordinate system such that in the new coordinate system, the first axis (called the first principal component (PC)) aligns with the direction of maximum variance of the data set. The amount of variance explained by the first PC, and by the next several PCs, can help to understand whether the data can be explained by simpler set of variables. PCA is often used for dimensionality reduction in modelling, for example.

In the context of composite indicators, PCA can be used first as an analysis tool. We can check for example, within an aggregation group, can the indicators mostly be explained by one PC? If so, this gives a little extra justification to aggregating the indicators because the information lost in aggregation will be less. We can also check this over the entire set of indicators.

The complications are in a composite indicator, the indicators are grouped and arranged into a hierarchy. This means that when performing a PCA, we have to decide which level to perform it at, and which groupings to use, if any. The `get_PCA()` function, using the `by_groups` argument, allows to automatically apply PCA by group if this is required.

The output of `get_PCA()` is a PCA object for each of the groups specified, which can then be examined using existing tools in R, see `vignette("analysis")`.

The other output of `get_PCA()` is a set of "PCA weights" if the `weights_to` argument is specified. Here we also need to say some words of caution. First, what constitutes "PCA weights" in composite indicators is not very well-defined. In COINr, a simple option is adopted. That is, the loadings of the first principal component are taken as the weights. The logic here is that these loadings should maximise the explained variance - the implication being that if we use these as weights in an aggregation, we should maximise the explained variance and hence the information passed from the indicators to the aggregate value. This is a nice property in a composite indicator, where one of the aims is to represent many indicators by single composite. See doi:10.1016/j.envsoft.2021.105208 for a discussion on this.

But. The weights that result from PCA have a number of downsides. First, they can often include negative weights which can be hard to justify. Also PCA may arbitrarily flip the axes (since from a variance point of view the direction is not important). In the quest for maximum variance, PCA will also weight the strongest-correlating indicators the highest, which means that other indicators may be neglected. In short, it often results in a very unbalanced set of weights. Moreover, PCA can only be performed on one level at a time.

All these considerations point to the fact: while PCA as an analysis tool is well-established, please use PCA weights with care and understanding of what is going on.

This function replaces the now-defunct `getPCA()` from COINr < v1.0.

Value

If `out2 = "coin"`, results are appended to the coin object. Specifically:

- A list is added to `.$Analysis` containing PCA weights (loadings) of the first principle component, and the output of stats::prcomp, for each aggregation group found in the targeted level.
- If `weights_to` is specified, a new set of PCA weights is added to `.$Meta$Weights` If `out2 = "list"` the same outputs are contained in a list.

See Also

- stats::prcomp Principle component analysis

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# PCA on "Sust" group of indicators
l_pca <- get_PCA(coin, dset = "Raw", iCodes = "Sust",
                 out2 = "list", nowarnings = TRUE)

# Summary of results for one of the sub-groups
summary(l_pca$PCAresults$Social$PCAres)
```

---

get_pvals *P-values for correlations in a data frame or matrix*

Description

This is a stripped down version of the "cor.mtest()" function from the "corrplot" package. It uses the `stats::cor.test()` function to calculate pairwise p-values. Unlike the corrplot version, this only calculates p-values, and not confidence intervals. Credit to corrplot for this code, I only replicate it here to avoid depending on their package for a single function.

Usage

```
get_pvals(X, ...)
```

Arguments

`X`	A numeric matrix or data frame
`...`	Additional arguments passed to function `cor.test()`, e.g. `conf.level = 0.95`.

Value

Matrix of p-values

Examples

```
# a matrix of random numbers, 3 cols
x <- matrix(runif(30), 10, 3)

# get correlations between cols
cor(x)

# get p values of correlations between cols
get_pvals(x)
```

`get_results` *Results summary tables*

Description

Generates fast results tables, either attached to the coin or as a data frame.

Usage

```
get_results(
  coin,
  dset,
  tab_type = "Summ",
  also_get = NULL,
  use = "scores",
  order_by = NULL,
  nround = 2,
  use_group = NULL,
  dset_indicators = NULL,
  out2 = "df"
)
```

Arguments

`coin`	The coin object, or a data frame of indicator data
`dset`	Name of data set in `.$Data`
`tab_type`	The type of table to generate. Either `"Summ"` (a single indicator plus rank), `"Aggs"` (all aggregated scores/ranks above indicator level), or `"Full"` (all scores/ranks plus all group, denominator columns).
`also_get`	Names of further columns to attach to table.
`use`	Either `"scores"` (default), `"ranks"`, or `"groupranks"`. For the latter, `use_group` must be specified.
`order_by`	A code of the indicator or aggregate to sort the table by. If not specified, defaults to the highest aggregate level, i.e. the index in most cases. If `use_group` is specified, rows will also be sorted by the specified group.
`nround`	The number of decimal places to round numerical values to. Defaults to 2.
`use_group`	An optional grouping variable. If specified, the results table includes this group column, and if `use = "groupranks"`, ranks will be returned with respect to the groups in this column.
`dset_indicators`	Optional data set from which to take only indicator (level 1) data from. This can be set to `"Raw"` for example, so that all aggregates come from the aggregated data set, and the indicators come from the raw data set. This can make more sense in presenting results in many cases, so that the "real" indicator data is visible.

out2 If "df", outputs a data frame (tibble). Else if "coin" attaches to .$Results in an updated coin.

Details

Although results are available in a coin in .$Data, the format makes it difficult to quickly present results. This function generates results tables that are suitable for immediate presentation, i.e. sorted by index or other indicators, and only including relevant columns. Scores are also rounded by default, and there is the option to present scores or ranks.

See also vignette("results") for more info.

This function replaces the now-defunct getResults() from COINr < v1.0.

Value

If out2 = "df", the results table is returned as a data frame. If out2 = "coin", this function returns an updated coin with the results table attached to .$Results.

Examples

```
# build full example coin
coin <- build_example_coin(quietly = TRUE)

# get results table
df_results <- get_results(coin, dset = "Aggregated", tab_type = "Aggs")

head(df_results)
```

---

get_sensitivity *Sensitivity and uncertainty analysis of a coin*

Description

This function performs global sensitivity and uncertainty analysis of a coin. You must specify which parameters of the coin to vary, and the alternatives/distributions for those parameters.

Usage

```
get_sensitivity(
  coin,
  SA_specs,
  N,
  SA_type = "UA",
  dset,
  iCode,
  Nboot = NULL,
  quietly = FALSE,
  check_addresses = TRUE
)
```

Arguments

coin	A coin
SA_specs	Specifications of the input uncertainties
N	The number of regenerations
SA_type	The type of analysis to run. "UA" runs an uncertainty analysis. "SA" runs a sensitivity analysis (which anyway includes an uncertainty analysis).
dset	The data set to extract the target variable from (passed to get_data()).
iCode	The variable within dset to use as the target variable (passed to get_data()).
Nboot	Number of bootstrap samples to take when estimating confidence intervals on sensitivity indices.
quietly	Set to TRUE to suppress progress messages.
check_addresses	Logical: if FALSE skips the check of the validity of the parameter addresses. Default TRUE, but useful to set to FALSE if running this e.g. in a Rmd document (because may require user input).

Details

COINr implements a flexible variance-based global sensitivity analysis approach, which allows almost any assumption to be varied, as long as the distribution of alternative values can be described. Variance-based "sensitivity indices" are estimated using a Monte Carlo design (running the composite indicator many times with a particular combination of input values). This follows the methodology described in doi:10.1111/j.1467985X.2005.00350.x.

To understand how this function works, please see vignette("sensitivity"). Here, we briefly recap the main input arguments.

First, you can select whether to run an uncertainty analysis SA_type = "UA" or sensitivity analysis SA_type = "SA". The number of replications (regenerations of the coin) is specified by N. Keep in mind that the *total* number of replications is N for an uncertainty analysis but is N*(d + 2) for a sensitivity analysis due to the experimental design used.

To run either types of analysis, you must specify *which* parts of the coin to vary and *what the distributions/alternatives are* This is done using SA_specs, a structured list. See vignette("sensitivity") for details and examples.

You also need to specify the target of the sensitivity analysis. This should be an indicator or aggregate that can be found in one of the data sets of the coin, and is specified using the dset and iCode arguments.

Finally, if SA_type = "SA", it is advisable to set Nboot to e.g. 100 or more, which is the number of bootstrap samples to take when estimating confidence intervals on sensitivity indices. This does *not* perform extra regenerations of the coin, so setting this to a higher number shouldn't have much impact on computational time.

This function replaces the now-defunct sensitivity() from COINr < v1.0.

Value

Sensitivity analysis results as a list, containing:

- `.$Scores` a data frame with a row for each unit, and columns are the scores for each replication.
- `.$Ranks` as `.$Scores` but for unit ranks
- `.$RankStats` summary statistics for ranks of each unit
- `.$Para` a list containing parameter values for each run
- `.$Nominal` the nominal scores and ranks of each unit (i.e. from the original COIN)
- `.$Sensitivity` (only if `SA_type = "SA"`) sensitivity indices for each parameter. Also confidence intervals if `Nboot` was specified.
- Some information on the time elapsed, average time, and the parameters perturbed.
- Depending on the setting of `store_results`, may also contain a list of Methods or a list of COINs for each replication.

Examples

```
# for examples, see `vignette("sensitivity")`
# (this is because package examples are run automatically and this function can
# take a few minutes to run at realistic settings)
```

`get_stats` *Statistics of columns/indicators*

Description

Generic function for reports various statistics from a data frame or coin. See method documentation:

Usage

```
get_stats(x, ...)
```

Arguments

`x`	Object (data frame or coin)
`...`	Further arguments to be passed to methods.

Details

- `get_stats.data.frame()`
- `get_stats.coin()`

See also `vignette("analysis")`.

This function replaces the now-defunct `getStats()` from COINr < v1.0.

Value

A data frame of statistics for each column

Examples

```
# see individual method documentation
```

`get_stats.coin` *Statistics of indicators*

Description

Given a coin and a specified data set (`dset`), returns a table of statistics with entries for each column.

Usage

```
## S3 method for class 'coin'
get_stats(
  x,
  dset,
  t_skew = 2,
  t_kurt = 3.5,
  t_avail = 0.65,
  t_zero = 0.5,
  t_unq = 0.5,
  nsignif = 3,
  out2 = "df",
  ...
)
```

Arguments

`x`	A coin
`dset`	A data set present in `.$Data`
`t_skew`	Absolute skewness threshold. See details.
`t_kurt`	Kurtosis threshold. See details.
`t_avail`	Data availability threshold. See details.
`t_zero`	A threshold between 0 and 1 for flagging indicators with high proportion of zeroes. See details.
`t_unq`	A threshold between 0 and 1 for flagging indicators with low proportion of unique values. See details.plot
`nsignif`	Number of significant figures to round the output table to.
`out2`	Either `"df"` (default) to output a data frame of indicator statistics, or `"coin"` to output an updated coin with the data frame attached under `.$Analysis`.
`...`	arguments passed to or from other methods.

Details

The statistics (columns in the output table) are as follows (entries correspond to each column):

- `Min`: the minimum
- `Max`: the maximum
- `Mean`: the (arirthmetic) mean
- `Median`: the median
- `Std`: the standard deviation
- `Skew`: the skew
- `Kurt`: the kurtosis
- `N.Avail`: the number of non-NA values
- `N.NonZero`: the number of non-zero values
- `N.Unique`: the number of unique values
- `Frc.Avail`: the fraction of non-NA values
- `Frc.NonZero`: the fraction of non-zero values
- `Frc.Unique`: the fraction of unique values
- `Flag.Avail`: a data availability flag - columns with `Frc.Avail` < `t_avail` will be flagged as `"LOW"`, else `"ok"`.
- `Flag.NonZero`: a flag for columns with a high proportion of zeros. Any columns with `Frc.NonZero` < `t_zero` are flagged as `"LOW"`, otherwise `"ok"`.
- `Flag.Unique`: a unique value flag - any columns with `Frc.Unique` < `t_unq` are flagged as `"LOW"`, otherwise `"ok"`.
- `Flag.SkewKurt`: a skew and kurtosis flag which is an indication of possible outliers. Any columns with `abs(Skew)` > `t_skew` AND `Kurt` > `t_kurt` are flagged as `"OUT"`, otherwise `"ok"`.

The aim of this table, among other things, is to check the basic statistics of each column/indicator, and identify any possible issues for each indicator. For example, low data availability, having a high proportion of zeros and/or a low proportion of unique values. Further, the combination of skew and kurtosis (i.e. the `Flag.SkewKurt` column) is a simple test for possible outliers, which may require treatment using `Treat()`.

The table can be returned either to the coin or as a standalone data frame - see `out2`.

See also `vignette("analysis")`.

Value

Either a data frame or updated coin - see `out2`.

Examples

```
# build example coin
coin <-  build_example_coin(up_to = "new_coin", quietly = TRUE)

# get table of indicator statistics for raw data set
get_stats(coin, dset = "Raw", out2 = "df")
```

`get_stats.data.frame` *Statistics of columns*

Description

Takes a data frame and returns a table of statistics with entries for each column.

Usage

```
## S3 method for class 'data.frame'
get_stats(
  x,
  t_skew = 2,
  t_kurt = 3.5,
  t_avail = 0.65,
  t_zero = 0.5,
  t_unq = 0.5,
  nsignif = 3,
  ...
)
```

Arguments

`x`	A data frame with only numeric columns.
`t_skew`	Absolute skewness threshold. See details.
`t_kurt`	Kurtosis threshold. See details.
`t_avail`	Data availability threshold. See details.
`t_zero`	A threshold between 0 and 1 for flagging indicators with high proportion of zeroes. See details.
`t_unq`	A threshold between 0 and 1 for flagging indicators with low proportion of unique values. See details.
`nsignif`	Number of significant figures to round the output table to.
`...`	arguments passed to or from other methods.

Details

The statistics (columns in the output table) are as follows (entries correspond to each column):

- `Min`: the minimum
- `Max`: the maximum
- `Mean`: the (arirthmetic) mean
- `Median`: the median
- `Std`: the standard deviation
- `Skew`: the skew

- Kurt: the kurtosis
- N.Avail: the number of non-NA values
- N.NonZero: the number of non-zero values
- N.Unique: the number of unique values
- Frc.Avail: the fraction of non-NA values
- Frc.NonZero: the fraction of non-zero values
- Frc.Unique: the fraction of unique values
- Flag.Avail: a data availability flag - columns with Frc.Avail < t_avail will be flagged as "LOW", else "ok".
- Flag.NonZero: a flag for columns with a high proportion of zeros. Any columns with Frc.NonZero < t_zero are flagged as "LOW", otherwise "ok".
- Flag.Unique: a unique value flag - any columns with Frc.Unique < t_unq are flagged as "LOW", otherwise "ok".
- Flag.SkewKurt: a skew and kurtosis flag which is an indication of possible outliers. Any columns with abs(Skew) > t_skew AND Kurt > t_kurt are flagged as "OUT", otherwise "ok".

The aim of this table, among other things, is to check the basic statistics of each column/indicator, and identify any possible issues for each indicator. For example, low data availability, having a high proportion of zeros and/or a low proportion of unique values. Further, the combination of skew and kurtosis (i.e. the Flag.SkewKurt column) is a simple test for possible outliers, which may require treatment using Treat().

See also vignette("analysis").

Value

A data frame of statistics for each column

Examples

```
# stats of mtcars
get_stats(mtcars)
```

---

get_str_weak *Generate strengths and weaknesses for a specified unit*

Description

Generates a table of strengths and weaknesses for a selected unit, based on ranks, or ranks within a specified grouping variable.

Usage

```
get_str_weak(
  coin,
  dset,
  usel = NULL,
  topN = 5,
  bottomN = 5,
  withcodes = TRUE,
  use_group = NULL,
  unq_discard = NULL,
  min_discard = TRUE,
  report_level = NULL,
  with_units = TRUE,
  adjust_direction = NULL,
  sig_figs = 3
)
```

Arguments

coin	A coin
dset	The data set to extract indicator data from, to use as strengths and weaknesses.
usel	A selected unit code
topN	The top N indicators to report
bottomN	The bottom N indicators to report
withcodes	If TRUE (default), also includes a column of indicator codes. Setting to FALSE may be more useful in generating reports, where codes are not helpful.
use_group	An optional grouping variable to use for reporting in-group ranks. Specifying this will report the ranks of the selected unit within the group of use_group to which it belongs.
unq_discard	Optional parameter for handling discrete indicators. Some indicators may be binary variables of the type "yes = 1", "no = 0". These may be picked up as strengths or weaknesses, when they may not be wanted to be highlighted, since e.g. maybe half of units will have a zero or a one. This argument takes a number between 0 and 1 specifying a unique value threshold for ignoring indicators as strengths. E.g. setting prc_unq_discard = 0.2 will ensure that only indicators with at least 20% unique values will be highlighted as strengths or weaknesses. Set to NULL to disable (default).
min_discard	If TRUE (default), discards any strengths which correspond to the minimum rank for the given indicator. See details.
report_level	Aggregation level to report parent codes from. For example, setting report_level = 2 (default) will add a column to the strengths and weaknesses tables which reports the aggregation group from level 2, to which each reported indicator belongs.
with_units	If TRUE (default), includes indicator units in output tables.

adjust_direction
: If TRUE, will adjust directions of indicators according to the "Direction" column of IndMeta. By default, this is TRUE *if* dset = "Raw", and FALSE otherwise.

sig_figs
: Number of significant figures to round values to. If NULL returns values as they are.

Details

This currently only works at the indicator level. Indicators with NA values for the selected unit are ignored. Strengths and weaknesses mean the topN-ranked indicators for the selected unit. Effectively, this takes the rank that the selected unit has in each indicator, sorts the ranks, and takes the top N highest and lowest.

This function must be used with a little care: indicators should be adjusted for their directions before use, otherwise a weakness might be counted as a strength, and vice versa. Use the adjust_direction parameter to help here.

A further useful parameter is unq_discard, which also filters out any indicators with a low number of unique values, based on a specified threshold. Also min_discard which filters out any indicators which have the minimum rank.

The best way to use this function is to play around with the settings a little bit. The reason being that in practice, indicators have very different distributions and these can sometimes lead to unexpected outcomes. An example is if you have an indicator with 50% zero values, and the rest non-zero (but unique). Using the sport ranking system, all units with zero values will receive a rank which is equal to the number of units divided by two. This then might be counted as a "strength" for some units with overall low scores. But a zero value can hardly be called a strength. This is where the min_discard function can help out.

Problems such as these mainly arise when e.g. generating a large number of country profiles.

This function replaces the now-defunct getStrengthNWeak() from COINr < v1.0.

Value

A list containing a data frame .$Strengths, and a data frame .$Weaknesses. Each data frame has columns with indicator code, name, rank and value (for the selected unit).

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# get strengths and weaknesses for ESP
get_str_weak(coin, dset = "Raw", usel = "ESP")
```

get_trends *Get time trends*

Description

Get time trends from a purse object. This function extracts a panel data set from a purse, and calculates trends for each indicator/unit pair using a specified function `f_trend`. For example, if `f_trend = "CAGR"`, this extracts the time series for each indicator/unit pair and passes it to `CAGR()`.

Usage

```
get_trends(
  purse,
  dset,
  uCodes = NULL,
  iCodes = NULL,
  Time = NULL,
  use_latest = NULL,
  f_trend = "CAGR",
  interp_at = NULL,
  adjust_directions = FALSE
)
```

Arguments

`purse`	A purse object
`dset`	Name of the data set to extract, passed to `get_data.purse()`
`uCodes`	Optional subset of unit codes to extract, passed to `get_data.purse()`
`iCodes`	Optional subset of indicator/aggregate codes to extract, passed to `get_data.purse()`
`Time`	Optional vector of time points to extract, passed to `get_data.purse()`
`use_latest`	A positive integer which specifies to use only the latest "n" data points. If this is specified, it overrides `Time`. If e.g. `use_latest = 5`, will use the latest five observations, working backwards from the latest non-`NA` point.
`f_trend`	Function that returns a metric describing the trend of the time series. See details.
`interp_at`	Option to linearly interpolate missing data points in each time series. Must be specified as a vector of time values where to apply interpolation. If `interp_at = "all"`, will attempt to interpolate at every time point. Uses linear interpolation - note that any `NA`s outside of the range of observed values will not be estimated, i.e. this does not *extrapolate* beyond the range of data. See `approx_df()`.
`adjust_directions`	Logical: if `TRUE`, trend metrics are adjusted according to indicator/aggregate directions input in `iMeta` (i.e. if the corresponding direction is -1, the metric will be multiplied by -1).

Details

This function requires a purse object as an input. The data set is selected using `get_data()`, such that a subset of the data set can be analysed using the `uCodes`, `iCodes` and `Time` arguments. The latter is useful especially if only a subset of the time series should be analysed.

The function `f_trend` is a function that, given a time series, returns a trend metric. This must follow a specific format. It must of course be available to call, and *must* have arguments `y` and `x`, which are respectively a vector of values and a vector indexing the values in time. See `prc_change()` and `CAGR()` for examples. The function *must* return a single value (not a vector with multiple entries, or a list). The function can return either numeric or character values.

Value

A data frame in long format, with trend metrics for each indicator/unit pair, plus data availability statistics.

Examples

```
#
```

`get_unit_summary` *Generate unit summary table*

Description

Generates a summary table for a single unit. This is mostly useful in unit reports.

Usage

```
get_unit_summary(coin, usel, Levels, dset = "Aggregated", nround = 2)
```

Arguments

`coin`	A coin
`usel`	A selected unit code
`Levels`	The aggregation levels to display results from.
`dset`	The data set within the coin to extract scores and ranks from
`nround`	Number of decimals to round scores to, default 2. Set to `NULL` to disable rounding.

Details

This returns the scores and ranks for each indicator/aggregate as specified in `aglevs`. It orders the table so that the highest aggregation levels are first. This means that if the index level is included, it will be first.

This function replaces the now-defunct `getUnitSummary()` from COINr < v1.0.

Value

A summary table as a data frame, containing scores and ranks for specified indicators/aggregates.

Examples

```
# build full example coin
coin <- build_example_coin(quietly = TRUE)

# summary of scores for IND at levels 4, 3 and 2
get_unit_summary(coin, usel = "IND", Levels = c(4,3,2), dset = "Aggregated")
```

---

icodes_to_inames *Convert iCodes to iNames*

Description

Convert iCodes to iNames

Usage

```
icodes_to_inames(coin, iCodes)
```

Arguments

coin	A coin
iCodes	A vector of iCodes

Value

Vector of iNames

---

import_coin_tool *Import data directly from COIN Tool*

Description

The COIN Tool is an Excel-based tool for building composite indicators. This function provides a direct interface for reading a COIN Tool input deck and converting it to COINr. You need to provide a COIN Tool file, with the "Database" sheet properly compiled.

Usage

```
import_coin_tool(fname, makecodes = FALSE, oldtool = FALSE, out2 = "list")
```

Arguments

fname	The file name and path to read, e.g. "C:/Documents/COINToolFile.xlsx".
makecodes	Logical: if TRUE, will generate short indicator codes based on indicator names, otherwise if FALSE, will use COIN Tool indicator codes "Ind.01", etc. Currently only does this for indicators, not aggregation groups.
oldtool	Logical: if TRUE, compatible with old COIN Tool (pre-release, early 2019 or earlier). There are some minor differences on where the elements are found.
out2	Either "list" (default) to output a list with iData and iMeta entries (for input into new_coin()), else "coin" to output a coin.

Details

This function replaces the now-defunct COINToolIn() from COINr < v1.0.

Value

Either a list or a coin, depending on out2

Examples

```
## Not run:
## This example downloads a COIN Tool spreadsheet containing example data,
## saves it to a temporary directory, unzips, and reads into R. Finally it
## assembles it into a COIN.

# Make temp zip filename in temporary directory
tmpz <- tempfile(fileext = ".zip")
# Download an example COIN Tool file to temporary directory
# NOTE: the download.file() command may need its "method" option set to a
# specific value depending on the platform you run this on. You can also
# choose to download/unzip this file manually.
download.file("https://knowledge4policy.ec.europa.eu/sites/default/
files/coin_tool_v1_lite_exampledata.zip", tmpz)
# Unzip
CTpath <- unzip(tmpz, exdir = tempdir())
# Read COIN Tool into R
l <- import_coin_tool(CTpath, makecodes = TRUE)
## End(Not run)
```

Impute *Imputation of missing data*

Description

This is a generic function with the following methods:

Usage

```
Impute(x, ...)
```

Arguments

x	Object to be imputed
...	arguments passed to or from other methods.

Details

- Impute.numeric()
- Impute.data.frame()
- Impute.coin()
- Impute.purse()

See those methods for individual documentation.

This function replaces the now-defunct impute() from COINr < v1.0.

Value

An object of the same class as x, but imputed.

Examples

```
# See individual method documentation
```

Impute.coin *Impute a data set in a coin*

Description

This imputes any NAs in the data set specified by dset by invoking the function f_i and any optional arguments f_i_para on each column at a time (if impute_by = "column"), or on each row at a time (if impute_by = "row"), or by passing the entire data frame to f_i if impute_by = "df".

Usage

```
## S3 method for class 'coin'
Impute(
  x,
  dset,
  f_i = NULL,
  f_i_para = NULL,
  impute_by = "column",
  use_group = NULL,
  group_level = NULL,
```

```
  normalise_first = NULL,
  out2 = "coin",
  write_to = NULL,
  disable = FALSE,
  warn_on_NAs = TRUE,
  ...
)
```

Arguments

x	A coin class object
dset	The name of the data set to apply the function to, which should be accessible in .$Data.
f_i	An imputation function. See details.
f_i_para	Further arguments to pass to f_i, other than x. See details.
impute_by	Specifies how to impute: if "column", passes each column (indicator) separately as a numerical vector to f_i; if "row", passes each *row* separately; and if "df" passes the entire data set (data frame) to f_i. The function called by f_i should be compatible with the type of data passed to it.
use_group	Optional grouping variable name to pass to imputation function if this supports group imputation.
group_level	A level of the framework to use for grouping indicators. This is only relevant if impute_by = "row" or "df". In that case, indicators will be split into their groups at the level specified by group_level, and imputation will be performed across rows of the group, rather than the whole data set. This can make more sense because indicators within a group are likely to be more similar.
normalise_first	Logical: if TRUE, each column is normalised using a min-max operation before imputation. By default this is FALSE unless impute_by = "row". See details.
out2	Either "coin" to return normalised data set back to the coin, or df to simply return a data frame.
write_to	Optional character string for naming the data set in the coin. Data will be written to .$Data[[write_to]]. Default is write_to == "Imputed".
disable	Logical: if TRUE will disable imputation completely and write the unaltered data set. This option is mainly useful in sensitivity and uncertainty analysis (to test the effect of turning imputation on/off).
warn_on_NAs	Logical: if TRUE will issue a warning if there are any NAs detected in the data frame after imputation has been applied. Set FALSE to suppress these warnings.
...	arguments passed to or from other methods.

Details

Clearly, the function f_i needs to be able to accept with the data class passed to it - if impute_by is "row" or "column" this will be a numeric vector, or if "df" it will be a data frame. Moreover, this function should return a vector or data frame identical to the vector/data frame passed to it except for NA values, which can be replaced. The function f_i is not required to replace *all* NA values.

COINr has several built-in imputation functions of the form `i_*()` for vectors which can be called by `Impute()`. See the online documentation for more details.

When imputing row-wise, prior normalisation of the data is recommended. This is because imputation will use e.g. the mean of the unit values over all indicators (columns). If the indicators are on very different scales, the result will likely make no sense. If the indicators are normalised first, more sensible results can be obtained. There are two options to pre-normalise: first is by setting `normalise_first = TRUE` - this is anyway the default if `impute_by = "row"`. In this case, you also need to supply a vector of directions. The data will then be normalised using a min-max approach before imputation, followed by the inverse operation to return the data to the original scales.

Another approach which gives more control is to simply run `Normalise()` first, and work with the normalised data from that point onwards. In that case it is better to set `normalise_first = FALSE`, since by default if `impute_by = "row"` it will be set to `TRUE`.

Checks are made on the format of the data returned by imputation functions, to ensure the type and that non-`NA` values have not been inadvertently altered. This latter check is allowed a degree of tolerance for numerical precision, controlled by the `sfigs` argument. This is because if the data frame is normalised, and/or depending on the imputation function, there may be a very small differences. By default `sfigs = 9`, meaning that the non-`NA` values pre and post-imputation are compared to 9 significant figures.

See also documentation for `Impute.data.frame()` and `Impute.numeric()` which are called by this function.

Value

An updated coin with imputed data set at `.$Data[[write_to]]`

Examples

```
#' # build coin
coin <- build_example_coin(up_to = "new_coin")

# impute raw data set using population groups
# output to data frame directly
Impute(coin, dset = "Raw", f_i = "i_mean_grp",
               use_group = "Pop_group", out2 = "df")
```

Impute.data.frame *Impute a data frame*

Description

Impute a data frame using any function, either column-wise, row-wise or by the whole data frame in one shot.

Usage

```
## S3 method for class 'data.frame'
Impute(
  x,
  f_i = NULL,
  f_i_para = NULL,
  impute_by = "column",
  normalise_first = NULL,
  directions = NULL,
  warn_on_NAs = TRUE,
  ...
)
```

Arguments

x	A data frame with only numeric columns.
f_i	A function to use for imputation. By default, imputation is performed by simply substituting the mean of non-NA values for each column at a time.
f_i_para	Any additional parameters to pass to f_i, apart from x
impute_by	Specifies how to impute: if "column", passes each column separately as a numerical vector to f_i; if "row", passes each *row* separately; and if "df" passes the entire data frame to f_i. The function called by f_i should be compatible with the type of data passed to it.
normalise_first	Logical: if TRUE, each column is normalised using a min-max operation before imputation. By default this is FALSE unless impute_by = "row". See details.
directions	A vector of directions: either -1 or 1 to indicate the direction of each column of x - this is only used if normalise_first = TRUE. See details.
warn_on_NAs	Logical: if TRUE will issue a warning if there are any NAs detected in the data frame after imputation has been applied. Set FALSE to suppress these warnings.
...	arguments passed to or from other methods.

Details

This function only accepts data frames with all numeric columns. It imputes any NAs in the data frame by invoking the function f_i and any optional arguments f_i_para on each column at a time (if impute_by = "column"), or on each row at a time (if impute_by = "row"), or by passing the entire data frame to f_i if impute_by = "df".

Clearly, the function f_i needs to be able to accept with the data class passed to it - if impute_by is "row" or "column" this will be a numeric vector, or if "df" it will be a data frame. Moreover, this function should return a vector or data frame identical to the vector/data frame passed to it except for NA values, which can be replaced. The function f_i is not required to replace *all* NA values.

COINr has several built-in imputation functions of the form i_*() for vectors which can be called by Impute(). See the online documentation for more details.

When imputing row-wise, prior normalisation of the data is recommended. This is because imputation will use e.g. the mean of the unit values over all indicators (columns). If the indicators are

on very different scales, the result will likely make no sense. If the indicators are normalised first, more sensible results can be obtained. There are two options to pre-normalise: first is by setting `normalise_first = TRUE` - this is anyway the default if `impute_by = "row"`. In this case, you also need to supply a vector of directions. The data will then be normalised using a min-max approach before imputation, followed by the inverse operation to return the data to the original scales.

Another approach which gives more control is to simply run `Normalise()` first, and work with the normalised data from that point onwards. In that case it is better to set `normalise_first = FALSE`, since by default if `impute_by = "row"` it will be set to `TRUE`.

Checks are made on the format of the data returned by imputation functions, to ensure the type and that non-`NA` values have not been inadvertently altered. This latter check is allowed a degree of tolerance for numerical precision, controlled by the `sfigs` argument. This is because if the data frame is normalised, and/or depending on the imputation function, there may be a very small differences. By default `sfigs = 9`, meaning that the non-`NA` values pre and post-imputation are compared to 9 significant figures.

Value

An imputed data frame

Examples

```
# a df of random numbers
X <- as.data.frame(matrix(runif(50), 10, 5))

# introduce NAs (2 in 3 of 5 cols)
X[sample(1:10, 2), 1] <- NA
X[sample(1:10, 2), 3] <- NA
X[sample(1:10, 2), 5] <- NA

# impute using column mean
Impute(X, f_i = "i_mean")

# impute using row median (no normalisation)
Impute(X, f_i = "i_median", impute_by = "row",
       normalise_first = FALSE)
```

`Impute.numeric` *Impute a numeric vector*

Description

Imputes missing values in a numeric vector using a function `f_i`. This function should return a vector identical to `x` except for `NA` values, which can be replaced. The function `f_i` is not required to replace *all* `NA` values.

Usage

```
## S3 method for class 'numeric'
Impute(x, f_i = NULL, f_i_para = NULL, ...)
```

Arguments

x	A numeric vector, possibly with NA values to be imputed.
f_i	A function that imputes missing values in a numeric vector. See description and details.
f_i_para	Optional further arguments to be passed to f_i()
...	arguments passed to or from other methods.

Details

This calls the function f_i(), with optionally further arguments f_i_para, to impute any missing values found in x. By default, f_i = "i_mean()", which simply imputes NAs with the mean of the non-NA values in x.

COINr has several built-in imputation functions of the form i_*() for vectors which can be called by Impute(). See the online documentation for more details.

You could also use one of the imputation functions directly (such as i_mean()). However, this function offers a few extra advantages, such as checking the input and output formats, and making sure the resulting imputed vector agrees with the input. It will also skip imputation entirely if there are no NAs at all.

Value

An imputed numeric vector of the same length of x.

Examples

```
# a vector with a missing value
x <- 1:10
x[3] <- NA
x

# impute using median
# this calls COINr's i_median() function
Impute(x, f_i = "i_median")
```

`Impute.purse` *Impute data sets in a purse*

Description

This function imputes the target data set `dset` in each coin using the imputation function `f_i`. This is performed in the same way as the coin method `Impute.coin()`, but with one "special case" for panel data. If `f_i = "impute_panel`, the data sets inside the purse are imputed using the `impute_panel()` function. In this case, coins are not imputed individually, but treated as a single data set. In this case, optionally set the imputation method as `f_i_para = list(imp_type = .)` and `f_i_para = list(max_time = .)` where `.` should be substituted with the maximum number of time points to search backwards for a non-NA value. See `impute_panel()` for more details. No further arguments need to be passed to `impute_panel()`. See `vignette("imputation")` for more details. See also `Impute.coin()` documentation.

Usage

```
## S3 method for class 'purse'
Impute(
  x,
  dset,
  f_i = NULL,
  f_i_para = NULL,
  impute_by = "column",
  group_level = NULL,
  use_group = NULL,
  normalise_first = NULL,
  write_to = NULL,
  warn_on_NAs = TRUE,
  ...
)
```

Arguments

`x`	A purse object
`dset`	The name of the data set to apply the function to, which should be accessible in `.$Data`.
`f_i`	An imputation function. For the "purse" class, if `f_i = "impute_panel` this is a special case: see details.
`f_i_para`	Further arguments to pass to `f_i`, other than `x`. See details.
`impute_by`	Specifies how to impute: if `"column"`, passes each column (indicator) separately as a numerical vector to `f_i`; if `"row"`, passes each *row* separately; and if `"df"` passes the entire data set (data frame) to `f_i`. The function called by `f_i` should be compatible with the type of data passed to it.

group_level — A level of the framework to use for grouping indicators. This is only relevant if impute_by = "row" or "df". In that case, indicators will be split into their groups at the level specified by group_level, and imputation will be performed across rows of the group, rather than the whole data set. This can make more sense because indicators within a group are likely to be more similar.

use_group — Optional grouping variable name to pass to imputation function if this supports group imputation.

normalise_first — Logical: if TRUE, each column is normalised using a min-max operation before imputation. By default this is FALSE unless impute_by = "row". See details.

write_to — Optional character string for naming the resulting data set in each coin. Data will be written to .$Data[[write_to]]. Default is write_to == "Imputed".

warn_on_NAs — Logical: if TRUE will issue a warning if there are any NAs detected in the data frame after imputation has been applied. Set FALSE to suppress these warnings.

... — arguments passed to or from other methods.

Value

An updated purse with imputed data sets added to each coin.

Examples

```
# see vignette("imputation")
```

---

impute_panel *Impute panel data*

Description

Given a data frame of panel data, with a time-index column time_col and a unit ID column unit_col, imputes other columns using the entry from the latest available time point.

Usage

```
impute_panel(
  iData,
  time_col = NULL,
  unit_col = NULL,
  cols = NULL,
  imp_type = NULL,
  max_time = NULL
)
```

Arguments

iData	A data frame of indicator data, containing a time index column time_col, a unit code column unit_col, and other numerical columns to be imputed.
time_col	The name of a column found in iData to be used as the time index column. Must point to a numeric column.
unit_col	The name of a column found in iData to be used as the unit code/ID column. Must point to a character column.
cols	Optionally, a character vector of names of columns to impute. If NULL (default), all columns apart from time_col and unit_col will be imputed where possible.
imp_type	One of "latest" "constant" or "linear". In the first case, missing points are imputed with the last non-NA observation for each time series, up to max_time. For "constant" or "linear", missing points are imputed using stats::approx(), passing "constant" or "linear" to the method argument, and points outside of the range of observed values are replaced with the nearest non-NA point. This is equivalent to rule = 2 in stats::approx() for each time series. The difference between "latest" and "constant" is that the latter allows control over the maximum number of time points to impute backwards (using max_time) whereas the former doesn't. Additionally, "constant" will impute outside of the observed range of values at the beginning of the time series, whereas "latest" won't.
max_time	The maximum number of time points to look backwards to impute from. E.g. if max_time = 1, if an NA is found at time t, it will only look for a replacement value at $t-1$ but not in any time points before that. By default, searches all time points available.

Details

This presumes that there are multiple observations for each unit code, i.e. one per time point. It then searches for any missing values in the target year, and replaces them with the equivalent points from previous time points. It will replace using the most recently available point or using linear interpolation: see imp_type argument.

Value

A list containing:

- .$iData_imp: An iData format data frame with missing data imputed using previous time points (where possible).
- .$DataT: A data frame in the same format as iData, where each entry shows which time point each data point came from.

Examples

```
# Copy example panel data
iData_p <- ASEM_iData_p

# we introduce two NAs: one for NZ in 2022 in LPI indicator
iData_p$LPI[iData_p$uCode == "NZ" & iData_p$Time == 2022] <-  NA
```

```
# one for AT, also in 2022, but for Flights indicator
iData_p$Flights[iData_p$uCode == "AT" & iData_p$Time == 2022] <- NA

# impute: target only the two columns where NAs introduced
l_imp <- impute_panel(iData_p, cols = c("LPI", "Flights"))
# get imputed df
iData_imp <- l_imp$iData_imp

# check the output is what we expect: both NAs introduced should now have 2021 values
iData_imp$LPI[iData_imp$uCode == "NZ" & iData_imp$Time == 2022] ==
  ASEM_iData_p$LPI[ASEM_iData_p$uCode == "NZ" & ASEM_iData_p$Time == 2021]

iData_imp$Flights[iData_imp$uCode == "AT" & iData_imp$Time == 2022] ==
  ASEM_iData_p$Flights[ASEM_iData_p$uCode == "AT" & ASEM_iData_p$Time == 2021]
```

`is.coin` *Check if object is coin class*

Description

Check if object is coin class

Usage

```
is.coin(x)
```

Arguments

`x`	An object to be checked.

Value

Logical

`is.purse` *Check if object is purse class*

Description

Check if object is purse class

Usage

```
is.purse(x)
```

Arguments

x	An object to be checked.

Value

Logical

---

i_mean *Impute by mean*

Description

Replaces NAs in a numeric vector with the mean of the non-NA values.

Usage

```
i_mean(x)
```

Arguments

x	A numeric vector

Value

A numeric vector

Examples

```
x <- c(1,2,3,4, NA)
i_mean(x)
```

---

i_mean_grp *Impute by group mean*

Description

Replaces NAs in a numeric vector with the grouped arithmetic means of the non-NA values. Groups are defined by the f argument.

Usage

```
i_mean_grp(x, f, skip_f_na = TRUE)
```

Arguments

x	A numeric vector
f	A grouping variable, of the same length of x, that specifies the group that each value of x belongs to. This will be coerced to a factor.
skip_f_na	If TRUE, will work around any NAs in f (the corresponding values of x will be excluded from the imputation and returned unaltered). Else if FALSE, will cause an error.

Value

A numeric vector

Examples

```
x <- c(NA, runif(10), NA)
f <- c(rep("a", 6), rep("b", 6))
i_mean_grp(x, f)
```

---

i_median *Impute by median*

Description

Replaces NAs in a numeric vector with the median of the non-NA values.

Usage

```
i_median(x)
```

Arguments

x	A numeric vector

Value

A numeric vector

Examples

```
x <- c(1,2,3,4, NA)
i_median(x)
```

i_median_grp *Impute by group median*

Description

Replaces NAs in a numeric vector with the grouped medians of the non-NA values. Groups are defined by the f argument.

Usage

```
i_median_grp(x, f, skip_f_na = TRUE)
```

Arguments

x	A numeric vector
f	A grouping variable, of the same length of x, that specifies the group that each value of x belongs to. This will be coerced to a factor.
skip_f_na	If TRUE, will work around any NAs in f (the corresponding values of x will be excluded from the imputation and returned unaltered). Else if FALSE, will cause an error.

Value

A numeric vector

Examples

```
x <- c(NA, runif(10), NA)
f <- c(rep("a", 6), rep("b", 6))
i_median_grp(x, f)
```

kurt *Calculate kurtosis*

Description

Calculates kurtosis of the values of a numeric vector. This uses the same definition of kurtosis as as the "kurtosis()" function in the e1071 package, where type == 2, which is equivalent to the definition of kurtosis used in Excel.

Usage

```
kurt(x, na.rm = FALSE)
```

Arguments

x	A numeric vector.
na.rm	Set TRUE to remove NA values, otherwise returns NA.

Value

A kurtosis value (scalar).

Examples

```
x <- runif(20)
kurt(x)
```

---

log_CT *Log-transform a vector*

Description

Performs a log transform on a numeric vector.

Usage

```
log_CT(x, na.rm = FALSE)
```

Arguments

x	A numeric vector.
na.rm	Set TRUE to remove NA values, otherwise returns NA.

Details

Specifically, this performs a modified "COIN Tool log" transform: `log(x-min(x) + a)`, where `a <- 0.01*(max(x)-min(x))`.

Value

A log-transformed vector of data, and treatment details wrapped in a list.

Examples

```
x <- runif(20)
log_CT(x)
```

log_CT_orig *Log-transform a vector*

Description

Performs a log transform on a numeric vector.

Usage

```
log_CT_orig(x, na.rm = FALSE)
```

Arguments

x	A numeric vector.
na.rm	Set TRUE to remove NA values, otherwise returns NA.

Details

Specifically, this performs a "COIN Tool log" transform: `log(x-min(x) + 1)`.

Value

A log-transformed vector of data, and treatment details wrapped in a list.

Examples

```
x <- runif(20)
log_CT_orig(x)
```

log_CT_plus *Log transform a vector (skew corrected)*

Description

Performs a log transform on a numeric vector, but with consideration for the direction of the skew. The aim here is to reduce the absolute value of skew, regardless of its direction.

Usage

```
log_CT_plus(x, na.rm = FALSE)
```

Arguments

x	A numeric vector
na.rm	Set TRUE to remove NA values, otherwise returns NA.

Details

Specifically:

If the skew of x is positive, this performs a modified "COIN Tool log" transform: `log(x-min(x) + a)`, where `a <- 0.01*(max(x)-min(x))`.

If the skew of x is negative, it performs an equivalent transformation `-log(xmax + a - x)`.

Value

A log-transformed vector of data, and treatment details wrapped in a list.

Examples

```
x <- runif(20)
log_CT(x)
```

---

log_GII *Log-transform a vector*

Description

Performs a log transform on a numeric vector. This function is currently not recommended - see comments below.

Usage

```
log_GII(x, na.rm = FALSE)
```

Arguments

x	A numeric vector.
na.rm	Set TRUE to remove NA values, otherwise returns NA.

Details

Specifically, this performs a "GII log" transform, which is what was encoded in the GII2020 spreadsheet.

Note that this transformation is currently NOT recommended because it seems quite volatile and can flip the direction of the indicator. If the maximum value of the indicator is less than one, this reverses the direction.

Value

A log-transformed vector of data.

Examples

```
x <- runif(20)
log_GII(x)
```

`names_to_codes` *Generate short codes from long names*

Description

Given a character vector of long names (probably with spaces), generates short codes. Intended for use when importing from the COIN Tool.

Usage

```
names_to_codes(cvec, maxword = 2, maxlet = 4)
```

Arguments

`cvec`	A character vector of names
`maxword`	The maximum number of words to use in building a short name (default 2)
`maxlet`	The number of letters to take from each word (default 4)

Details

This function replaces the now-defunct `names2Codes()` from COINr < v1.0.

Value

A corresponding character vector, but with short codes, and no duplicates.

See Also

- `import_coin_tool()` Import data from the COIN Tool (Excel).

Examples

```
# get names from example data
iNames <- ASEM_iMeta$iName

# convert to codes
names_to_codes(iNames)
```

new_coin *Create a new coin*

Description

Creates a new "coin" class object, or a "purse" class object (time-indexed collection of coins). A purse class object is created if panel data is supplied. Coins and purses are the main object classes used in COINr, although a number of functions also support other classes such as data frames and vectors.

Usage

```
new_coin(
  iData,
  iMeta,
  exclude = NULL,
  split_to = NULL,
  level_names = NULL,
  retain_all_uCodes_on_split = FALSE,
  quietly = FALSE
)
```

Arguments

iData	The indicator data and metadata of each unit
iMeta	Indicator metadata
exclude	Optional character vector of any indicator codes (iCodes) to exclude from the coin(s).
split_to	This is used to split panel data into multiple coins, a so-called "purse". Should be either "all", or a subset of entries in iData$Time. See Details.
level_names	Optional character vector of names of levels. Must have length equal to the number of levels in the hierarchy (max(iMeta$Level, na.rm = TRUE)).
retain_all_uCodes_on_split	Logical: if panel data is input and split to a purse using split_to, this controls how units with no data at certain time points are handled. If set FALSE, then unit at time t with no data in any indicators will be removed completely from the coin for that time point. If TRUE, all units will be included in every time point. The latter option may be useful if you impute over time.
quietly	If TRUE, suppresses all messages

Details

A coin object is fundamentally created by passing two data frames to new_coin(): iData which specifies the data points for each unit and indicator, as well as other optional variables; and iMeta which specifies details about each indicator/variable found in iData, including its type, name, position in the index, units, and other properties.

These data frames need to follow fairly strict requirements regarding their format and consistency. Run `check_iData()` and `check_iMeta()` to validate your data frames, and these should generate helpful error messages when things go wrong.

It is worth reading a little about coins and purses to use COINr. See `vignette("coins")` for more details.

`iData`**:**

`iData` should be a data frame with required column `uCode` which gives the code assigned to each unit (alphanumeric, not starting with a number). All other columns are defined by corresponding entries in `iMeta`, with the following special exceptions:

- `Time` is an optional column which allows panel data to be input, consisting of e.g. multiple rows for each `uCode`: one for each `Time` value. This can be used to split a set of panel data into multiple coins (a so-called "purse") which can be input to COINr functions.
- `uName` is an optional column which specifies a longer name for each unit. If this column is not included, unit codes (`uCode`) will be used as unit names where required.

`iMeta`**:**

Required columns for `iMeta` are:

- `Level`: Level in aggregation, where 1 is indicator level, 2 is the level resulting from aggregating indicators, 3 is the result of aggregating level 2, and so on. Set to `NA` for entries that are not included in the index (groups, denominators, etc).
- `iCode`: Indicator code, alphanumeric. Must not start with a number.
- `Parent`: Group (`iCode`) to which indicator/aggregate belongs in level immediately above. Each entry here should also be found in `iCode`. Set to `NA` only for the highest (Index) level (no parent), or for entries that are not included in the index (groups, denominators, etc).
- `Direction`: Numeric, either -1 or 1
- `Weight`: Numeric weight, will be rescaled to sum to 1 within aggregation group. Set to `NA` for entries that are not included in the index (groups, denominators, etc).
- `Type`: The type, corresponding to `iCode`. Can be either `Indicator`, `Aggregate`, `Group`, `Denominator`, or `Other`.

Optional columns that are recognised in certain functions are:

- `iName`: Name of the indicator: a longer name which is used in some plotting functions.
- `Unit`: the unit of the indicator, e.g. USD, thousands, score, etc. Used in some plots if available.
- `Target`: a target for the indicator. Used if normalisation type is distance-to-target.

The `iMeta` data frame essentially gives details about each of the columns found in `iData`, as well as details about additional data columns eventually created by aggregating indicators. This means that the entries in `iMeta` must include *all* columns in `iData`, *except* the three special column names: `uCode`, `uName`, and `Time`. In other words, all column names of `iData` should appear in `iMeta$iCode`, except the three special cases mentioned. The `iName` column optionally can be used to give longer names to each indicator which can be used for display in plots.

`iMeta` also specifies the structure of the index, by specifying the parent of each indicator and aggregate. The `Parent` column must refer to entries that can be found in `iCode`. Try `View(ASEM_iMeta)` for an example of how this works.

`Level` is the "vertical" level in the hierarchy, where 1 is the bottom level (indicators), and each successive level is created by aggregating the level below according to its specified groups.

`Direction` is set to 1 if higher values of the indicator should result in higher values of the index, and -1 in the opposite case.

The `Type` column specifies the type of the entry: `Indicator` should be used for indicators at level 1. `Aggregate` for aggregates created by aggregating indicators or other aggregates. Otherwise set to `Group` if the variable is not used for building the index but instead is for defining groups of units. Set to `Denominator` if the variable is to be used for scaling (denominating) other indicators. Finally, set to `Other` if the variable should be ignored but passed through. Any other entries here will cause an error.

Note: this function requires the columns above as specified, but extra columns can also be added without causing errors.

Other arguments:

The `exclude` argument can be used to exclude specified indicators. If this is specified, `.$Data$Raw` will be built excluding these indicators, as will all subsequent build operations. However the full data set will still be stored in `.$Log$new_coin`. The codes here should correspond to entries in the `iMeta$iCode`. This option is useful e.g. in generating alternative coins with different indicator sets, and can be included as a variable in a sensitivity analysis.

The `split_to` argument allows panel data to be used. Panel data must have a `Time` column in `iData`, which consists of some numerical time variable, such as a year. Panel data has multiple observations for each `uCode`, one for each unique entry in `Time`. The `Time` column is required to be numerical, because it needs to be possible to order it. To split panel data, specify `split_to = "all"` to split to a single coin for each of the unique entries in `Time`. Alternatively, you can pass a vector of entries in `Time` which allows to split to a subset of the entries to `Time`.

Splitting panel data results in a so-called "purse" class, which is a data frame of COINs, indexed by `Time`. See `vignette("coins")` for more details.

This function replaces the now-defunct `assemble()` from COINr < v1.0.

Value

A "coin" object or a "purse" object.

Examples

```
# build a coin using example data frames
ASEM_coin <- new_coin(iData = ASEM_iData,
                      iMeta = ASEM_iMeta,
                      level_names = c("Indicator", "Pillar", "Sub-index", "Index"))
# view coin contents
ASEM_coin

# build example purse class
ASEM_purse <- new_coin(iData = ASEM_iData_p,
                       iMeta = ASEM_iMeta,
                       split_to = "all",
                       quietly = TRUE)
# view purse contents
ASEM_purse

# see vignette("coins") for further info
```

Normalise *Normalise data*

Description

This is a generic function for normalising variables and indicators, i.e. bringing them onto a common scale. Please see individual method documentation depending on your data class:

Usage

```
Normalise(x, ...)
```

Arguments

x	Object to be normalised
...	Further arguments to be passed to methods.

Details

- Normalise.numeric()
- Normalise.data.frame()
- Normalise.coin()
- Normalise.purse()

See also vignette("normalise") for more details.

This function replaces the now-defunct normalise() from COINr < v1.0.

Examples

```
# See individual method documentation.
```

Normalise.coin *Create a normalised data set*

Description

Creates a normalised data set using specifications specified in global_specs. Columns of dset can also optionally be normalised with individual specifications using the indiv_specs argument. If indicators should have their directions reversed, this can be specified using the directions argument. Non-numeric columns are ignored automatically by this function. By default, this function normalises each indicator using the "min-max" method, scaling indicators to lie between 0 and 100. This calls the n_minmax() function. COINr has a number of built-in normalisation functions of the form n_*(). See online documentation for details.

Usage

```
## S3 method for class 'coin'
Normalise(
  x,
  dset,
  global_specs = NULL,
  indiv_specs = NULL,
  directions = NULL,
  out2 = "coin",
  write_to = NULL,
  write2log = TRUE,
  ...
)
```

Arguments

x	A coin
dset	A named data set found in .$Data
global_specs	Specifications to apply to all columns, apart from those specified by indiv_specs. See details.
indiv_specs	Specifications applied to specific columns, overriding those specified in global_specs. See details.
directions	An optional data frame containing the following columns: • iCode The indicator code, corresponding to the column names of the data set • Direction numeric vector with entries either -1 or 1 If directions is not specified, the directions will be taken from the iMeta table in the coin, if available.
out2	Either "coin" to return normalised data set back to the coin, or df to simply return a data frame.
write_to	Optional character string for naming the data set in the coin. Data will be written to .$Data[[write_to]]. Default is write_to == "Normalised".
write2log	Logical: if FALSE, the arguments of this function are not written to the coin log, so this function will not be invoked when regenerating. Recommend to keep TRUE unless you have a good reason to do otherwise.
...	arguments passed to or from other methods.

Details

Global specification:

The global_specs argument is a list which specifies the normalisation function and any function parameters that should be used to normalise the indicators found in the data set. Unless indiv_specs is specified, this will be applied to all indicators. The list should have two entries:

- .$f_n: the name of the function to use to normalise each indicator

- `.$f_n_para`: any further parameters to pass to `f_n`, apart from the numeric vector (each column of the data set)

In this list, `f_n` should be a character string which is the name of a normalisation function. For example, `f_n = "n_minmax"` calls the `n_minmax()` function. `f_n_para` is a list of any further arguments to `f_n`. This means that any function can be passed to `Normalise()`, as long as its first argument is `x`, a numeric vector, and it returns a numeric vector of the same length. See `n_minmax()` for an example.

`f_n_para` is *required* to be a named list. So e.g. if we define a function `f1(x, arg1, arg2)` then we should specify `f_n = "f1"`, and `f_n_para = list(arg1 = val1, arg2 = val2)`, where `val1` and `val2` are the values assigned to the arguments `arg1` and `arg2` respectively.

The default list for `global_specs` is: `list(f_n = "n_minmax", f_n_para = list(l_u = c(0,100)))`, i.e. min-max normalisation between 0 and 100.

Note, all COINr normalisation functions (passed to `f_n`) are of the form `n_*()`. Type `n_` in the R Studio console and press the Tab key to see a list.

Individual parameter specification with iMeta:

For some normalisation methods we may use the same function for all indicators but use different parameters - for example, using distance to target normalisation or goalpost normalisation. COINr now supports specifying these parameters in the `iMeta` table. To enable this, set `f_n_para = "use_iMeta"` within the `global_specs` list.

For this to work you will also need to add the correct-named columns in the `iMeta` table. To see which column names to add, check the function documentation of the normalisation function you wish to use (e.g. `n_goalposts()`). See also examples in the normalisation vignette. These columns should be added before construction of the coin.

Individual column specification:

To give full individual control, indicators can be normalised with different normalisation functions and parameters using the `indiv_specs` argument. This must be specified as a named list e.g. `list(i1 = specs1, i2 = specs2)` where `i1` and `i2` are `iCodes` to apply individual normalisation to, and `specs1` and `specs2` are respectively lists of the same format as `global_specs` (see above). In other words, `indiv_specs` is a big list wrapping together `global_specs`-style lists. Any `iCodes` not named in `indiv_specs` (i.e. those not in `names(indiv_specs)`) are normalised using the specifications from `global_specs`. So `indiv_specs` lists the exceptions to `global_specs`.

See also `vignette("normalise")` for more details.

Value

An updated coin

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin")

# normalise the raw data set
coin <- Normalise(coin, dset = "Raw")
```

`Normalise.data.frame` *Normalise a data frame*

Description

Normalises a data frame using specifications specified in `global_specs`. Columns can also optionally be normalised with individual specifications using the `indiv_specs` argument. If variables should have their directions reversed, this can be specified using the `directions` argument. Non-numeric columns are ignored automatically by this function. By default, this function normalises each indicator using the "min-max" method, scaling indicators to lie between 0 and 100. This calls the `n_minmax()` function. COINr has a number of built-in normalisation functions of the form `n_*()`. See online documentation for details.

Usage

```
## S3 method for class 'data.frame'
Normalise(x, global_specs = NULL, indiv_specs = NULL, directions = NULL, ...)
```

Arguments

`x`	A data frame
`global_specs`	Specifications to apply to all columns, apart from those specified by `indiv_specs`. See details.
`indiv_specs`	Specifications applied to specific columns, overriding those specified in `global_specs`. See details.
`directions`	An optional data frame containing the following columns: • `iCode` The indicator code, corresponding to the column names of the data frame • `Direction` numeric vector with entries either `-1` or `1` If `directions` is not specified, the directions will all be assigned as `1`. Non-numeric columns do not need to have directions assigned.
`...`	arguments passed to or from other methods.

Details

Global specification:

The `global_specs` argument is a list which specifies the normalisation function and any function parameters that should be used to normalise the columns of `x`. Unless `indiv_specs` is specified, this will be applied to all numeric columns of `x`. The list should have two entries:

- `.$f_n`: the name of the function to use to normalise each column
- `.$f_n_para`: any further parameters to pass to `f_n`, apart from the numeric vector (each column of `x`)

In this list, f_n should be a character string which is the name of a normalisation function. For example, f_n = "n_minmax" calls the n_minmax() function. f_n_para is a list of any further arguments to f_n. This means that any function can be passed to Normalise(), as long as its first argument is x, a numeric vector, and it returns a numeric vector of the same length. See n_minmax() for an example.

f_n_para is *required* to be a named list. So e.g. if we define a function f1(x, arg1, arg2) then we should specify f_n = "f1", and f_n_para = list(arg1 = val1, arg2 = val2), where val1 and val2 are the values assigned to the arguments arg1 and arg2 respectively.

The default list for global_specs is: list(f_n = "n_minmax", f_n_para = list(l_u = c(0,100))).

Note, all COINr normalisation functions (passed to f_n) are of the form n_*(). Type n_ in the R Studio console and press the Tab key to see a list.

Individual column specification:

Optionally, columns of x can be normalised with different normalisation functions and parameters using the indiv_specs argument. This must be specified as a named list e.g. list(i1 = specs1, i2 = specs2) where i1 and i2 are column names of x to apply individual normalisation to, and specs1 and specs2 are respectively lists of the same format as global_specs (see above). In other words, indiv_specs is a big list wrapping together global_specs-style lists. Any numeric columns of x not named in indiv_specs (i.e. those not in names(indiv_specs)) are normalised using the specifications from global_specs. So indiv_specs lists the exceptions to global_specs.

See also vignette("normalise") for more details.

Value

A normalised data frame

Examples

```
iris_norm <- Normalise(iris)
head(iris_norm)
```

Normalise.numeric *Normalise a numeric vector*

Description

Normalise a numeric vector using a specified function f_n, with possible reversal of direction using direction.

Usage

```
## S3 method for class 'numeric'
Normalise(x, f_n = NULL, f_n_para = NULL, direction = 1, ...)
```

Arguments

x	Object to be normalised
f_n	The normalisation method, specified as string which refers to a function of the form f_n(x, npara). See details. Defaults to "n_minmax" which is the min-max function.
f_n_para	Supporting list of arguments for f_n. This is required to be a list.
direction	If direction = -1 the highest values of x will correspond to the lowest values of the normalised x. Else if direction = 1 the direction of x in unaltered.
...	arguments passed to or from other methods.

Details

Normalisation is specified using the f_n and f_n_para arguments. In these, f_n should be a character string which is the name of a normalisation function. For example, f_n = "n_minmax" calls the n_minmax() function. f_n_para is a list of any further arguments to f_n. This means that any function can be passed to Normalise(), as long as its first argument is x, a numeric vector, and it returns a numeric vector of the same length. See n_minmax() for an example.

COINr has a number of built-in normalisation functions of the form n_*(). See online documentation for details.

f_n_para is *required* to be a named list. So e.g. if we define a function f1(x, arg1, arg2) then we should specify f_n = "f1", and f_n_para = list(arg1 = val1, arg2 = val2), where val1 and val2 are the values assigned to the arguments arg1 and arg2 respectively.

See also vignette("normalise") for more details.

Value

A normalised numeric vector

Examples

```
# example vector
x <- runif(10)

# normalise using distance to reference (5th data point)
x_norm <- Normalise(x, f_n = "n_dist2ref", f_n_para = list(iref = 5))

# view side by side
data.frame(x, x_norm)
```

`Normalise.purse` *Create normalised data sets in a purse of coins*

Description

This creates normalised data sets for each coin in the purse. In most respects, this works in a similar way to normalising on a coin, for which reason please see `Normalise.coin()` for most documentation. There is however a special case in terms of operating on a purse of coins. This is because, when dealing with time series data, it is often desirable to normalise over the whole panel data set at once rather than independently for each time point. This makes the resulting index and aggregates comparable over time. Here, the `global` argument controls whether to normalise each coin independently or to normalise across all data at once. In other respects, this function behaves the same as `Normalise.coin()`.

Usage

```
## S3 method for class 'purse'
Normalise(
  x,
  dset,
  global_specs = NULL,
  indiv_specs = NULL,
  directions = NULL,
  global = TRUE,
  write_to = NULL,
  ...
)
```

Arguments

`x`	A purse object
`dset`	The data set to normalise in each coin
`global_specs`	Default specifications
`indiv_specs`	Individual specifications
`directions`	An optional data frame containing the following columns: • `iCode` The indicator code, corresponding to the column names of the data set • `Direction` numeric vector with entries either `-1` or `1` If `directions` is not specified, the directions will be taken from the `iMeta` table in the coin, if available.
`global`	Logical: if `TRUE`, normalisation is performed "globally" across all coins, by using e.g. the max and min of each indicator in any coin. This effectively makes normalised scores comparable between coins because they are all scaled using the same parameters. Otherwise if `FALSE`, coins are normalised individually.

write_to	Optional character string for naming the data set in each coin. Data will be written to .$Data[[write_to]]. Default is write_to == "Normalised".
...	arguments passed to or from other methods.

Details

The same specifications are passed to each coin in the purse. This means that each coin is normalised using the same set of specifications and directions. If you need control over individual coins, you will have to normalise coins individually.

Value

An updated purse with new normalised data sets added at .$Data$Normalised in each coin

Examples

```
# build example purse
purse <- build_example_purse(up_to = "new_coin", quietly = TRUE)

# normalise raw data set
purse <- Normalise(purse, dset = "Raw", global = TRUE)
```

---

n_borda *Normalise using Borda scores*

Description

Calculates Borda scores as rank(x) - 1.

Usage

```
n_borda(x, ties.method = "min")
```

Arguments

x	A numeric vector
ties.method	This argument is passed to base::rank() - see there for details.

Value

Numeric vector

Examples

```
x <- runif(20)
n_borda(x)
```

n_dist2max *Normalise as distance to maximum value*

Description

A measure of the distance to the maximum value, where the maximum value is the highest-scoring value. The formula used is:

Usage

```
n_dist2max(x)
```

Arguments

x	A numeric vector

Details

$$1 - (x_{max} - x)/(x_{max} - x_{min})$$

This means that the closer a value is to the maximum, the higher its score will be. Scores will be in the range of 0 to 1.

Value

Numeric vector

Examples

```
x <- runif(20)
n_dist2max(x)
```

n_dist2ref *Normalise as distance to reference value*

Description

A measure of the distance to a specific value found in `x`, specified by `iref`. The formula is:

Usage

```
n_dist2ref(x, iref, cap_max = FALSE)
```

Arguments

x	A numeric vector
iref	An integer which indexes x to specify the reference value. The reference value will be x[iref].
cap_max	If TRUE, any value of x that exceeds x[iref] will be assigned a score of 1, otherwise will have a score greater than 1.

Details

$$1 - (x_{ref} - x)/(x_{ref} - x_{min})$$

Values exceeding x_ref can be optionally capped at 1 if cap_max = TRUE.

Value

Numeric vector

Examples

```
x <- runif(20)
n_dist2ref(x, 5)
```

---

n_dist2targ *Normalise as distance to target*

Description

A measure of the distance of each value of x to a specified target which can be a high or low target depending on direction. See details below.

Usage

```
n_dist2targ(x, targ, direction = 1, cap_max = FALSE)
```

Arguments

x	A numeric vector
targ	An target value
direction	Either 1 (default) or -1. In the former case, the indicator is assumed to be "positive" so that the target is at the higher end of the range. In the latter, the indicator is "negative" so that the target is typically at the low end of the range.
cap_max	If TRUE, any value of x that exceeds targ will be assigned a score of 1, otherwise will have a score greater than 1.

Details

If `direction = 1`, the formula is:

$$\frac{x - x_{min}}{x_{targ} - x_{min}}$$

else if `direction = -1`:

$$\frac{x_{max} - x}{x_{max} - x_{targ}}$$

Values surpassing `x_targ` in either case can be optionally capped at 1 if `cap_max = TRUE`.

This function also supports parameter specification in `iMeta` for the `Normalise.coin()` method. To do this, add columns `Target`, and `dist2targ_cap_max` to the `iMeta` table, which correspond to the `targ` and `cap_max` parameters respectively. Then set `f_n_para = "use_iMeta"` within the `global_specs` list. See also examples in the normalisation vignette.

Value

Numeric vector

Examples

```
x <- runif(20)
n_dist2targ(x, 0.8, cap_max = TRUE)
```

`n_fracmax` *Normalise as fraction of max value*

Description

The ratio of each value of `x` to `max(x)`.

Usage

```
n_fracmax(x)
```

Arguments

`x`	A numeric vector

Details

$$x/x_{max}$$

Value

Numeric vector

Examples

```
x <- runif(20)
n_fracmax(x)
```

`n_goalposts` *Normalise using goalpost method*

Description

The fraction of the distance of each value of `x` from the lower "goalpost" to the upper one. Goalposts are specified by `gposts = c(l, u, a)`, where `l` is the lower bound, `u` is the upper bound, and `a` is a scaling parameter.

Usage

```
n_goalposts(x, gposts, direction = 1, trunc2posts = TRUE)
```

Arguments

`x`	A numeric vector
`gposts`	A numeric vector `c(l, u, a)`, where `l` is the lower bound, `u` is the upper bound, and `a` is a scaling parameter.
`direction`	Either 1 or -1. Set to -1 to flip goalposts.
`trunc2posts`	If `TRUE` (default) will truncate any values that fall outside of the goalposts.

Details

Specify `direction = -1` to "flip" the goalposts. In this case, the fraction from the upper to the lower goalpost is measured.

The goalposts equations are:

$$(x - GP_{low})/(GP_{high} - GP_{low})$$

and for a negative directionality indicator:

$$(x - GP_{high})/(GP_{low} - GP_{high})$$

This function also supports parameter specification in `iMeta` for the `Normalise.coin()` method. To do this, add columns:

- `goalpost_lower`: the lower goalpost

- goalpost_upper: the upper goalpost
- goalpost_scale: the scaling parameter
- goalpost_trunc2posts: corresponds to the trunc2posts argument

to the iMeta table. Then set f_n_para = "use_iMeta" within the global_specs list. See also examples in the normalisation vignette.

Value

Numeric vector

Examples

```
# positive direction
n_goalposts(1, gposts = c(0, 10, 1))
# negative direction
n_goalposts(1, gposts = c(0, 10, 1), direction = -1)
```

---

n_minmax *Minmax a vector*

Description

Scales a vector using min-max method.

Usage

```
n_minmax(x, l_u = c(0, 100))
```

Arguments

x	A numeric vector
l_u	A vector c(l, u), where l is the lower bound and u is the upper bound. x will be scaled exactly onto this interval.

Details

This function also supports parameter specification in iMeta for the Normalise.coin() method. To do this, add columns minmax_lower, and minmax_upper to the iMeta table, which specify the lower and upper bounds to scale each indicator to. Then set f_n_para = "use_iMeta" within the global_specs list. See also examples in the normalisation vignette.

Value

Normalised vector

Examples

```
x <- runif(20)
n_minmax(x)
```

`n_prank` *Normalise using percentile ranks*

Description

Calculates percentile ranks of a numeric vector using "sport" ranking. Ranks are calculated by `base::rank()` and converted to percentile ranks. The `ties.method` can be changed - this is directly passed to `base::rank()`.

Usage

```
n_prank(x, ties.method = "min")
```

Arguments

`x`	A numeric vector
`ties.method`	This argument is passed to `base::rank()` - see there for details.

Value

Numeric vector

Examples

```
x <- runif(20)
n_prank(x)
```

`n_rank` *Normalise using ranks*

Description

This is simply a wrapper for `base::rank()`. Higher scores will give higher ranks.

Usage

```
n_rank(x, ties.method = "min")
```

Arguments

x	A numeric vector
ties.method	This argument is passed to base::rank() - see there for details.

Value

Numeric vector

Examples

```
x <- runif(20)
n_rank(x)
```

---

n_scaled *Scale a vector*

Description

Scales a vector for normalisation using the method applied in the GII2020 for some indicators. This does x_scaled <- (x-l)/(u-l) * scale_factor. Note this is *not* the minmax transformation (see n_minmax()). This is a linear transformation with shift u and scaling factor u-l.

Usage

```
n_scaled(x, npara = c(0, 100), scale_factor = 100)
```

Arguments

x	A numeric vector
npara	Parameters as a vector c(l, u). See description.
scale_factor	Optional scaling factor to apply to the result. Default 100.

Details

This function also supports parameter specification in iMeta for the Normalise.coin() method. To do this, add columns scaled_lower, scaled_upper and scale_factor to the iMeta table, which specify the first and second elements of npara, respectively. Then set f_n_para = "use_iMeta" within the global_specs list. See also examples in the normalisation vignette.

Value

Scaled vector

Examples

```
x <- runif(20)
n_scaled(x, npara = c(1,10))
```

n_zscore *Z-score a vector*

Description

Standardises a vector `x` by scaling it to have a mean and standard deviation specified by `m_sd`.

Usage

```
n_zscore(x, m_sd = c(0, 1))
```

Arguments

`x`	A numeric vector
`m_sd`	A vector `c(m, sd)`, where `m` is desired mean and `sd` is the target standard deviation.

Details

This function also supports parameter specification in `iMeta` for the `Normalise.coin()` method. To do this, add columns `zscore_mean`, and `zscore_sd` to the `iMeta` table, which specify the mean and standard deviation to scale each indicator to, respectively. Then set `f_n_para = "use_iMeta"` within the `global_specs` list. See also examples in the normalisation vignette.

Value

Numeric vector

Examples

```
x <- runif(20)
n_zscore(x)
```

outrankMatrix *Outranking matrix*

Description

Constructs an outranking matrix based on a data frame of indicator data and corresponding weights.

Usage

```
outrankMatrix(X, w = NULL)
```

Arguments

X	A data frame or matrix of indicator data, with observations as rows and indicators as columns. No other columns should be present (e.g. label columns).
w	A vector of weights, which should have length equal to ncol(X). Weights are relative and will be re-scaled to sum to 1. If w is not specified, defaults to equal weights.

Value

A list with:

- .$OutRankMatrix the outranking matrix with nrow(X) rows and columns (matrix class).
- .$nDominant the number of dominance/robust pairs
- .$fracDominant the percentage of dominance/robust pairs

Examples

```
# get a sample of a few indicators
ind_data <- COINr::ASEM_iData[12:16]
# calculate outranking matrix
outlist <- outrankMatrix(ind_data)
# see fraction of dominant pairs (robustness)
outlist$fracDominant
```

---

plot_bar *Bar chart*

Description

Plot bar charts of single indicators. Bar charts can be coloured by an optional grouping variable by_group, or if iCode points to an aggregate, setting stack_children = TRUE will plot iCode coloured by its underlying scores.

Usage

```
plot_bar(
  coin,
  dset,
  iCode,
  ...,
  uLabel = "uCode",
  axes_label = "iCode",
  by_group = NULL,
  filter_to_ends = NULL,
  dset_label = FALSE,
  log_scale = FALSE,
```

```
  stack_children = FALSE,
  bar_colours = NULL,
  flip_coords = FALSE
)
```

Arguments

coin	A coin object.
dset	Data set from which to extract the variable to plot. Passed to get_data().
iCode	Code of variable or indicator to plot. Passed to get_data().
...	Further arguments to pass to get_data(), e.g. for filtering units.
uLabel	How to label units: either "uCode", or "uName".
axes_label	How to label the y axis and group legend: either "iCode" or "iName".
by_group	Optional group variable to use to colour bars. Cannot be used if stack_children = TRUE.
filter_to_ends	Optional way to filter the bar chart to only display the top/bottom N units. This is useful in cases where the number of units is large. Specify as e.g. list(top = 10) or list(bottom = 10) to return only the top or bottom ten units respectively (the value 10 can be changed of course).
dset_label	Logical: whether to include the data set in the y axis label.
log_scale	Logical: if TRUE uses a log scale for the y axis.
stack_children	Logical: if TRUE and iCode refers to an aggregate, will plot iCode with each bar split into its underlying component values (the underlying indicators/aggregates used to create iCode). To use this, you must have aggregated your data and dset must point to a data set where the underlying (child) scores of iCode are available.
bar_colours	Optional vector of colour codes for colouring bars.
flip_coords	Logical; if TRUE flips to horizontal bars.

Details

This function uses ggplot2 to generate plots, so the plot can be further manipulated using ggplot2 commands. See vignette("visualisation) for more details on plotting.

Value

A ggplot2 plot object.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# bar plot of CO2 by GDP per capita group
plot_bar(coin, dset = "Raw", iCode = "CO2",
         by_group = "GDPpc_group", axes_label = "iName")
```

plot_corr *Static heatmaps of correlation matrices*

Description

Generates heatmaps of correlation matrices using ggplot2, which can be tailored according to the grouping and structure of the index. This enables correlating any set of indicators against any other, and supports calling named aggregation groups of indicators. The `withparent` argument generates tables of correlations only with parents of each indicator. Also supports discrete colour maps using `flagcolours`, different types of correlation, and groups plots by higher aggregation levels.

Usage

```
plot_corr(
  coin,
  dset,
  iCodes = NULL,
  Levels = 1,
  ...,
  cortype = "pearson",
  withparent = FALSE,
  grouplev = NULL,
  box_level = NULL,
  showvals = TRUE,
  flagcolours = FALSE,
  flagthresh = NULL,
  pval = 0.05,
  insig_colour = "#F0F0F0",
  text_colour = NULL,
  discrete_colours = NULL,
  box_colour = NULL,
  order_as = NULL,
  use_directions = FALSE
)
```

Arguments

`coin`	The coin object
`dset`	The target data set.
`iCodes`	An optional list of character vectors where the first entry specifies the indicator/aggregate codes to correlate against the second entry (also a specification of indicator/aggregate codes)
`Levels`	The aggregation levels to take the two groups of indicators from. See `get_data()` for details.
`...`	Optional further arguments to pass to `get_data()`.

`cortype` The type of correlation to calculate, either `"pearson"`, `"spearman"`, or `"kendall"` (see `stats::cor()`).

`withparent` If `aglev[1] != aglev[2]`, and equal `TRUE` will only plot correlations of each row with its parent. If `"family"`, plots the lowest aggregation level in `Levels` against all its parent levels. If `FALSE` plots the full correlation matrix (default).

`grouplev` The aggregation level to group correlations by if `aglev[1] == aglev[2]`. By default, groups correlations into the aggregation level above. Set to 0 to disable grouping and plot the full matrix.

`box_level` The aggregation level to draw boxes around if `aglev[1] == aglev[2]`.

`showvals` If `TRUE`, shows correlation values. If `FALSE`, no values shown.

`flagcolours` If `TRUE`, uses discrete colour map with thresholds defined by `flagthresh`. If `FALSE` uses continuous colour map.

`flagthresh` A 3-length vector of thresholds for highlighting correlations, if `flagcolours = TRUE`. `flagthresh[1]` is the negative threshold (default -0.4). Below this value, values will be flagged red. `flagthresh[2]` is the "weak" threshold (default 0.3). Values between `flagthresh[1]` and `flagthresh[2]` are coloured grey. `flagthresh[3]` is the "high" threshold (default 0.9). Anything between `flagthresh[2]` and `flagthresh[3]` is flagged "OK", and anything above `flagthresh[3]` is flagged "high".

`pval` The significance level for plotting correlations. Correlations with $p < pval$ will be shown, otherwise they will be plotted as the colour specified by `insig_colour`. Set to 0 to disable this.

`insig_colour` The colour to plot insignificant correlations. Defaults to a light grey.

`text_colour` The colour of the correlation value text (default white).

`discrete_colours`
An optional 4-length character vector of colour codes or names to define the discrete colour map if `flagcolours = TRUE` (from high to low correlation categories). Defaults to a green/blue/grey/purple.

`box_colour` The line colour of grouping boxes, default black.

`order_as` Optional list for ordering the plotting of variables. If specified, this must be a list of length 2, where each entry of the list is a character vector of the iCodes plotted on the x and y axes of the plot. The plot will then follow the order of these character vectors. Note this must be used with care because the `grouplev` and `boxlev` arguments will not follow the reordering. Hence this argument is probably best used for plots with no grouping, or for simply re-ordering within groups.

`use_directions` Logical: if `TRUE` the extracted data is adjusted using directions found inside the coin (i.e. the "Direction" column input in `iMeta`: any indicators with negative direction will have their values multiplied by -1 which will reverse the direction of correlation). This should only be set to `TRUE` if the data set has *not* yet been normalised. For example, this can be useful to set to `TRUE` to analyse correlations in the raw data, but would make no sense to analyse correlations in the normalised data because that already has the direction adjusted! So you would reverse direction twice. In other words, use this at your discretion.

Details

This function calls `get_corr()`.

Note that this function can only call correlations within the same data set (i.e. only one data set in `.$Data`).

This function uses ggplot2 to generate plots, so the plot can be further manipulated using ggplot2 commands. See `vignette("visualisation")` for more details on plotting.

This function replaces the now-defunct `plotCorr()` from COINr < v1.0.

Value

A plot object generated with ggplot2, which can be edited further with ggplot2 commands.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "Normalise", quietly = TRUE)

# plot correlations between indicators in Sust group, using Normalised dset
plot_corr(coin, dset = "Normalised", iCodes = list("Sust"),
          grouplev = 2, flagcolours = TRUE)
```

---

plot_dist *Static indicator distribution plots*

Description

Plots indicator distributions using box plots, dot plots, violin plots, violin-dot plots, and histograms. Supports plotting multiple indicators by calling aggregation groups.

Usage

```
plot_dist(
  coin,
  dset,
  iCodes,
  ...,
  type = "Box",
  normalise = FALSE,
  global_specs = NULL
)
```

Arguments

coin	The coin object, or a data frame of indicator data
dset	The name of the data set to apply the function to, which should be accessible in .$Data.
iCodes	Indicator code(s) to plot. See details.
...	Further arguments passed to get_data() (other than coin, dset and iCodes).
type	The type of plot. Currently supported "Box", "Dot", "Violin", "Violindot", "Histogram".
normalise	Logical: if TRUE, normalises the data first, using global_specs. If FALSE (default), data is not normalised.
global_specs	Specifications for normalising data if normalise = TRUE. This is passed to the global_specs argument of Normalise().

Details

This function uses ggplot2 to generate plots, so the plot can be further manipulated using ggplot2 commands. See vignette("visualisation) for more details on plotting.

This function replaces the now-defunct plotIndDist() from COINr < v1.0.

Value

A ggplot2 plot object.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin")

# plot all indicators in P2P group
plot_dist(coin, dset = "Raw", iCodes = "P2P", Level = 1, type = "Violindot")
```

---

plot_dot *Dot plots of single indicator with highlighting*

Description

Plots a single indicator as a line of dots, and optionally highlights selected units and statistics. This is intended for showing the relative position of units to other units, rather than as a statistical plot. For the latter, use plot_dist().

Usage

```
plot_dot(
  coin,
  dset,
  iCode,
  Level = NULL,
  ...,
  usel = NULL,
  marker_type = "circle",
  add_stat = NULL,
  stat_label = NULL,
  show_ticks = TRUE,
  plabel = NULL,
  usel_label = TRUE,
  vert_adjust = 0.5
)
```

Arguments

coin	The coin
dset	The name of the data set to apply the function to, which should be accessible in .$Data.
iCode	Code of indicator or aggregate found in dset. Required to be of length 1.
Level	The level in the hierarchy to extract data from. See get_data().
...	Further arguments to pass to get_data(), other than those explicitly specified here.
usel	A subset of units to highlight.
marker_type	The type of marker, either "circle" (default) or "cross", or a marker number to pass to ggplot2 (0-25).
add_stat	A statistic to overlay, either "mean", "median" or else a specified value.
stat_label	An optional string to use as label at the point specified by add_stat.
show_ticks	Set FALSE to remove axis ticks.
plabel	Controls the labelling of the indicator. If NULL (default), returns the indicator code. Otherwise if "iName", returns only indicator name, if "iName+unit", returns indicator name plus unit (if found), if "unit" returns only unit (if found), otherwise if "none", displays no text. Finally, any other string can be passed, so e.g. "My indicator" will display this on the axis.
usel_label	If TRUE (default) also labels selected units with their unit codes. FALSE to disable.
vert_adjust	Adjusts the vertical height of text labels and stat lines, which matters depending on plot size. Takes a value between 0 to 2 (higher will probably remove the label from the axis space).

Details

This function uses ggplot2 to generate plots, so the plot can be further manipulated using ggplot2 commands. See `vignette("visualisation")` for more details on plotting.

This function replaces the now-defunct `plotIndDot()` from COINr < v1.0.

Value

A ggplot2 plot object.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin")

# dot plot of LPI, highlighting two countries and with median shown
plot_dot(coin, dset = "Raw", iCode = "LPI", usel = c("JPN", "ESP"),
         add_stat = "median", stat_label = "Median", plabel = "iName+unit")
```

`plot_framework` *Framework plots*

Description

Plots the hierarchical indicator framework. If `type = "sunburst"` (default), the framework is plotted as a sunburst plot. If `type = "stack"` it is plotted as a linear stack. In both cases, the size of each component is reflected by its weight and the weight of its parent, i.e. its "effective weight" in the framework.

Usage

```
plot_framework(
  coin,
  type = "sunburst",
  colour_level = NULL,
  text_colour = NULL,
  text_size = NULL,
  transparency = TRUE
)
```

Arguments

`coin`	A coin class object
`type`	Either `"sunburst"` or `"stack"`.
`colour_level`	The framework level, as an integer, to colour from. See details.
`text_colour`	Colour of label text - default `"white"`.
`text_size`	Text size of labels, default 2.5
`transparency`	If `TRUE`, levels below `colour_level` are differentiated with some transparency.

Details

The colouring of the plot is defined to some extent by the `colour_level` argument. This should be specified as an integer between 1 and the highest level in the framework (i.e. the maximum of the `iMeta$Level` column). Levels higher than and including `colour_level` are coloured with individual colours from the standard colour palette. Any levels *below* `colour_level` are coloured with the same colours as their parents, to emphasise that they belong to the same group, and also to avoid repeating the colour palette. Levels below `colour_level` can be additionally differentiated by setting `transparency = TRUE` which will apply increasing transparency to lower levels.

This function returns a ggplot2 class object. If you want more control over the appearance of the plot, pass return the output of this function to a variable, and manipulate this further with ggplot2 commands to e.g. change colour palette, individual colours, add titles, etc. See `vignette("visualisation)` for more details on plotting.

This function replaces the now-defunct `plotframework()` from COINr < v1.0.

Value

A ggplot2 plot object

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# plot framework as sunburst, colouring at level 2 upwards
plot_framework(coin, colour_level = 2, transparency = TRUE)
```

---

plot_scatter *Scatter plot of two variables*

Description

This is a convenient quick scatter plot function for plotting any two variables x and y in a coin against each other. At a minimum, you must specify the data set and iCode of both x and y using the `dsets` and `iCodes` arguments.

Usage

```
plot_scatter(
  coin,
  dsets,
  iCodes,
  ...,
  by_group = NULL,
  alpha = 0.5,
  axes_label = "iCode",
  dset_label = TRUE,
```

```
  point_label = NULL,
  check_overlap = TRUE,
  nudge_y = 5,
  log_scale = c(FALSE, FALSE)
)
```

Arguments

coin	A coin object
dsets	A 2-length character vector specifying the data sets to extract v1 and v2 from, respectively (passed as dset argument to get_data(). Alternatively specify as a single string which will be used for both x and y.
iCodes	A 2-length character vector specifying the iCodes to use as v1 and v2, respectively (passed as iCodes argument to get_data(). Alternatively specify as a single string which will be used for both x and y.
...	Optional further arguments to be passed to get_data(), e.g. to specify which uCodes to plot.
by_group	A string specifying an optional group variable. If specified, the plot will be coloured by this grouping variable.
alpha	Transparency value for points between 0 and 1, passed to ggplot2.
axes_label	A string specifying how to label axes and legend. Either "iCode" to use the respective codes of each variable, or else "iName" to use the names (as specified in iMeta).
dset_label	Logical: if TRUE (default), also adds to the axis labels which data set each variable is from.
point_label	Specifies whether and how to label points. If "uCode", points are labelled with their unit codes, else if "uName", points are labelled with their unit names. Set NULL to remove labels (default).
check_overlap	Logical: if TRUE (default), point labels that overlap are removed - this results in a legible plot but some labels may be missing. Else if FALSE, all labels are plotted.
nudge_y	Parameter passed to ggplot which controls the vertical adjustment of the text labels if present.
log_scale	A 2-length logical vector specifying whether to use log axes for x and y respectively: if TRUE, a log axis will be used. Defaults to not-log.

Details

Optionally, the scatter plot can be coloured by grouping variables specified in the coin (see by_group). Points and axes can be labelled using other arguments.

This function is powered by ggplot2 and outputs a ggplot2 object. To further customise the plot, assign the output of this function to a variable and use ggplot2 commands to further edit. See vignette("visualisation) for more details on plotting.

Value

A ggplot2 object.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin")

# scatter plot of Flights against Population
# coloured by GDP per capita
# log scale applied to population
plot_scatter(coin, dsets = c("uMeta", "Raw"),
             iCodes = c("Population", "Flights"),
             by_group = "GDPpc_group", log_scale = c(TRUE, FALSE))
```

---

plot_sensitivity *Plot sensitivity indices*

Description

Plots sensitivity indices as bar or pie charts.

Usage

```
plot_sensitivity(SAresults, ptype = "bar")
```

Arguments

SAresults	A list of sensitivity/uncertainty analysis results from plot_sensitivity().
ptype	Type of plot to generate - either "bar", "pie" or "box".

Details

To use this function you first need to run get_sensitivity(). Then enter the resulting list as the SAresults argument here.

See vignette("sensitivity").

This function replaces the now-defunct plotSA() from COINr < v1.0.

Value

A plot of sensitivity indices generated by ggplot2.

See Also

- get_sensitivity() Perform global sensitivity or uncertainty analysis on a COIN
- plot_uncertainty() Plot confidence intervals on ranks following a sensitivity analysis

Examples

```
# for examples, see `vignette("sensitivity")`
# (this is because package examples are run automatically and sensitivity analysis
# can take a few minutes to run at realistic settings)
```

`plot_uncertainty` *Plot ranks from an uncertainty/sensitivity analysis*

Description

Plots the ranks resulting from an uncertainty and sensitivity analysis, in particular plots the median, and 5th/95th percentiles of ranks.

Usage

```
plot_uncertainty(
  SAresults,
  plot_units = NULL,
  order_by = "nominal",
  dot_colour = NULL,
  line_colour = NULL
)
```

Arguments

`SAresults`	A list of sensitivity/uncertainty analysis results from `get_sensitivity()`.
`plot_units`	A character vector of units to plot. Defaults to all units. You can also set to `"top10"` to only plot top 10 units, and `"bottom10"` for bottom ten.
`order_by`	If set to `"nominal"`, orders the rank plot by nominal ranks (i.e. the original ranks prior to the sensitivity analysis). Otherwise if `"median"`, orders by median ranks.
`dot_colour`	Colour of dots representing median ranks.
`line_colour`	Colour of lines connecting 5th and 95th percentiles.

Details

To use this function you first need to run `get_sensitivity()`. Then enter the resulting list as the `SAresults` argument here.

See `vignette("sensitivity")`.

This function replaces the now-defunct `plotSARanks()` from COINr < v1.0.

Value

A plot of rank confidence intervals, generated by 'ggplot2'.

See Also

- `get_sensitivity()` Perform global sensitivity or uncertainty analysis on a coin
- `plot_sensitivity()` Plot sensitivity indices following a sensitivity analysis.

Examples

```
# for examples, see `vignette("sensitivity")`
# (this is because package examples are run automatically and sensitivity analysis
# can take a few minutes to run at realistic settings)
```

`prc_change` *Percentage change of time series*

Description

Calculates the percentage change in a time series from the initial value. The time series is defined by `y` the response variable, indexed by `x`, the time variable. The `per` argument can optionally be used to scale the result according to a period of time. E.g. if the units of `x` are years, setting `x = 10` will measure the percentage change per decade.

Usage

```
prc_change(y, x, per = 1)
```

Arguments

`y`	A numeric vector
`x`	A numeric vector of the same length as `y`, indexing `y` in time. No `NA` values are allowed in `x`.
`per`	Numeric value to scale the change according to a period of time. See description.

Details

This function operates in two ways, depending on the number of data points. If `x` and `y` have two non-`NA` observations, percentage change is calculated using the first and last values. If three or more points are available, a linear regression is used to estimate the average percentage change. If fewer than two points are available, the percentage change cannot be estimated and `NA` is returned.

If all `y` values are equal, it will return a change of zero.

Value

Percentage change as a scalar value.

Examples

```
# a time vector
x <- 2011:2020

# some random points
y <- runif(10)

# find percentage change per decade
prc_change(y, x, 10)
```

`print.coin` *Print coin*

Description

Some details about the coin

Usage

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

Arguments

`x`	A coin
`...`	Arguments to be passed to or from other methods.

Value

Text output

`print.purse` *Print purse*

Description

Some details about the purse

Usage

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

Arguments

`x`	A purse
`...`	Arguments to be passed to or from other methods.

Value

Text output

`qNormalise` *Quick normalisation*

Description

This is a generic wrapper function for `Normalise()`, which offers a simpler syntax but less flexibility.

Usage

```
qNormalise(x, ...)
```

Arguments

`x`	Object to be normalised
`...`	arguments passed to or from other methods.

Details

See individual method documentation:

- `qNormalise.data.frame()`
- `qNormalise.coin()`
- `qNormalise.purse()`

Value

A normalised object

`qNormalise.coin` *Quick normalisation of a coin*

Description

This is a wrapper function for `Normalise()`, which offers a simpler syntax but less flexibility. It normalises a data set within a coin using a specified function `f_n` which is used to normalise each indicator, with additional function arguments passed by `f_n_para`. By default, `f_n = "n_minmax"` and `f_n_para` is set so that the indicators are normalised using the min-max method, between 0 and 100.

Usage

```
## S3 method for class 'coin'
qNormalise(
  x,
  dset,
  f_n = "n_minmax",
  f_n_para = list(l_u = c(0, 100)),
  directions = NULL,
  ...
)
```

Arguments

x	A coin
dset	Name of data set to normalise
f_n	Name of a normalisation function (as a string) to apply to each indicator. Default "n_minmax".
f_n_para	Any further arguments to pass to f_n, as a named list.
directions	An optional data frame containing the following columns: • iCode The indicator code, corresponding to the column names of the data frame • Direction numeric vector with entries either -1 or 1 If directions is not specified, the directions will be taken from the iMeta table in the coin, if available.
...	arguments passed to or from other methods.

Details

Essentially, this function is similar to Normalise() but brings parameters into the function arguments rather than being wrapped in a list. It also does not allow individual normalisation.

See Normalise() documentation for more details, and vignette("normalise").

Value

An updated coin with normalised data set.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# normalise raw data set using min max, but change to scale 1-10
coin <- qNormalise(coin, dset = "Raw", f_n = "n_minmax",
                   f_n_para = list(l_u = c(1,10)))
```

`qNormalise.data.frame` *Quick normalisation of a data frame*

Description

This is a wrapper function for `Normalise()`, which offers a simpler syntax but less flexibility. It normalises a data frame using a specified function `f_n` which is used to normalise each column, with additional function arguments passed by `f_n_para`. By default, `f_n = "n_minmax"` and `f_n_para` is set so that the columns of `x` are normalised using the min-max method, between 0 and 100.

Usage

```
## S3 method for class 'data.frame'
qNormalise(x, f_n = "n_minmax", f_n_para = NULL, directions = NULL, ...)
```

Arguments

`x`	A numeric data frame
`f_n`	Name of a normalisation function (as a string) to apply to each column of `x`. Default `"n_minmax"`.
`f_n_para`	Any further arguments to pass to `f_n`, as a named list. If `f_n = "n_minmax"`, this defaults to `list(l_u = c(0,100))` (scale between 0 and 100).
`directions`	An optional data frame containing the following columns: • `iCode` The indicator code, corresponding to the column names of the data frame • `Direction` numeric vector with entries either `-1` or `1` If `directions` is not specified, the directions will all be assigned as `1`. Non-numeric columns do not need to have directions assigned.
`...`	arguments passed to or from other methods.

Details

Essentially, this function is similar to `Normalise()` but brings parameters into the function arguments rather than being wrapped in a list. It also does not allow individual normalisation.

See `Normalise()` documentation for more details, and `vignette("normalise")`.

Value

A normalised data frame

Examples

```
# some made up data
X <- data.frame(uCode = letters[1:10],
                a = runif(10),
                b = runif(10)*100)
# normalise (defaults to min-max)
qNormalise(X)
```

---

qNormalise.purse *Quick normalisation of a purse*

Description

This is a wrapper function for `Normalise()`, which offers a simpler syntax but less flexibility. It normalises data sets within a purse using a specified function `f_n` which is used to normalise each indicator, with additional function arguments passed by `f_n_para`. By default, `f_n = "n_minmax"` and `f_n_para` is set so that the indicators are normalised using the min-max method, between 0 and 100.

Usage

```
## S3 method for class 'purse'
qNormalise(
  x,
  dset,
  f_n = "n_minmax",
  f_n_para = list(l_u = c(0, 100)),
  directions = NULL,
  global = TRUE,
  ...
)
```

Arguments

`x`	A purse
`dset`	Name of data set to normalise
`f_n`	Name of a normalisation function (as a string) to apply to each indicator. Default `"n_minmax"`.
`f_n_para`	Any further arguments to pass to `f_n`, as a named list.
`directions`	An optional data frame containing the following columns: • `iCode` The indicator code, corresponding to the column names of the data frame • `Direction` numeric vector with entries either `-1` or `1` If `directions` is not specified, the directions will be taken from the `iMeta` table in the coin, if available.

global	Logical: if TRUE, normalisation is performed "globally" across all coins, by using e.g. the max and min of each indicator in any coin. This effectively makes normalised scores comparable between coins because they are all scaled using the same parameters. Otherwise if FALSE, coins are normalised individually.
...	arguments passed to or from other methods.

Details

Essentially, this function is similar to Normalise() but brings parameters into the function arguments rather than being wrapped in a list. It also does not allow individual normalisation.

Normalisation can either be performed independently on each coin, or over the entire panel data set simultaneously. See the discussion in Normalise.purse() and vignette("normalise").

Value

An updated purse with normalised data sets

Examples

```
# build example purse
purse <- build_example_purse(up_to = "new_coin", quietly = TRUE)

# normalise using min-max, globally
purse <- qNormalise(purse, dset = "Raw", global = TRUE)
```

---

qTreat *Quick outlier treatment*

Description

This is a generic wrapper function for Treat(). It offers a simpler syntax but less flexibility.

Usage

```
qTreat(x, ...)
```

Arguments

x	Object to be normalised.
...	arguments passed to or from other methods.

Details

See individual method documentation:

- qTreat.data.frame()
- qTreat.coin()
- qTreat.purse()

Value

A treated object

Examples

```
# See individual method examples
```

---

qTreat.coin *Quick outlier treatment of a coin*

Description

A simplified version of `Treat()` which allows direct access to the default parameters. This has less flexibility, but is an easier interface and probably more convenient if the objective is to use the default treatment process but with some minor adjustments.

Usage

```
## S3 method for class 'coin'
qTreat(
  x,
  dset,
  winmax = 5,
  skew_thresh = 2,
  kurt_thresh = 3.5,
  f2 = "log_CT",
  ...
)
```

Arguments

x	A coin
dset	Name of data set to treat for outliers
winmax	Maximum number of points to Winsorise for each indicator. Default 5.
skew_thresh	Absolute skew threshold - default 2.
kurt_thresh	Kurtosis threshold - default 3.5.
f2	Function to call if Winsorisation does not bring skew and kurtosis within limits. Default "log_CT".
...	arguments passed to or from other methods.

Details

This function treats each indicator in the data set targeted by `dset` using the following process:

- First, it checks whether skew and kurtosis are within the specified limits of `skew_thresh` and `kurt_thresh`
- If the indicator is not within the limits, it applies the `winsorise()` function, with maximum number of winsorised points specified by `winmax`.
- If winsorisation does not bring the indicator within the skew/kurtosis limits, it is instead passed to `f2`, which is a second outlier treatment function, default `log_CT()`.

The arguments of `qTreat()` are passed to `Treat()`.

See `Treat()` documentation for more details, and `vignette("treat")`.

Value

An updated coin with treated data set at `.$Data$Treated`.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# treat with winmax = 3
coin <- qTreat(coin, dset = "Raw", winmax = 3)
```

---

qTreat.data.frame *Quick outlier treatment of a data frame*

Description

A simplified version of `Treat()` which allows direct access to the default parameters. This has less flexibility, but is an easier interface and probably more convenient if the objective is to use the default treatment process but with some minor adjustments.

Usage

```
## S3 method for class 'data.frame'
qTreat(x, winmax = 5, skew_thresh = 2, kurt_thresh = 3.5, f2 = "log_CT", ...)
```

Arguments

`x`	A numeric data frame
`winmax`	Maximum number of points to Winsorise for each column. Default 5.
`skew_thresh`	Absolute skew threshold - default 2.
`kurt_thresh`	Kurtosis threshold - default 3.5.
`f2`	Function to call if Winsorisation does not bring skew and kurtosis within limits. Default `"log_CT"`.
`...`	arguments passed to or from other methods.

Details

This function treats each column in x using the following process:

- First, it checks whether skew and kurtosis are within the specified limits of skew_thresh and kurt_thresh
- If the column is not within the limits, it applies the winsorise() function, with maximum number of winsorised points specified by winmax.
- If winsorisation does not bring the column within the skew/kurtosis limits, it is instead passed to f2, which is a second outlier treatment function, default log_CT().

The arguments of qTreat() are passed to Treat().

See Treat() documentation for more details, and vignette("treat").

Value

A list

Examples

```
# select three indicators
df1 <- ASEM_iData[c("Flights", "Goods", "Services")]

# treat data frame, changing winmax and skew/kurtosis limits
l_treat <- qTreat(df1, winmax = 1, skew_thresh = 1.5, kurt_thresh = 3)

# Now we check what the results are:
l_treat$Dets_Table
```

---

qTreat.purse *Quick outlier treatment of a purse*

Description

A simplified version of Treat() which allows direct access to the default parameters. This has less flexibility, but is an easier interface and probably more convenient if the objective is to use the default treatment process but with some minor adjustments.

Usage

```
## S3 method for class 'purse'
qTreat(
  x,
  dset,
  winmax = 5,
  skew_thresh = 2,
  kurt_thresh = 3.5,
```

```
  f2 = "log_CT",
  ...
)
```

Arguments

x	A purse
dset	Name of data set to treat for outliers in each coin
winmax	Maximum number of points to Winsorise for each indicator. Default 5.
skew_thresh	Absolute skew threshold - default 2.
kurt_thresh	Kurtosis threshold - default 3.5.
f2	Function to call if Winsorisation does not bring skew and kurtosis within limits. Default "log_CT".
...	arguments passed to or from other methods.

Details

This function simply applies the same data treatment to each coin. See documentation for `Treat.coin()`, `qTreat.coin()` and `vignette("treat")`.

Value

An updated purse

Examples

```
#
```

---

rank_df *Convert a data frame to ranks*

Description

Replaces all numerical columns of a data frame with their ranks. Uses sport ranking, i.e. ties share the highest rank place. Ignores non-numerical columns. See `rank()`. Optionally, returns in-group ranks using a specified grouping column.

Usage

```
rank_df(df, use_group = NULL)
```

Arguments

df	A data frame
use_group	An optional column of df (specified as a string) to use as a grouping variable. If specified, returns ranks inside each group present in this column.

Details

This function replaces the now-defunct `rankDF()` from COINr < v1.0.

Value

A data frame equal to the data frame that was input, but with any numerical columns replaced with ranks.

Examples

```
# some random data, with a column of characters
df <- data.frame(RName = c("A", "B", "C"),
Score1 = runif(3), Score2 = runif(3))
# convert to ranks
rank_df(df)
# grouped ranking - use some example data
df1 <- ASEM_iData[c("uCode", "GDP_group", "Goods", "LPI")]
rank_df(df1, use_group = "GDP_group")
```

Regen *Regenerate a coin or purse*

Description

Methods for regenerating coins and purses. Regeneration is re-running all the functions used to build the coin/purse, using the order and parameters found in the `.$Log` list of the coin.

Usage

```
Regen(x, from = NULL, quietly = TRUE)
```

Arguments

`x`	A coin or purse object to be regenerated
`from`	Optional: a construction function name. If specified, regeneration begins from this function, rather than re-running all functions.
`quietly`	If `TRUE` (default), messages are suppressed during building.

Details

Please see individual method documentation:

- `Regen.coin()`
- `Regen.purse()`

See also `vignette("adjustments")`.

This function replaces the now-defunct `regen()` from COINr < v1.0.

Value

A regenerated object

Examples

```
# see individual method examples
```

`Regen.coin` *Regenerate a coin*

Description

Regenerates the `.$Data` entries in a coin by rerunning the construction functions according to the specifications in `.$Log`. This effectively regenerates the results. Different variations of coins can be quickly achieved by editing the saved arguments in `.$Log` and regenerating.

Usage

```
## S3 method for class 'coin'
Regen(x, from = NULL, quietly = TRUE, ...)
```

Arguments

`x`	A coin class object
`from`	Optional: a construction function name. If specified, regeneration begins from this function, rather than re-running all functions.
`quietly`	If `TRUE` (default), messages are suppressed during building.
`...`	arguments passed to or from other methods.

Details

The `from` argument allows partial regeneration, starting from a specified function. This can be helpful to speed up regeneration in some cases. However, keep in mind that if you change a `.$Log` argument from a function that is run before the point that you choose to start running from, it will not affect the results.

Note that while sets of weights will be passed to the regenerated COIN, anything in `.$Analysis` will be removed and will have to be recalculated.

See also `vignette("adjustments")` for more info on regeneration.

Value

Updated coin object with regenerated results (data sets).

Examples

```
# build full example coin
coin <- build_example_coin(quietly = TRUE)

# copy coin
coin2 <- coin

# change to prank function (percentile ranks)
# we don't need to specify any additional parameters (f_n_para) here
coin2$Log$Normalise$global_specs <- list(f_n = "n_prank")

# regenerate
coin2 <- Regen(coin2)

# compare index, sort by absolute rank difference
compare_coins(coin, coin2, dset = "Aggregated", iCode = "Index",
              sort_by = "Abs.diff", decreasing = TRUE)
```

`Regen.purse` *Regenerate a purse*

Description

Regenerates the `.$Data` entries in all coins by rerunning the construction functions according to the specifications in `.$Log`, for each coin in the purse. This effectively regenerates the results.

Usage

```
## S3 method for class 'purse'
Regen(x, from = NULL, quietly = TRUE, ...)
```

Arguments

`x`	A purse class object
`from`	Optional: a construction function name. If specified, regeneration begins from this function, rather than re-running all functions.
`quietly`	If `TRUE` (default), messages are suppressed during building.
`...`	arguments passed to or from other methods.

Details

The `from` argument allows partial regeneration, starting from a specified function. This can be helpful to speed up regeneration in some cases. However, keep in mind that if you change a `.$Log` argument from a function that is run before the point that you choose to start running from, it will not affect the results.

Note that for the moment, regeneration of purses is only partially supported. This is because usually, in the normalisation step, it is necessary to normalise across the full panel data set (see the `global` argument in `Normalise()`). At the moment, purse regeneration is performed by regenerating each coin individually, but this does not allow for global normalisation which has to be done at the purse level. This may be fixed in future releases.

See also documentation for `Regen.coin()` and `vignette("adjustments")`.

Value

Updated purse object with regenerated results.

Examples

```
# see examples from Regen.coin() and vignette("adjustments")
```

`remove_elements` *Check the effect of removing indicators or aggregates*

Description

This is an analysis function for seeing what happens when elements of the composite indicator are removed. This can help with "what if" experiments and acts as different measure of the influence of each indicator or aggregate.

Usage

```
remove_elements(coin, Level, dset, iCode, quietly = FALSE)
```

Arguments

`coin`	A coin class object, which must be constructed up to and including the aggregation step, i.e. using `Aggregate()`.
`Level`	The level at which to remove elements. For example, `Level = 1` would check the effect of removing each indicator, one at a time. `Level = 2` would check the effect of removing each of the aggregation groups above the indicator level, one at a time.
`dset`	The name of the data set to take `iCode` from. Most likely this should be name of the aggregated data set, typically `"Aggregated"`.
`iCode`	A character string indicating the indicator or aggregate code to extract from each iteration. I.e. normally this would be set to the index code to compare the ranks of the index upon removing each indicator or aggregate. But it can be any code that is present in `.$Data[[dset]]`.
`quietly`	Logical: if `FALSE` (default) will output to the console an indication of progress. Might be useful when iterating over many indicators. Otherwise set to `TRUE` to shut this up.

Details

One way of looking at indicator "importance" in a composite indicator is via correlations. A different way is to see what happens if we remove the indicator completely from the framework. If removing an indicator or a whole aggregation of indicators results in very little rank change, it is one indication that perhaps it is not necessary to include it. Emphasis on *one*: there may be many other things to take into account.

This function works by successively setting the weight of each indicator or aggregate to zero. If the analysis is performed at the indicator level, it creates a copy of the coin, sets the weight of the first indicator to zero, regenerates the results, and compares to the nominal results (results when no weights are set to zero). It repeats this for each indicator in turn, such that each time one indicator is set to zero weights, and the others retain their original weights. The output is a series of tables comparing scores and ranks (see Value).

Note that "removing the indicator" here means more precisely "setting its weight to zero". In most cases the first implies the second, but check that the aggregation method that you are using satisfies this relationship. For example, if the aggregation method does not use any weights, then setting the weight to zero will have no effect.

This function replaces the now-defunct `removeElements()` from COINr < v1.0.

Value

A list with elements as follows:

- `.$Scores`: a data frame where each column is the scores for each unit, with indicator/aggregate corresponding to the column name removed. E.g. `.$Scores$Ind1` gives the scores resulting from removing "Ind1".
- `.$Ranks`: as above but ranks
- `.$RankDiffs`: as above but difference between nominal rank and rank on removing each indicator/aggregate
- `.$RankAbsDiffs`: as above but absolute rank differences
- `.$MeanAbsDiffs`: as above, but the mean of each column. So it is the mean (over units) absolute rank change resulting from removing each indicator or aggregate.

Examples

```
# build example coin
coin <- build_example_coin(quietly = TRUE)

# run function removing elements in level 2
l_res <- remove_elements(coin, Level = 3, dset = "Aggregated", iCode = "Index")

# get summary of rank changes
l_res$MeanAbsDiff
```

`replace_df` *Replace multiple values in a data frame*

Description

Given a data frame (or vector), this function replaces values according to a look up table or dictionary. In COINr this may be useful for exchanging categorical data with numeric scores, prior to assembly. Or for changing codes.

Usage

```
replace_df(df, lookup)
```

Arguments

`df`	A data frame or a vector
`lookup`	A data frame with columns `old` (the values to be replaced) and `new` the values to replace with. See details.

Details

The lookup data frame must not have any duplicated values in the `old` column. This function looks for exact matches of elements of the `old` column and replaces them with the corresponding value in the `new` column. For each row of `lookup`, the class of the old value must match the class of the new value. This is to keep classes of data frames columns consistent. If you wish to replace with a different class, you should convert classes in your data frame before using this function.

This function replaces the now-defunct `replaceDF()` from COINr < v1.0.

Value

A data frame with replaced values

Examples

```
# replace sub-pillar codes in ASEM indicator metadata
codeswap <- data.frame(old = c("Conn", "Sust"), new = c("SI1", "SI2"))
# swap codes in both iMeta
replace_df(ASEM_iMeta, codeswap)
```

round_df *Round down a data frame*

Description

Tiny function just to round down a data frame for display in a table, ignoring non-numeric columns.

Usage

```
round_df(df, decimals = 2)
```

Arguments

df	A data frame to input
decimals	The number of decimal places to round to (default 2)

Details

This function replaces the now-defunct `roundDF()` from COINr < v1.0.

Value

A data frame, with any numeric columns rounded to the specified amount.

Examples

```
round_df( as.data.frame(matrix(runif(20),10,2)), decimals = 3)
```

SA_estimate *Estimate sensitivity indices*

Description

Post process a sample to obtain sensitivity indices. This function takes a univariate output which is generated as a result of running a Monte Carlo sample from `SA_sample()` through a system. Then it estimates sensitivity indices using this sample.

Usage

```
SA_estimate(yy, N, d, Nboot = NULL)
```

Arguments

yy	A vector of model output values, as a result of a $N(d+2)$ Monte Carlo design.
N	The number of sample points per dimension.
d	The dimensionality of the sample
Nboot	Number of bootstrap draws for estimates of confidence intervals on sensitivity indices. If this is not specified, bootstrapping is not applied.

Details

This function is built to be used inside `get_sensitivity()`.

Value

A list with the output variance, plus a data frame of first order and total order sensitivity indices for each variable, as well as bootstrapped confidence intervals if `!is.null(Nboot)`.

See Also

- `get_sensitivity()` Perform global sensitivity or uncertainty analysis on a COIN
- `SA_sample()` Input design for estimating sensitivity indices

Examples

```
# This is a generic example rather than applied to a COIN (for reasons of speed)

# A simple test function
testfunc <- function(x){
x[1] + 2*x[2] + 3*x[3]
}

# First, generate a sample
X <- SA_sample(500, 3)

# Run sample through test function to get corresponding output for each row
y <- apply(X, 1, testfunc)

# Estimate sensitivity indices using sample
SAinds <- SA_estimate(y, N = 500, d = 3, Nboot = 1000)
SAinds$SensInd
# Notice that total order indices have narrower confidence intervals than first order.
```

SA_sample *Generate sample for sensitivity analysis*

Description

Generates an input sample for a Monte Carlo estimation of global sensitivity indices. Used in the `get_sensitivity()` function. The total sample size will be $N(d+2)$.

Usage

```
SA_sample(N, d)
```

Arguments

N	The number of sample points per dimension.
d	The dimensionality of the sample

Details

This function generates a Monte Carlo sample as described e.g. in the Global Sensitivity Analysis: The Primer book.

Value

A matrix with $N(d+2)$ rows and d columns.

See Also

- `get_sensitivity()` Perform global sensitivity or uncertainty analysis on a COIN.
- `SA_estimate()` Estimate sensitivity indices from system output, as a result of input design from SA_sample().

Examples

```
# sensitivity analysis sample for 3 dimensions with 100 points per dimension
X <- SA_sample(100, 3)
```

Screen *Screen units based on data availability*

Description

This is a generic function for screening units/rows based on data availability. See method documentation for more details:

Usage

```
Screen(x, ...)
```

Arguments

x	Object to be screened
...	arguments passed to or from other methods.

Details

This function replaces the now-defunct `checkData()` from COINr < v1.0.

- `Screen.data.frame()`
- `Screen.coin()`
- `Screen.purse()`

Value

An object of the same class as `x`

Screen.coin *Screen units based on data availability*

Description

Screens units based on a data availability threshold and presence of zeros. Units can be optionally "forced" to be included or excluded, making exceptions for the data availability threshold.

Usage

```
## S3 method for class 'coin'
Screen(
  x,
  dset,
  unit_screen,
  dat_thresh = NULL,
  nonzero_thresh = NULL,
  Force = NULL,
  out2 = "coin",
  write_to = NULL,
  ...
)
```

Arguments

x	A coin
dset	The data set to be checked/screened
unit_screen	Specifies whether and how to screen units based on data availability or zero values. • If set to "byNA", screens units with data availability below dat_thresh • If set to "byzeros", screens units with non-zero values below nonzero_thresh • If set to "byNAandzeros", screens units based on either of the previous two criteria being true.
dat_thresh	A data availability threshold (>= 1 and <= 0) used for flagging low data and screening units if unit_screen != "none". Default 0.66.
nonzero_thresh	As dat_thresh but for non-zero values. Defaults to 0.05, i.e. it will flag any units with less than 5% non-zero values (equivalently more than 95% zero values).
Force	A data frame with any additional countries to force inclusion or exclusion. Required columns uCode (unit code(s)) and Include (logical: TRUE to include and FALSE to exclude). Specifications here override exclusion/inclusion based on data rules.
out2	Where to output the results. If "COIN" (default for COIN input), appends to updated COIN, otherwise if "list" outputs to data frame.
write_to	If specified, writes the aggregated data to .$Data[[write_to]]. Default write_to = "Screened".
...	arguments passed to or from other methods.

Details

The two main criteria of interest are NA values, and zeros. The summary table gives percentages of NA values for each unit, across indicators, and percentage zero values (*as a percentage of non*-NA *values*). Each unit is flagged as having low data or too many zeros based on thresholds.

See also vignette("screening").

Value

An updated coin with data frames showing missing data in `.$Analysis`, and a new data set `.$Data$Screened`. If `out2 = "list"` wraps missing data stats and screened data set into a list.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin", quietly = TRUE)

# screen units from raw dset
coin <- Screen(coin, dset = "Raw", unit_screen = "byNA",
               dat_thresh = 0.85, write_to = "Filtered_85pc")

# some details about the coin by calling its print method
coin
```

Screen.data.frame *Screen units based on data availability*

Description

Screens units (rows) based on a data availability threshold and presence of zeros. Units can be optionally "forced" to be included or excluded, making exceptions for the data availability threshold.

Usage

```
## S3 method for class 'data.frame'
Screen(
  x,
  id_col = NULL,
  unit_screen,
  dat_thresh = NULL,
  nonzero_thresh = NULL,
  Force = NULL,
  ...
)
```

Arguments

`x`	A data frame
`id_col`	Name of column of the data frame to be used as the identifier, e.g. normally this would be `uCode` for indicator data sets used in coins. This must be specified if `Force` is specified.
`unit_screen`	Specifies whether and how to screen units based on data availability or zero values. • If set to `"byNA"`, screens units with data availability below `dat_thresh`

- If set to "byzeros", screens units with non-zero values below nonzero_thresh
- If set to "byNAandzeros", screens units based on either of the previous two criteria being true.

dat_thresh — A data availability threshold (>= 1 and <= 0) used for flagging low data and screening units if unit_screen != "none". Default 0.66.

nonzero_thresh — As dat_thresh but for non-zero values. Defaults to 0.05, i.e. it will flag any units with less than 5% non-zero values (equivalently more than 95% zero values).

Force — A data frame with any additional units to force inclusion or exclusion. Required columns uCode (unit code(s)) and Include (logical: TRUE to include and FALSE to exclude). Specifications here override exclusion/inclusion based on data rules.

... — arguments passed to or from other methods.

Details

The two main criteria of interest are NA values, and zeros. The summary table gives percentages of NA values for each unit, across indicators, and percentage zero values (*as a percentage of non*-NA *values*). Each unit is flagged as having low data or too many zeros based on thresholds.

See also vignette("screening").

Value

Missing data stats and screened data as a list.

Examples

```
# example data
iData <- ASEM_iData[40:51, c("uCode", "Research", "Pat", "CultServ", "CultGood")]

# screen to 75% data availability (by row)
l_scr <- Screen(iData, unit_screen = "byNA", dat_thresh = 0.75)

# summary of screening
head(l_scr$DataSummary)
```

Screen.purse *Screen units based on data availability*

Description

Screens units based on a data availability threshold and presence of zeros. Units can be optionally "forced" to be included or excluded, making exceptions for the data availability threshold.

Usage

```
## S3 method for class 'purse'
Screen(
  x,
  dset,
  unit_screen,
  dat_thresh = NULL,
  nonzero_thresh = NULL,
  Force = NULL,
  write_to = NULL,
  ...
)
```

Arguments

x	A purse object
dset	The data set to be checked/screened
unit_screen	Specifies whether and how to screen units based on data availability or zero values. • If set to "byNA", screens units with data availability below dat_thresh • If set to "byzeros", screens units with non-zero values below nonzero_thresh • If set to "byNAandzeros", screens units based on either of the previous two criteria being true.
dat_thresh	A data availability threshold (>= 1 and <= 0) used for flagging low data and screening units if unit_screen != "none". Default 0.66.
nonzero_thresh	As dat_thresh but for non-zero values. Defaults to 0.05, i.e. it will flag any units with less than 5% non-zero values (equivalently more than 95% zero values).
Force	A data frame with any additional countries to force inclusion or exclusion. Required columns uCode (unit code(s)) and Include (logical: TRUE to include and FALSE to exclude). Specifications here override exclusion/inclusion based on data rules.
write_to	If specified, writes the aggregated data to .$Data[[write_to]]. Default write_to = "Screened".
...	arguments passed to or from other methods.

Details

The two main criteria of interest are NA values, and zeros. The summary table gives percentages of NA values for each unit, across indicators, and percentage zero values (*as a percentage of non*-NA *values*). Each unit is flagged as having low data or too many zeros based on thresholds.

See also vignette("screening").

Value

An updated purse with coins screened and updated.

Examples

```
# see vignette("screening") for an example.
```

---

signif_df *Round a data frame to specified significant figures*

Description

Tiny function just to round down a data frame by significant figures for display in a table, ignoring non-numeric columns.

Usage

```
signif_df(df, digits = 3)
```

Arguments

df	A data frame to input
digits	The number of decimal places to round to (default 3)

Value

A data frame, with any numeric columns rounded to the specified amount.

Examples

```
signif_df( as.data.frame(matrix(runif(20),10,2)), digits = 3)
```

---

skew *Calculate skewness*

Description

Calculates skewness of the values of a numeric vector. This uses the same definition of skewness as the "skewness()" function in the "e1071" package where `type == 2`, which is equivalent to the definition of skewness used in Excel.

Usage

```
skew(x, na.rm = FALSE)
```

Arguments

x	A numeric vector.
na.rm	Set TRUE to remove NA values, otherwise returns NA.

Value

A skewness value (scalar).

Examples

```
x <- runif(20)
skew(x)
```

Treat *Treat outliers*

Description

Generic function for treating outliers using a two-step process. See individual method documentation:

Usage

```
Treat(x, ...)
```

Arguments

x	Object to be treated
...	arguments passed to or from other methods.

Details

- Treat.numeric()
- Treat.data.frame()
- Treat.coin()
- Treat.purse()

See also vignette("treat").

This function replaces the now-defunct treat() from COINr < v1.0.

Value

Treated object plus details.

Treat.coin *Treat a data set in a coin for outliers*

Description

Operates a two-stage data treatment process on the data set specified by `dset`, based on two data treatment functions, and a pass/fail function which detects outliers. The method of data treatment can be either specified by the `global_specs` argument (which applies the same specifications to all indicators in the specified data set), or else (additionally) by the `indiv_specs` argument which allows different methods to be applied for each indicator. See details. For a simpler function for data treatment, see the wrapper function `qTreat()`.

Usage

```
## S3 method for class 'coin'
Treat(
  x,
  dset,
  global_specs = NULL,
  indiv_specs = NULL,
  combine_treat = FALSE,
  out2 = "coin",
  write_to = NULL,
  write2log = TRUE,
  disable = FALSE,
  ...
)
```

Arguments

`x`	A coin
`dset`	A named data set available in `.$Data`
`global_specs`	A list specifying the treatment to apply to all columns. This will be applied to all columns, except any that are specified in the `indiv_specs` argument. Alternatively, set to `"none"` to apply no treatment. See details.
`indiv_specs`	A list specifying any individual treatment to apply to specific columns, overriding `global_specs` for those columns. See details.
`combine_treat`	By default, if `f1` fails to pass `f_pass`, then `f2` is applied to the original `x`, rather than the treated output of `f1`. If `combine_treat = TRUE`, `f2` will instead be applied to the output of `f1`, so the two treatments will be combined.
`out2`	The type of function output: either `"coin"` to return an updated coin, or `"list"` to return a list with treated data and treatment details.
`write_to`	If specified, writes the aggregated data to `.$Data[[write_to]]`. Default `write_to = "Treated"`.

write2log	Logical: if FALSE, the arguments of this function are not written to the coin log, so this function will not be invoked when regenerating. Recommend to keep TRUE unless you have a good reason to do otherwise.
disable	Logical: if TRUE will disable data treatment completely and write the unaltered data set. This option is mainly useful in sensitivity and uncertainty analysis (to test the effect of turning imputation on/off).
...	arguments passed to or from other methods.

Value

An updated coin with a new data set .Data$Treated added, plus analysis information in .$Analysis$Treated.

Global specifications

If the same method of data treatment should be applied to all indicators, use the global_specs argument. This argument takes a structured list which looks like this:

```
global_specs = list(f1 = .,
                    f1_para = list(.),
                    f2 = .,
                    f2_para = list(.),
                    f_pass = .,
                    f_pass_para = list()
                    )
```

The entries in this list correspond to arguments in Treat.numeric(), and the meanings of each are also described in more detail here below. In brief, f1 is the name of a function to apply at the first round of data treatment, f1_para is a list of any additional parameters to pass to f1, f2 and f2_para are equivalently the function name and parameters of the second round of data treatment, and f_pass and f_pass_para are the function and additional arguments to check for the existence of outliers.

The default values for global_specs are as follows:

```
global_specs = list(f1 = "winsorise",
                    f1_para = list(na.rm = TRUE,
                                   winmax = 5,
                                   skew_thresh = 2,
                                   kurt_thresh = 3.5,
                                   force_win = FALSE),
                    f2 = "log_CT",
                    f2_para = list(na.rm = TRUE),
                    f_pass = "check_SkewKurt",
                    f_pass_para = list(na.rm = TRUE,
                                       skew_thresh = 2,
                                       kurt_thresh = 3.5))
```

This shows that by default (i.e. if global_specs is not specified), each indicator is checked for outliers by the check_SkewKurt() function, which uses skew and kurtosis thresholds as its parameters.

Then, if outliers exist, the first function `winsorise()` is applied, which also uses skew and kurtosis parameters, as well as a maximum number of winsorised points. If the Winsorisation function does not satisfy `f_pass`, the `log_CT()` function is invoked.

To change the global specifications, you don't have to supply the whole list. If, for example, you are happy with all the defaults but want to simply change the maximum number of Winsorised points, you could specify e.g. `global_specs = list(f1_para = list(winmax = 3))`. In other words, a subset of the list can be specified, as long as the structure of the list is correct.

Individual specifications

The `indiv_specs` argument allows different specifications for each indicator. This is done by wrapping multiple lists of the format of the list described in `global_specs` into one single list, named according to the column names of `x`. For example, if the date set has indicators with codes "x1", "x2" and "x3", we could specify individual treatment as follows:

```
indiv_specs = list(x1 = list(.),
                   x2 = list(.)
                   x3 = list(.))
```

where each `list(.)` is a specifications list of the same format as `global_specs`. Any indicators that are *not* named in `indiv_specs` are treated using the specifications from `global_specs` (which will be the defaults if it is not specified). As with `global_specs`, a subset of the `global_specs` list may be specified for each entry. Additionally, as a special case, specifying a list entry as e.g. `x1 = "none"` will apply no data treatment to the indicator "x1". See `vignette("treat")` for examples of individual treatment.

Function methodology

This function is set up to allow any functions to be passed as the data treatment functions (`f1` and `f2`), as well as any function to be passed as the outlier detection function `f_pass`, as specified in the `global_specs` and `indiv_specs` arguments.

The arrangement of this function is inspired by a fairly standard data treatment process applied to indicators, which consists of checking skew and kurtosis, then if the criteria are not met, applying Winsorisation up to a specified limit. Then if Winsorisation still does not bring skew and kurtosis within limits, applying a nonlinear transformation such as log or Box-Cox.

This function generalises this process by using the following general steps:

1. Check if variable passes or fails using `f_pass`
2. If `f_pass` returns `FALSE`, apply `f1`, else return `x` unmodified
3. Check again using `*f_pass`
4. If `f_pass` still returns `FALSE`, apply `f2`
5. Return the modified `x` as well as other information.

For the "typical" case described above `f1` is a Winsorisation function, `f2` is a nonlinear transformation and `f_pass` is a skew and kurtosis check. Parameters can be passed to each of these three functions in a named list, for example to specify a maximum number of points to Winsorise, or Box-Cox parameters, or anything else. The constraints are that:

- All of `f1`, `f2` and `f_pass` must follow the format `function(x, f_para)`, where `x` is a numerical vector, and `f_para` is a list of other function parameters to be passed to the function, which is specified by `f1_para` for `f1` and similarly for the other functions. If the function has no parameters other than `x`, then `f_para` can be omitted.
- `f1` and `f2` should return either a list with `.$x` as the modified numerical vector, and any other information to be attached to the list, OR, simply `x` as the only output.
- `f_pass` must return a logical value, where `TRUE` indicates that the `x` passes the criteria (and therefore doesn't need any (more) treatment), and `FALSE` means that it fails to meet the criteria.

See also `vignette("treat")`.

Examples

```
# build example coin
coin <- build_example_coin(up_to = "new_coin")

# treat raw data set
coin <- Treat(coin, dset = "Raw")

# summary of treatment for each indicator
head(coin$Analysis$Treated$Dets_Table)
```

`Treat.data.frame` *Treat a data frame for outliers*

Description

Operates a two-stage data treatment process, based on two data treatment functions, and a pass/fail function which detects outliers. The method of data treatment can be either specified by the `global_specs` argument (which applies the same specifications to all columns in `x`), or else (additionally) by the `indiv_specs` argument which allows different methods to be applied for each column. See details. For a simpler function for data treatment, see the wrapper function `qTreat()`.

Usage

```
## S3 method for class 'data.frame'
Treat(x, global_specs = NULL, indiv_specs = NULL, combine_treat = FALSE, ...)
```

Arguments

`x`	A data frame. Can have both numeric and non-numeric columns.
`global_specs`	A list specifying the treatment to apply to all columns. This will be applied to all columns, except any that are specified in the `indiv_specs` argument. Alternatively, set to `"none"` to apply no treatment. See details.
`indiv_specs`	A list specifying any individual treatment to apply to specific columns, overriding `global_specs` for those columns. See details.

combine_treat — By default, if f1 fails to pass f_pass, then f2 is applied to the original x, rather than the treated output of f1. If combine_treat = TRUE, f2 will instead be applied to the output of f1, so the two treatments will be combined.

... — arguments passed to or from other methods.

Value

A treated data frame of data

Global specifications

If the same method of data treatment should be applied to all the columns, use the global_specs argument. This argument takes a structured list which looks like this:

```
global_specs = list(f1 = .,
                    f1_para = list(.),
                    f2 = .,
                    f2_para = list(.),
                    f_pass = .,
                    f_pass_para = list()
                    )
```

The entries in this list correspond to arguments in Treat.numeric(), and the meanings of each are also described in more detail here below. In brief, f1 is the name of a function to apply at the first round of data treatment, f1_para is a list of any additional parameters to pass to f1, f2 and f2_para are equivalently the function name and parameters of the second round of data treatment, and f_pass and f_pass_para are the function and additional arguments to check for the existence of outliers.

The default values for global_specs are as follows:

```
global_specs = list(f1 = "winsorise",
                    f1_para = list(na.rm = TRUE,
                                   winmax = 5,
                                   skew_thresh = 2,
                                   kurt_thresh = 3.5,
                                   force_win = FALSE),
                    f2 = "log_CT",
                    f2_para = list(na.rm = TRUE),
                    f_pass = "check_SkewKurt",
                    f_pass_para = list(na.rm = TRUE,
                                       skew_thresh = 2,
                                       kurt_thresh = 3.5))
```

This shows that by default (i.e. if global_specs is not specified), each column is checked for outliers by the check_SkewKurt() function, which uses skew and kurtosis thresholds as its parameters. Then, if outliers exist, the first function winsorise() is applied, which also uses skew and kurtosis parameters, as well as a maximum number of winsorised points. If the Winsorisation function does not satisfy f_pass, the log_CT() function is invoked.

To change the global specifications, you don't have to supply the whole list. If, for example, you are happy with all the defaults but want to simply change the maximum number of Winsorised points, you could specify e.g. `global_specs = list(f1_para = list(winmax = 3))`. In other words, a subset of the list can be specified, as long as the structure of the list is correct.

Individual specifications

The `indiv_specs` argument allows different specifications for each column in `x`. This is done by wrapping multiple lists of the format of the list described in `global_specs` into one single list, named according to the column names of `x`. For example, if `x` has column names "x1", "x2" and "x3", we could specify individual treatment as follows:

```
indiv_specs = list(x1 = list(.),
                   x2 = list(.)
                   x3 = list(.))
```

where each `list(.)` is a specifications list of the same format as `global_specs`. Any columns that are not named in `indiv_specs` are treated using the specifications from `global_specs` (which will be the defaults if it is not specified). As with `global_specs`, a subset of the `global_specs` list may be specified for each entry. Additionally, as a special case, specifying a list entry as e.g. `x1 = "none"` will apply no data treatment to the column "x1". See `vignette("treat")` for examples of individual treatment.

Function methodology

This function is set up to allow any functions to be passed as the data treatment functions (`f1` and `f2`), as well as any function to be passed as the outlier detection function `f_pass`, as specified in the `global_specs` and `indiv_specs` arguments.

The arrangement of this function is inspired by a fairly standard data treatment process applied to indicators, which consists of checking skew and kurtosis, then if the criteria are not met, applying Winsorisation up to a specified limit. Then if Winsorisation still does not bring skew and kurtosis within limits, applying a nonlinear transformation such as log or Box-Cox.

This function generalises this process by using the following general steps:

1. Check if variable passes or fails using `f_pass`
2. If `f_pass` returns `FALSE`, apply `f1`, else return `x` unmodified
3. Check again using `*f_pass`
4. If `f_pass` still returns `FALSE`, apply `f2`
5. Return the modified `x` as well as other information.

For the "typical" case described above `f1` is a Winsorisation function, `f2` is a nonlinear transformation and `f_pass` is a skew and kurtosis check. Parameters can be passed to each of these three functions in a named list, for example to specify a maximum number of points to Winsorise, or Box-Cox parameters, or anything else. The constraints are that:

- All of `f1`, `f2` and `f_pass` must follow the format `function(x, f_para)`, where `x` is a numerical vector, and `f_para` is a list of other function parameters to be passed to the function, which is specified by `f1_para` for `f1` and similarly for the other functions. If the function has no parameters other than `x`, then `f_para` can be omitted.

- f1 and f2 should return either a list with .$x as the modified numerical vector, and any other information to be attached to the list, OR, simply x as the only output.
- f_pass must return a logical value, where TRUE indicates that the x passes the criteria (and therefore doesn't need any (more) treatment), and FALSE means that it fails to meet the criteria.

See also vignette("treat").

Examples

```
# select three indicators
df1 <- ASEM_iData[c("Flights", "Goods", "Services")]

# treat the data frame using defaults
l_treat <- Treat(df1)

# details of data treatment for each column
l_treat$Dets_Table
```

Treat.numeric *Treat a numeric vector for outliers*

Description

Operates a two-stage data treatment process, based on two data treatment functions, and a pass/fail function which detects outliers. This function is set up to allow any functions to be passed as the data treatment functions (f1 and f2), as well as any function to be passed as the outlier detection function f_pass.

Usage

```
## S3 method for class 'numeric'
Treat(
  x,
  f1,
  f1_para = NULL,
  f2 = NULL,
  f2_para = NULL,
  f_pass,
  f_pass_para = NULL,
  combine_treat = FALSE,
  ...
)
```

Arguments

x	A numeric vector.
f1	First stage data treatment function e.g. as a string.
f1_para	First stage data treatment function parameters as a named list.
f2	First stage data treatment function as a string.
f2_para	First stage data treatment function parameters as a named list.
f_pass	A string specifying an outlier detection function - see details. Default "check_SkewKurt"
f_pass_para	Any further arguments to pass to f_pass(), as a named list.
combine_treat	By default, if f1 fails to pass f_pass, then f2 is applied to the original x, rather than the treated output of f1. If combine_treat = TRUE, f2 will instead be applied to the output of f1, so the two treatments will be combined.
...	arguments passed to or from other methods.

Details

The arrangement of this function is inspired by a fairly standard data treatment process applied to indicators, which consists of checking skew and kurtosis, then if the criteria are not met, applying Winsorisation up to a specified limit. Then if Winsorisation still does not bring skew and kurtosis within limits, applying a nonlinear transformation such as log or Box-Cox.

This function generalises this process by using the following general steps:

1. Check if variable passes or fails using f_pass
2. If f_pass returns FALSE, apply f1, else return x unmodified
3. Check again using *f_pass
4. If f_pass still returns FALSE, apply f2 (by default to the original x, see combine_treat parameter)
5. Return the modified x as well as other information.

For the "typical" case described above f1 is a Winsorisation function, f2 is a nonlinear transformation and f_pass is a skew and kurtosis check. Parameters can be passed to each of these three functions in a named list, for example to specify a maximum number of points to Winsorise, or Box-Cox parameters, or anything else. The constraints are that:

- All of f1, f2 and f_pass must follow the format function(x, f_para), where x is a numerical vector, and f_para is a list of other function parameters to be passed to the function, which is specified by f1_para for f1 and similarly for the other functions. If the function has no parameters other than x, then f_para can be omitted.
- f1 and f2 should return either a list with .$x as the modified numerical vector, and any other information to be attached to the list, OR, simply x as the only output.
- f_pass must return a logical value, where TRUE indicates that the x passes the criteria (and therefore doesn't need any (more) treatment), and FALSE means that it fails to meet the criteria.

See also vignette("treat").

Value

A treated vector of data.

Examples

```
# numbers between 1 and 10
x <- 1:10

# two outliers
x <- c(x, 30, 100)

# check whether passes skew/kurt test
check_SkewKurt(x)

# treat using winsorisation
l_treat <- Treat(x, f1 = "winsorise", f1_para = list(winmax = 2),
                 f_pass = "check_SkewKurt")

# plot original against treated
plot(x, l_treat$x)
```

`Treat.purse` *Treat a purse of coins for outliers*

Description

This function calls `Treat.coin()` for each coin in the purse. See the documentation of that function for details. See also `vignette("treat")`.

Usage

```
## S3 method for class 'purse'
Treat(
  x,
  dset,
  global_specs = NULL,
  indiv_specs = NULL,
  combine_treat = FALSE,
  write_to = NULL,
  disable = FALSE,
  ...
)
```

Arguments

`x`	A purse object
`dset`	The data set to treat in each coin.

global_specs	Default specifications. See details in Treat.coin().
indiv_specs	Individual specifications. See details in Treat.coin().
combine_treat	By default, if f1 fails to pass f_pass, then f2 is applied to the original x, rather than the treated output of f1. If combine_treat = TRUE, f2 will instead be applied to the output of f1, so the two treatments will be combined.
write_to	If specified, writes the aggregated data to .$Data[[write_to]]. Default write_to = "Treated".
disable	Logical: if TRUE will disable data treatment completely and write the unaltered data set. This option is mainly useful in sensitivity and uncertainty analysis (to test the effect of turning imputation on/off).
...	arguments passed to or from other methods.

Value

An updated purse with new treated data sets added at .$Data$Treated in each coin, plus analysis information at .$Analysis$Treated

Examples

```
# See `vignette("treat")`.
```

---

ucodes_to_unames *Convert uCodes to uNames*

Description

Convert uCodes to uNames

Usage

```
ucodes_to_unames(coin, uCodes)
```

Arguments

coin	A coin
uCodes	A vector of uCodes

Value

Vector of uNames

`winsorise` *Winsorise a vector*

Description

Follows a "standard" Winsorisation approach: points are successively Winsorised in order to bring skew and kurtosis thresholds within specified limits. Specifically, aims to bring absolute skew to below a threshold (default 2.25) and kurtosis below another threshold (default 3.5).

Usage

```
winsorise(
  x,
  na.rm = FALSE,
  winmax = 5,
  skew_thresh = 2,
  kurt_thresh = 3.5,
  force_win = FALSE
)
```

Arguments

`x`	A numeric vector.
`na.rm`	Set `TRUE` to remove `NA` values, otherwise returns `NA`.
`winmax`	Maximum number of points to Winsorise. Default 5. Set `NULL` to have no limit.
`skew_thresh`	A threshold for absolute skewness (positive). Default 2.25.
`kurt_thresh`	A threshold for kurtosis. Default 3.5.
`force_win`	Logical: if `TRUE`, forces winsorisation up to winmax (regardless of skew/kurt). Default `FALSE`. Note - this option should be used with care because the direction of Winsorisation is based on the direction of skew. Successively Winsorising can switch the direction of skew and hence the direction of Winsorisation, which may not produce the expected behaviour.

Details

Winsorisation here is defined as reassigning the point with the highest/lowest value with the value of the next highest/lowest point. Whether to Winsorise at the high or low end of the scale is decided by the direction of the skewness of `x`.

This function replaces the now-defunct `coin_win()` from COINr < v1.0.

Value

A list containing winsorised data, number of winsorised points, and the individual points that were treated.

Examples

```
# numbers between 1 and 10
x <- 1:10

# two outliers
x <- c(x, 30, 100)

# winsorise
l_win <- winsorise(x, skew_thresh = 2, kurt_thresh = 3.5)

# see treated vector, number of winsorised points and details
l_win
```

WorldDenoms *World denomination data*

Description

A small selection of common denominator indicators, which includes GDP, Population, Area, GDP per capita and income group. All data sourced from the World Bank as of Feb 2021 (data is typically from 2019). Note that this is intended as example data, and it would be a good idea to use updated data from the World Bank when needed. In this data set, country names have been altered slightly so as to include no accents - this is simply to make it more portable between distributions.

Usage

```
WorldDenoms
```

Format

A data frame with 249 rows and 7 variables.

Source

https://data.worldbank.org/

Index