Package: piar (via r-universe)

October 27, 2026

Title Price Index Aggregation

Version 0.11.0.9001

Description Most price indexes are made with a two-step procedure, where period-over-period elementary indexes are first calculated for a collection of elementary aggregates at each point in time, and then aggregated according to a price index aggregation structure. These indexes can then be chained together to form a time series that gives the evolution of prices with respect to a fixed base period. This package contains a collection of functions that revolve around this work flow, making it easy to build standard price indexes, and implement the methods described by Balk (2008, <doi:10.1017/CBO9780511720758>), von der Lippe (2007, <doi:10.3726/978-3-653-01120-3>), and the CPI manual (2020, <doi:10.5089/9781484354841.069>) and PPI manual (2004, <doi:10.5089/9781589063044.069>) for bilateral price indexes.

Depends R (>= 4.4)

Imports stats, utils

Suggests data.tree, litedown (>= 0.11), Matrix (>= 1.5-0), rbibutils, rsmatrix, spelling, sps, tinytest, treemap

License GPL (>= 3)

Encoding UTF-8

BugReports https://codeberg.org/marberts/piar/issues

LazyData true

VignetteBuilder litedown

Roxygen list(markdown = TRUE)

Config/roxygen2/version 8.1.0

Language en-US

Repository https://marberts.r-universe.dev

Date/Publication 2026-10-27 02:35:55 UTC

RemoteUrl https://codeberg.org/marberts/piar

RemoteRef HEAD

RemoteSha 6249ac12ce3e00296545ab718891fb70c10f346f

Contents

[.piar_index . . . 3
aggregate.piar_index . . . 4
aggregation_structure . . . 7
as.data.frame.piar_index . . . 9
as.matrix.piar_aggregation_structure . . . 10
as.ts.piar_index . . . 12
as_aggregation_structure . . . 13
as_index . . . 14
back_period . . . 16
chain . . . 17
combine_classifications . . . 19
contrib . . . 20
cut.piar_aggregation_structure . . . 22
elementary_index . . . 23
emean . . . 26
expand_classification . . . 28
geks_index . . . 29
gmean . . . 31
head.piar_index . . . 33
impute_prices . . . 34
is.na.piar_index . . . 37
is_aggregation_structure . . . 38
is_index . . . 39
lag.piar_index . . . 39
levels.piar_aggregation_structure . . . 40
levels.piar_index . . . 41
mean.piar_index . . . 41
merge.piar_index . . . 44
nested_gmean . . . 45
outliers . . . 46
piar_index . . . 48
price_data . . . 48
price_relative . . . 49
scale_weights . . . 50
splice_index . . . 50
split.piar_index . . . 51
split_classification . . . 52
stack.piar_index . . . 53
time.piar_index . . . 55
transmute_weights . . . 56
transmute_weights2 . . . 57
update.piar_aggregation_structure . . . 59

update_weights 60
weights.piar_aggregation_structure 61
window.piar_index 63

Index **65**

`[.piar_index` *Extract index values*

Description

Methods to extract and replace index values like a matrix.

Usage

```
## S3 method for class 'piar_index'
x[i, j, ...]

## S3 replacement method for class 'piar_index'
x[i, j, ...] <- value
```

Arguments

`x`	`[piar_index]` A price index, as made by, e.g., `elementary_index()`.
`i, j`	`[integer | logical | character | matrix]` Indices for the levels and time periods of a price index. See details.
`...`	Not currently used.
`value`	`[numeric > 0 | piar_index | list]` A numeric vector, price index, or list of price indexes. See details.

Details

The extraction method treats `x` like a matrix of index values with (named) rows for each level and columns for each time period in `x`. Unlike a matrix, dimensions are never dropped as subscripting `x` always returns an index object. The one exception is when subscripting with a matrix, in which case a list of index values is returned. As `x` is not an atomic vector, subscripting with a single index like `x[1]` extracts all time periods for that level.

The levels and time periods of an index must be unique, so it is not possible to expand the number of levels or time periods when subscripting like with a matrix. Instead, passing a named character vector or vector of positive integers to subscript `x` will use these names to assign new levels and/or time periods. This makes it possible to assign unique labels with duplicated subscripts.

The replacement method similarly treats `x` like a matrix. If `value` is an index object with the same number of time periods as `x[i, j]` and it inherits from the same class as `x`, then the index values and percent-change contributions of `x[i, j]` are replaced with those for the corresponding levels of `value`. If `value` is not an index, then it is coerced to a numeric vector and behaves the same as replacing values in a matrix. Note that replacing the values of an index will remove the corresponding percent-change contributions (if any). When replacing with a matrix, `value` can be a list of index objects.

Value

A price index that inherits from the same class as x.

See Also

Other index methods: aggregate.piar_index, as.data.frame.piar_index(), as.ts.piar_index(), chain(), contrib(), head.piar_index(), is.na.piar_index(), lag.piar_index(), levels.piar_index(), mean.piar_index, merge.piar_index(), split.piar_index(), stack.piar_index(), time.piar_index(), window.piar_index()

Examples

```
index <- as_index(matrix(1:6, 2))

index["1", ]

index[, 2]

index[1, ] <- 1 # can be useful for doing specific imputations

index
```

---

aggregate.piar_index *Aggregate elementary price indexes*

Description

Aggregate elementary price indexes with a price index aggregation structure.

Usage

```
## S3 method for class 'chainable_piar_index'
aggregate(
  x,
  pias,
  ...,
  pias2 = NULL,
  na.rm = FALSE,
  contrib = TRUE,
  order = 1,
  r = order,
  include_ea = TRUE,
  duplicate_contrib = c("sum", "make.unique"),
  impute_rules = NULL
)

## S3 method for class 'direct_piar_index'
aggregate(
```

```
  x,
  pias,
  ...,
  pias2 = NULL,
  na.rm = FALSE,
  contrib = TRUE,
  order = 1,
  r = order,
  include_ea = TRUE,
  duplicate_contrib = c("sum", "make.unique"),
  impute_rules = NULL
)
```

Arguments

x	[piar_index] A price index, usually made by elementary_index().
pias	[piar_aggregation_structure] A price index aggregation structure or something that can be coerced into one. This can be made with aggregation_structure().
...	Not currently used.
pias2	[piar_aggregation_structure] An optional secondary aggregation structure, usually with current-period weights, to make a superlative index. See details.
na.rm	[logical(1)] Should missing values be removed? By default, missing values are not removed. Setting na.rm = TRUE is equivalent to overall-mean imputation.
contrib	[logical(1)] Aggregate percent-change contributions in x? By default contributions are aggregated.
order	[numeric(1)] Order of the generalized mean to aggregate index values. 0 for a geometric index (the default for making elementary indexes), 1 for an arithmetic index (the default for aggregating elementary indexes and averaging indexes over subperiods), or -1 for a harmonic index (usually for a Paasche index). Other values are possible; see gmean() for details. If pias2 is given then the index is aggregated with a quadratic mean of order 2*order.
r	Deprecated.
include_ea	[logical(1)] Should indexes for the elementary aggregates be included along with the aggregated indexes? By default, all index values are returned.
duplicate_contrib	[character(1)] The method to deal with duplicate product contributions. Either "make.unique" to treat duplicate products as distinct products and make their names unique with base::make.unique() or "sum" to add contributions for each product (the default).
impute_rules	[function] (Experimental) A function that applies imputation rules to the elementary indexes in each time period prior to aggregation. It takes two arguments, the elementary indexes for a given time period and the (price updated) aggregation structure, and must return back the elementary indexes.

Details

The aggregate() method loops over each time period in x and

1. aggregates the elementary indexes with `gmean()` for each level of `pias`;
2. aggregates percent-change contributions for each level of `pias` (if there are any and `contrib = TRUE`);
3. price updates the weights in `pias` with `update_weights()` (only for period-over-period elementary indexes).

The result is a collection of aggregated period-over-period indexes that can be chained together to get a fixed-base index when `x` are period-over-period elementary indexes. Otherwise, when `x` are fixed-base elementary indexes, the result is a collection of aggregated fixed-base (direct) indexes.

By default, missing elementary indexes will propagate when aggregating the index. Missing elementary indexes can be due to both missingness of these values in `x`, and the presence of elementary aggregates in `pias` that are not part of `x`. Setting `na.rm = TRUE` ignores missing values, and is equivalent to overall-mean (or parental) imputation. As an aggregated price index generally cannot have missing values (for otherwise it can't be chained over time and weights can't be price updated), any missing values for a level of `pias` are removed and recursively replaced by the value of its immediate parent.

In most cases aggregation is done with an arithmetic mean (the default), and this is detailed in the CPI manual (IMF, ILO, Eurostat, UNECE, OECD, and World Bank 2020, chapter 8, pp. 190–198), with analogous details in the PPI manual (ILO, IMF, OECD, UNECE, and World Bank 2004, chapter 9). Aggregating with a non-arithmetic mean follows the same steps, except that the elementary indexes are aggregated with a mean of a different order (e.g., harmonic for a Paasche index), and the method for price updating the weights is slightly different. Note that, because aggregation is done with a generalized mean, the resulting index is consistent-in-aggregation at each point in time.

Aggregating percent-change contributions uses the method in the CPI manual (IMF et al. 2020, equations 9.26 and 9.28) when aggregating with an arithmetic mean. With a non-arithmetic mean, arithmetic weights are constructed using `transmute_weights()` in order to apply this method.

There may not be contributions for all prices relatives in an elementary aggregate if the elementary indexes are built from several sources (as with `merge()`). In this case the contribution for a price relative in the aggregated index will be correct, but the sum of all contributions will not equal the change in the value of the index.

If two aggregation structures are given then the steps above are done for each aggregation structure, with the aggregation for `pias` done with a generalized mean of order `order` the aggregation for `pias2` done with a generalized mean of order `-order`. The resulting indexes are combined with a geometric mean to make a superlative quadratic mean of order `2*order` index. Percent-change contributions are combined using a generalized van IJzeren decomposition; see `transmute_weights2()` for details.

Value

An aggregate price index that inherits from the class of `x`.

Note

For large indexes it can be much faster to turn the aggregation structure into an aggregation matrix with `as.matrix()`, then aggregate elementary indexes as a matrix operation when there are no missing values. See the examples for details.

References

Balk BM (2008). *Price and Quantity Index Numbers*. Cambridge University Press. doi:10.1017/CBO9780511720758.

ILO, IMF, OECD, UNECE, World Bank (2004). *Producer price index manual: Theory and practice*. International Monetary Fund. doi:10.5089/9781589063044.069.

IMF, ILO, Eurostat, UNECE, OECD, World Bank (2020). *Consumer price index manual: Concepts and methods*. International Monetary Fund. doi:10.5089/9781484354841.069.

von der Lippe P (2007). *Index Theory and Price Statistics*. Peter Lang. doi:10.3726/9783653-011203.

See Also

Other index methods: `[.piar_index()`, `as.data.frame.piar_index()`, `as.ts.piar_index()`, `chain()`, `contrib()`, `head.piar_index()`, `is.na.piar_index()`, `lag.piar_index()`, `levels.piar_index()`, `mean.piar_index`, `merge.piar_index()`, `split.piar_index()`, `stack.piar_index()`, `time.piar_index()`, `window.piar_index()`

Examples

```
prices <- data.frame(
  rel = 1:8,
  period = rep(1:2, each = 4),
  ea = rep(letters[1:2], 4)
)

# A two-level aggregation structure.
pias <- aggregation_structure(
  list(c("top", "top", "top"), c("a", "b", "c")),
  weights = 1:3
)

# Calculate Jevons elementary indexes.
(elementary <- elementary_index(prices, rel ~ period + ea))

# Aggregate (note the imputation for elementary index 'c').
(index <- aggregate(elementary, pias, na.rm = TRUE))

# Aggregation can equivalently be done as matrix multiplication.
as.matrix(pias) %*% as.matrix(chain(index[letters[1:3]]))
```

`aggregation_structure` *Make a price index aggregation structure*

Description

Create a price index aggregation structure from a hierarchical classification and aggregation weights that can be used to aggregate elementary indexes.

Usage

```
aggregation_structure(x, weights = NULL)
```

Arguments

x	[list] A list of character vectors that give the codes/labels for each level of the classification, ordered so that moving down the list goes down the hierarchy. The last vector gives the elementary aggregates, which should have no duplicates. All vectors should be the same length, without NAs, and there should be no duplicates across different levels of x. Names for x are used as level names; otherwise, levels are named level1, level2, ..., ea.
weights	[numeric >= 0] A numeric vector of aggregation weights for the elementary aggregates (i.e., the last vector in x), or something that can be coerced into one. The default is to give each elementary aggregate the same weight.

Value

A price index aggregation structure of class piar_aggregation_structure. This is a list-S3 class with the following components.

child	A nested list that gives the positions of the immediate children for each node in each level of the aggregation structure above the terminal nodes.
parent	A list that gives the position of the immediate parent for each node of the aggregation structure below the initial nodes.
levels	A named list of character vectors that give the levels of x.
weights	A vector giving the weight for each elementary aggregate.

Warning

The aggregation_structure() function does its best to check its arguments, but there should be no expectation that the result of aggregation_structure() will make any sense if x does not represent a nested hierarchy.

See Also

aggregate() to aggregate price indexes made with elementary_index().

expand_classification() to make x from a character representation of a hierarchical aggregation structure.

as_aggregation_structure() to coerce tabular data into an aggregation structure.

as.data.frame() and as.matrix() to coerce an aggregation structure into a tabular form.

weights() to get the weights for an aggregation structure.

update() for updating a price index aggregation structure with an aggregated index.

Examples

```
# A simple aggregation structure.
#            1
#      |-----+-----|
#      11          12
#  |---+---|       |
#  111     112     121
#  (1)     (3)     (4)
aggregation_weights <- data.frame(
  level1 = c("1", "1", "1"),
  level2 = c("11", "11", "12"),
  ea     = c("111", "112", "121"),
  weight = c(1, 3, 4)
)

aggregation_structure(
  aggregation_weights[1:3],
  weights = aggregation_weights[[4]]
)

# The aggregation structure can also be made by expanding the
# elementary aggregates.
with(
  aggregation_weights,
  aggregation_structure(expand_classification(ea), weight)
)
```

---

as.data.frame.piar_index

Coerce an index into a tabular form

Description

Turn an index into a data frame or a matrix.

Usage

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

## S3 method for class 'piar_index'
as.matrix(x, ...)
```

Arguments

x	[piar_index] A price index, as made by, e.g., elementary_index().
row.names	[character] See base::as.data.frame().
...	Not currently used for the matrix method. Extra arguments to base::as.data.frame.list() for the data frame method.
contrib	[logical(1)] Include percent-change contributions (the default does not include them).
stringsAsFactors, optional	[logical(1)] See base::as.data.frame().

Value

as.data.frame() returns the index values in x as a data frame with three columns: period, level, and value. If contrib = TRUE then there is a fourth (list) column contrib containing percent-change contributions.

as.matrix() returns the index values in x as a matrix with a row for each level and a column for each time period in x.

See Also

as_index() to coerce a matrix/data frame of index values into an index object.

Other index methods: [.piar_index(), aggregate.piar_index, as.ts.piar_index(), chain(), contrib(), head.piar_index(), is.na.piar_index(), lag.piar_index(), levels.piar_index(), mean.piar_index, merge.piar_index(), split.piar_index(), stack.piar_index(), time.piar_index(), window.piar_index()

Examples

```
index <- as_index(matrix(1:6, 2))

as.data.frame(index)

as.matrix(index)
```

---

as.matrix.piar_aggregation_structure

Coerce an aggregation structure into a tabular form

Description

Coerce a price index aggregation structure into an aggregation matrix or a data frame.

Usage

```
## S3 method for class 'piar_aggregation_structure'
as.matrix(x, ..., sparse = FALSE)

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

Arguments

x	[piar_agregation_structure] A price index aggregation structure, as made by aggregation_structure().
...	Not currently used for the matrix method. Extra arguments to base::as.data.frame.list() for the data frame method.
sparse	[logical(1)] Should the result be a sparse matrix from the **Matrix** package? This is faster for large aggregation structures. The default returns an ordinary dense matrix.
row.names	[character] See base::as.data.frame().
optional	[logical(1)] See base::as.data.frame().

Value

as.matrix() represents an aggregation structure as a matrix, such that multiplying with a (column) vector of elementary indexes gives the aggregated index.

as.data.frame() takes an aggregation structure and returns a data frame that could have generated it.

See Also

as_aggregation_structure() for coercing into an aggregation structure.

The treemap() function in the **treemap** package and as.Node() from the **data.tree** package for visualizing an aggregation structure.

Other aggregation structure methods: cut.piar_aggregation_structure(), levels.piar_aggregation_structure(), update.piar_aggregation_structure(), weights.piar_aggregation_structure()

Examples

```
# A simple aggregation structure.
#            1
#      |-----+-----|
#      11          12
#  |---+---|       |
#  111     112     121
#  (1)     (3)     (4)
aggregation_weights <- data.frame(
  level1 = c("1", "1", "1"),
  level2 = c("11", "11", "12"),
  ea     = c("111", "112", "121"),
  weight = c(1, 3, 4)
```

```
)

pias <- as_aggregation_structure(aggregation_weights)

as.matrix(pias)

all.equal(as.data.frame(pias), aggregation_weights)

## Not run:
# Visualize as a treemap.
treemap::treemap(
  aggregation_weights,
  index = names(aggregation_weights)[-4],
  vSize = "weight",
  title = "aggregation structure"
)

# Or turn into a more genereal tree object and plot.
aggregation_weights$pathString <- do.call(
  \(...) paste(..., sep = "/"),
  aggregation_weights[-4]
)
plot(data.tree::as.Node(aggregation_weights))

## End(Not run)
```

`as.ts.piar_index` *Coerce an index into a time series*

Description

Turn an index into a regular time series, represented as a `stats::ts` object.

Usage

```
## S3 method for class 'piar_index'
as.ts(x, ...)
```

Arguments

`x`	`[piar_index]` A price index, as made by, e.g., `elementary_index()`.
`...`	Additional arguments passed to `stats::ts()`.

Value

A time series object.

See Also

Other index methods: [.piar_index(), aggregate.piar_index, as.data.frame.piar_index(), chain(), contrib(), head.piar_index(), is.na.piar_index(), lag.piar_index(), levels.piar_index(), mean.piar_index, merge.piar_index(), split.piar_index(), stack.piar_index(), time.piar_index(), window.piar_index()

Examples

```
as.ts(as_index(matrix(1:9, 3)))
```

---

as_aggregation_structure

Coerce to an aggregation structure

Description

Coerce an object into an aggregation structure object.

Usage

```
as_aggregation_structure(x, ...)

## Default S3 method:
as_aggregation_structure(x, ..., weights = NULL)

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

## S3 method for class 'matrix'
as_aggregation_structure(x, ...)
```

Arguments

x	[object] An object to coerce into an aggregation structure.
...	Further arguments passed to or used by methods.
weights	[numeric >= 0] A numeric vector of aggregation weights for the elementary aggregates. The default is to give each elementary aggregate the same weight.

Details

The default method attempts to coerce x into a list prior to calling aggregation_structure().

The data frame and matrix methods treat x as a table with a row for each elementary aggregate, a column of labels for each level in the aggregation structure, and a column of weights for the elementary aggregates.

Value

A price index aggregation structure that inherits from piar_aggregation_structure.

See Also

as.matrix() and as.data.frame() for coercing an aggregation structure into a tabular form.

Examples

```
# A simple aggregation structure
#            1
#      |-----+-----|
#      11          12
#  |---+---|       |
#  111     112     121
#  (1)     (3)     (4)
aggregation_weights <- data.frame(
  level1 = c("1", "1", "1"),
  level2 = c("11", "11", "12"),
  ea     = c("111", "112", "121"),
  weight = c(1, 3, 4)
)

pias <- aggregation_structure(
  aggregation_weights[1:3],
  weights = aggregation_weights[[4]]
)

all.equal(
  pias,
  as_aggregation_structure(aggregation_weights)
)

all.equal(
  pias,
  as_aggregation_structure(as.matrix(aggregation_weights))
)
```

---

as_index *Coerce to a price index*

Description

Coerce pre-computed index values into an index object.

Usage

```
as_index(x, ...)

## Default S3 method:
as_index(x, ...)

## S3 method for class 'matrix'
as_index(x, ..., chainable = TRUE, contrib = FALSE)

## S3 method for class 'data.frame'
as_index(x, ..., contrib = FALSE)

## S3 method for class 'chainable_piar_index'
as_index(x, ..., chainable = TRUE)

## S3 method for class 'direct_piar_index'
as_index(x, ..., chainable = FALSE)

## S3 method for class 'mts'
as_index(x, ...)
```

Arguments

x	[object] An object to coerce into a price index.
...	Further arguments passed to or used by methods.
chainable	[logical(1)] Are the index values in x period-over-period indexes, suitable for a chained calculation (the default)? This should be FALSE when x contains fixed-base (direct) index values.
contrib	[logical(1)] Should the index values in x be used to construct percent-change contributions? The default does not make contributions.

Details

Numeric matrices are coerced into an index object by treating each column as a separate time period, and each row as a separate level of the index (e.g., an elementary aggregate). Column names are used to denote time periods, and row names are used to denote levels (so they must be unique). This essentially reverses calling `as.matrix()` on an index object. If a dimension is unnamed, then it is given a sequential label from 1 to the size of that dimension. The default and multiple time series methods coerces x to a matrix prior to using the matrix method.

The data frame method for `as_index()` is best understood as reversing the effect of `as.data.frame()` on an index object. It constructs a matrix by taking the levels of `x[[1]]` as columns and the levels of `x[[2]]` as rows (coercing to a factor if necessary). It then populates this matrix with the corresponding values in `x[[3]]`, and uses the matrix method for `as_index()`. If `contrib = TRUE` and there is a fourth list column of product contributions then these are also included in the resulting index.

If x is a period-over-period index then it is returned unchanged when `chainable = TRUE` and chained otherwise. Similarly, if x is a fixed-base index then it is returned unchanged when `chainable = FALSE` and unchain otherwise.

Value

A price index that inherits from `piar_index`. If `chainable = TRUE` then this is a period-over-period price index that also inherits from `chainable_piar_index`; otherwise, it is a fixed-base index that inherits from `direct_piar_index`.

See Also

`as.matrix()` and `as.data.frame()` for coercing an index into a tabular form.

Examples

```
prices <- data.frame(
  rel = 1:8,
  period = rep(1:2, each = 4),
  ea = rep(letters[1:2], 4)
)

index <- elementary_index(prices, rel ~ period + ea)

all.equal(as_index(as.data.frame(index)), index)
all.equal(as_index(as.matrix(index)), index)
```

---

back_period *Back period*

Description

Offset a vector prices or quantities by computing the position one (or more) period backwards for each product.

Usage

```
back_period(period, product = NULL, match_first = TRUE, offset = 1L)
```

Arguments

`period`	`[factor]` A factor, or something that can be coerced into one, that gives the time period for each transaction. The ordering of time periods follows the levels of `period` to agree with `base::cut.Date()`.
`product`	`[factor]` A factor, or something that can be coerced into one, that gives the product identifier for each transaction. The default is to assume that all transactions are for the same product.
`match_first`	`[logical(1)]` Should products in the first period match with themselves (the default)?
`offset`	`[integer(1) >= 0]` The number of periods to offset. The default offsets by one period (back period). Setting to `nlevels(period)` gives the the base period.

Value

A numeric vector of indices giving the position of the the back periods.

Note

By definition, there must be at most one transaction for each product in each time period to determine a back period. If multiple transactions correspond to a period-product pair, then the back period at a point in time is always the first position for that product in the previous period.

See Also

`outliers()` for common methods to detect outliers for price relatives.

`rs_pairs` in the **rsmatrix** package for making sales pairs.

Examples

```
prices <- data.frame(
  price = 1:6,
  product = factor(c("a", "b")),
  period = factor(c(1, 1, 2, 2, 3, 3))
)

with(prices, back_period(period, product))

# Make fixed-base price relatives.
with(
  prices,
  price / price[back_period(period, product, offset = nlevels(period))]
)

# Change the base period with relevel().
with(
  prices,
  price / price[
    back_period(relevel(period, "2"), product, offset = nlevels(period))
  ]
)
```

`chain` *Chain and rebase a price index*

Description

Chain a period-over-period index by taking the cumulative product of its values to turn it into a fixed-base (direct) index.

Unchain a fixed-base index by dividing its values for successive periods to get a period-over-period index.

Rebase a fixed-base index by dividing its values with the value of the index in the new base period.

Usage

```
chain(x, ...)

## Default S3 method:
chain(x, ...)

## S3 method for class 'chainable_piar_index'
chain(x, ..., link = NULL)

unchain(x, ...)

## Default S3 method:
unchain(x, ...)

## S3 method for class 'direct_piar_index'
unchain(x, ..., base = NULL)

rebase(x, ...)

## Default S3 method:
rebase(x, ...)

## S3 method for class 'direct_piar_index'
rebase(x, ..., base = NULL)
```

Arguments

x	[piar_index] A price index, as made by, e.g., elementary_index().
...	Further arguments passed to or used by methods.
link	[numeric > 0] A numeric vector, or something that can coerced into one, of link values for each level in x. The default is equivalent to a vector of 1s so that no linking is done.
base	[numeric > 0 | character(1)] A numeric vector, or something that can coerced into one, of base-period index values for each level in x. The default is a equivalent to a vector of 1s so that the base period remains the same. If base is a length-one character vector giving a time period of x then the index values for this time period are used as the base-period values.

Details

The default methods attempt to coerce x into an index with as_index() prior to chaining/unchaining/rebasing.

Chaining an index takes the cumulative product of the index values for each level; this is roughly the same as t(apply(as.matrix(x), 1, cumprod)) * link. Unchaining does the opposite, so these are inverse operations. Note that unchaining a period-over-period index does nothing, as does chaining a fixed-base index.

Rebasing a fixed-base index divides the values for each level of this index by the corresponding values for each level in the new base period. It's roughly the same as as.matrix(x) / base. Like unchaining, rebasing a period-over-period index does nothing.

Percent-change contributions are removed when chaining/unchaining/rebasing an index as it's not usually possible to update them correctly.

Value

`chain()` and `rebase()` return a fixed-base index that inherits from `direct_piar_index`.

`unchain()` returns a period-over-period index that inherits from `chainable_piar_index`.

See Also

Other index methods: `[.piar_index()`, `aggregate.piar_index`, `as.data.frame.piar_index()`, `as.ts.piar_index()`, `contrib()`, `head.piar_index()`, `is.na.piar_index()`, `lag.piar_index()`, `levels.piar_index()`, `mean.piar_index`, `merge.piar_index()`, `split.piar_index()`, `stack.piar_index()`, `time.piar_index()`, `window.piar_index()`

Examples

```
index <- as_index(matrix(1:9, 3))

# Make period 0 the fixed base period.
chain(index)

# Chaining and unchaining reverse each other.
all.equal(index, unchain(chain(index)))

# Change the base period to period 2 (note the
# loss of information for period 0).
index <- chain(index)
rebase(index, base = index[, 2])
```

`combine_classifications`

Combine hierarchical classifications

Description

Combine hierarchical classifications by stacking one classification after another.

Usage

```
combine_classifications(..., sep = ".")
```

Arguments

`...`	`[list]` A collection of lists, one for each classification, each giving the "digits" that represent each level in the hierarchy, as made by `expand_classification()` or `split_classification()`.
`sep`	`[character(1)]` A character used to separate the classifications in `....` The default separates levels across classifications by `"."`.

Value

A list with a entry for each level in the combined classification.

Examples

```
# Combine an unbalanced industry classification with a balanced
# geographic classification.
industry <- c("111", "112", "12")
region <- c("11", "21", "22")

combine_classifications(
  expand_classification(industry, pad = "0"),
  expand_classification(region)
)
```

`contrib` *Extract percent-change contributions*

Description

Extract a matrix or data frame of percent-change contributions from a price index.

Usage

```
contrib(x, level = NULL, period = NULL, pad = 0)

contrib2DF(x, levels = NULL, period = NULL)

contrib(x, level = NULL, period = NULL) <- value

set_contrib(x, level = NULL, period = NULL, value)

set_contrib_from_index(x)
```

Arguments

`x`	`[piar_index]` A price index, as made by, e.g., `elementary_index()`.
`level, levels`	`[character]` The level of an index for which percent-change contributions are desired, defaulting to the first level (usually the top-level for an aggregate index). `contrib2DF()` can accept multiple levels.
`period`	`[character]` The time periods for which percent-change contributions are desired, defaulting to all time periods.
`pad`	`[numeric(1)]` A numeric value to pad contributions so that they fit into a rectangular array when products differ over time. The default is 0.
`value`	`[matrix]` A numeric matrix of replacement contributions with a row for each product and a column for each time period. Recycling occurs along time periods.

Value

`contrib()` returns a matrix of percent-change contributions with a column for each `period` and a row for each product (sorted) for which there are contributions in `level`. Contributions are padded with `pad` to fit into a rectangular array when products differ over time. The replacement methods returns a copy of `x` with contributions given by the matrix `value`. (`set_contrib()` is an alias that's easier to use with pipes.) `set_contrib_from_index()` is a helper to return a copy of `x` with all contributions set to the corresponding index value minus 1.

`contrib2DF()` returns a data frame of contributions with four columns: `period`, `level`, `product`, and `value`.

See Also

Other index methods: `[.piar_index()`, `aggregate.piar_index`, `as.data.frame.piar_index()`, `as.ts.piar_index()`, `chain()`, `head.piar_index()`, `is.na.piar_index()`, `lag.piar_index()`, `levels.piar_index()`, `mean.piar_index`, `merge.piar_index()`, `split.piar_index()`, `stack.piar_index()`, `time.piar_index()`, `window.piar_index()`

Examples

```
prices <- data.frame(
  rel = 1:8,
  period = rep(1:2, each = 4),
  ea = rep(letters[1:2], 4)
)

index <- elementary_index(prices, rel ~ period + ea, contrib = TRUE)

pias <- aggregation_structure(
  list(c("top", "top", "top"), c("a", "b", "c")),
  weights = 1:3
)

index <- aggregate(index, pias, na.rm = TRUE)

# Percent-change contributions for the top-level index.
contrib(index)

contrib2DF(index)

# Calculate EA contributions for the chained index.
arithmetic_contributions <- function(x, w, order = 1) {
  (x - 1) * transmute_weights(x, w, order, to = 1)
}

arithmetic_contributions(
  as.matrix(chain(index))[c("a", "b", "c"), 2],
  weights(pias)
)
```

`cut.piar_aggregation_structure` *Cut an aggregation structure*

Description

Keep only the part of an aggregation structure above or below a certain level.

Usage

```
## S3 method for class 'piar_aggregation_structure'
cut(x, level, ..., na.rm = FALSE, upper = TRUE)
```

Arguments

`x`	`[piar_aggregation_structure]` A price index aggregation structure, as made by `aggregation_structure()`.
`level`	`[integer(1) > 0]` A positive integer, or something that can be coerced into one, giving the level at which to cut `x`.
`...`	Not currently used.
`na.rm`	`[logical(1)]` Should missing values be removed when aggregating the weights? By default, missing values are not removed.
`upper`	`[logical(1)]` Keep only the part of `x` above `level` (the default); otherwise, return the part of `x` below `level`.

Value

A price index aggregation structure of class `piar_aggregation_structure`.

See Also

Other aggregation structure methods: `as.matrix.piar_aggregation_structure()`, `levels.piar_aggregation_structu`
`update.piar_aggregation_structure()`, `weights.piar_aggregation_structure()`

Examples

```
# A simple aggregation structure.
#            1
#      |-----+-----|
#      11          12
#  |---+---|       |
#  111     112     121
#  (1)     (3)     (4)
aggregation_weights <- data.frame(
  level1 = c("1", "1", "1"),
  level2 = c("11", "11", "12"),
  ea     = c("111", "112", "121"),
  weight = c(1, 3, 4)
```

```
)

pias <- aggregation_structure(
  aggregation_weights[1:3],
  weights = aggregation_weights[[4]]
)

# Turn it into
#            1
#      |-----+-----|
#      11          12
#     (4)         (4)
cut(pias, 2)
```

`elementary_index` *Make elementary/elemental price indexes*

Description

Compute period-over-period (chainable) or fixed-base (direct) elementary price indexes, with optional percent-change contributions for each product.

Usage

```
elementary_index(x, ...)

## Default S3 method:
elementary_index(x, ...)

## S3 method for class 'numeric'
elementary_index(
  x,
  ...,
  period = NULL,
  ea = NULL,
  weights = NULL,
  product = NULL,
  chainable = TRUE,
  na.rm = FALSE,
  contrib = FALSE,
  order = 0,
  r = order
)

## S3 method for class 'data.frame'
elementary_index(x, formula, ..., weights = NULL, product = NULL)

elemental_index(x, ...)
```

Arguments

x	[object] Period-over-period or fixed-base price relatives. Currently there are methods for numeric vectors (which can be made with price_relative()) and data frames.
...	Further arguments passed to or used by methods.
period	[factor] A factor, or something that can be coerced into one, giving the time period associated with each price relative in x. The ordering of time periods follows of the levels of period, to agree with base::cut.Date(). The default makes an index for one time period.
ea	[factor] A factor, or something that can be coerced into one, giving the elementary aggregate associated with each price relative in x. The default makes an index for one elementary aggregate.
weights	[numeric >= 0] A numeric vector of weights for the price relatives in x, or something that can be coerced into one. The default is equal weights. This is evaluated in x for the data frame method.
product	[character] A character vector of product names, or something that can be coerced into one, for each price relative in x when making percent-change contributions. The default uses the names of x, if any; otherwise, elements of x are given sequential names within each elementary aggregate. This is evaluated in x for the data frame method.
chainable	[logical(1)] Are the price relatives in x period-over-period relatives that are suitable for a chained calculation (the default)? This should be FALSE when x contains fixed-base relatives.
na.rm	[logical(1)] Should missing values be removed? By default, missing values are not removed. Setting na.rm = TRUE is equivalent to overall-mean imputation.
contrib	[logical(1)] Should percent-change contributions be calculated? The default does not calculate contributions.
order	[numeric(1)] Order of the generalized mean to aggregate price relatives. 0 for a geometric index (the default for making elementary indexes), 1 for an arithmetic index (the default for aggregating elementary indexes and averaging indexes over subperiods), or -1 for a harmonic index (usually for a Paasche index). Other values are possible; see gmean() for details.
r	Deprecated.
formula	[formula] A two-sided formula, or something that can be coerced into one, with price relatives on the left-hand side and time periods and elementary aggregates (in that order) on the right-hand side.

Details

When supplied with a numeric vector, elementary_index() is a simple wrapper that applies gmean() and transmute_weights() (if contrib = TRUE) to x and weights grouped by ea and period. That is, for every combination of elementary aggregate and time period, elementary_index() calculates an index based on a generalized mean of order order and, optionally, percent-change contributions. Product names should be unique within each elementary aggregate at each time period when making contributions and, if not, are passed to base::make.unique() with a warning. The default (order = 0 and no weights) makes Jevons elementary indexes. See the CPI manual

(IMF, ILO, Eurostat, UNECE, OECD, and World Bank 2020, chapter 8, pp. 175–190) for more detail about making elementary indexes, or the PPI manual (ILO, IMF, OECD, UNECE, and World Bank 2004, chapter 9), and Balk (2008, chapter 5).

The default method simply coerces `x` to a numeric vector prior to calling the method above. The data frame method provides a formula interface to specify columns of price relatives, time periods, and elementary aggregates and call the method above.

The interpretation of the index depends on how the price relatives in `x` are made. If these are period-over-period relatives, then the result is a collection of period-over-period (chainable) elementary indexes; if these are fixed-base relatives, then the result is a collection of fixed-base (direct) elementary indexes. For the latter, `chainable` should be set to `FALSE` so that no subsequent methods assume that a chained calculation should be used.

By default, missing price relatives in `x` will propagate throughout the index calculation. Ignoring missing values with `na.rm = TRUE` is the same as overall-mean imputation, and needs to be explicitly set in the call to `elementary_index()`. Explicit imputation of missing relatives, and especially imputation of missing prices, should be done prior to calling `elementary_index()`.

Indexes based on nested generalized means, like the Fisher index (and superlative quadratic mean indexes more generally), can be calculated by supplying the appropriate weights with `transmute_weights2()`; see the example below. It is important to note that there are several ways to make these weights, and this affects how percent-change contributions are calculated.

`elemental_index()` is an alias for `elementary_index()`.

Value

A price index that inherits from `piar_index`. If `chainable = TRUE` then this is a period-over-period index that also inherits from `chainable_piar_index`; otherwise, it is a fixed-based index that inherits from `direct_piar_index`.

References

Balk BM (2008). *Price and Quantity Index Numbers*. Cambridge University Press. doi:10.1017/CBO9780511720758.

ILO, IMF, OECD, UNECE, World Bank (2004). *Producer price index manual: Theory and practice*. International Monetary Fund. doi:10.5089/9781589063044.069.

IMF, ILO, Eurostat, UNECE, OECD, World Bank (2020). *Consumer price index manual: Concepts and methods*. International Monetary Fund. doi:10.5089/9781484354841.069.

von der Lippe P (2007). *Index Theory and Price Statistics*. Peter Lang. doi:10.3726/9783653-011203.

See Also

`price_relative()` for making price relatives for the same products over time, and `impute_prices()` for imputation of missing prices.

`as_index()` to turn pre-computed (elementary) index values into an index object.

`chain()` for chaining period-over-period indexes, and `rebase()` for rebasing an index.

`aggregate()` to aggregate elementary indexes according to an aggregation structure.

`as.matrix()` and `as.data.frame()` for coercing an index into a tabular form.

Examples

```
prices <- data.frame(
  rel = 1:8,
  period = rep(1:2, each = 4),
  ea = rep(letters[1:2], 4)
)

# Calculate Jevons elementary indexes.
elementary_index(prices, rel ~ period + ea)

# Same as using lm() or tapply().
exp(coef(lm(log(rel) ~ ea:factor(period) - 1, prices)))

with(
  prices,
  t(tapply(rel, list(period, ea), \(x) gmean(x, order = 0, na.rm = TRUE)))
)

# Calculate a CSWD index (same as the Jevons in this example)
# as an arithmetic index by using the appropriate weights.
cswd_weights <- with(
  prices,
  lapply(split(rel, list(period, ea)), transmute_weights2)
)

elementary_index(
  prices,
  rel ~ period + ea,
  weights = unsplit(cswd_weights, interaction(period, ea)),
  order = 1
)
```

---

emean *Extended mean*

Description

Calculate the component-wise extended mean.

Usage

```
emean(x, y, order = c(0, 1), tol = .Machine$double.eps^0.5)
```

Arguments

x, y	[numeric > 0] A strictly positive numeric vector.
order	[numeric(2)] A pair of finite numbers giving the order of the extended mean. The default calculates the ordinary logarithmic mean. Setting either the first or second element to 1 gives the generalized logarithmic mean.

`tol` [numeric > 0] The tolerance used to determine if `x == y`. The default value is the same as `base::all.equal()`.

Details

The extended mean is also called the difference mean, Stolarsky mean, or extended mean-value mean; see Bullen (2003, p. 393) for details.

Both `x` and `y` should be strictly positive. This is not enforced, but the results may not make sense when the extended mean is not defined. The usual recycling rules apply when `x` and `y` are not the same length.

By definition, the extended mean of `x` and `y` is `x` when `x == y`. The `tol` argument is used to test equality by checking if `abs(x - y) <= tol`. In some cases it's useful to multiply `tol` by a scale factor, such as `max(abs(x), abs(y))`. This often doesn't matter when making price indexes, however, as `x` and `y` are usually around 1.

Value

A numeric vector, the same length as `max(length(x), length(y))`, giving the component-wise extended mean of `x` and `y`.

References

Bullen PS (2003). *Handbook of Means and their Inequalities*. Springer Science+Business Media. doi:10.1007/9789401703994.

See Also

Other math functions: `gmean()`, `nested_gmean()`, `scale_weights()`, `transmute_weights()`, `transmute_weights2()`, `update_weights()`

Examples

```
x <- 8:5
y <- 1:4

# The arithmetic and geometric means are special cases of the
# generalized logarithmic mean.
all.equal(emean(x, y, c(2, 1)), (x + y) / 2)
all.equal(emean(x, y, c(-1, 1)), sqrt(x * y))

# The harmonic mean cannot be expressed as a logarithmic mean, but can
# be expressed as an extended mean.
all.equal(emean(x, y, c(-2, -1)), 2 / (1 / x + 1 / y))

# The quadratic mean is also a type of extended mean.
all.equal(emean(x, y, c(2, 4)), sqrt(x^2 / 2 + y^2 / 2))

# As are heronian and centroidal means.
all.equal(
  emean(x, y, c(0.5, 1.5)),
```

```
  (x + sqrt(x * y) + y) / 3
)
all.equal(
  emean(x, y, c(2, 3)),
  2 / 3 * (x^2 + x * y + y^2) / (x + y)
)
```

`expand_classification` *Expand a hierarchical classification*

Description

Expand a character representation of a hierarchical classification to make a price index aggregation structure. Expanded classifications be interacted together to get all combinations of aggregation structures.

Usage

```
expand_classification(x, width = 1L, pad = NA)

interact_classifications(..., sep = ":")
```

Arguments

`x`	`[character]` A character vector, or something that can be coerced into one, of codes/labels for a specific level in a classification (e.g., 5-digit COICOP, 5-digit NAICS, 4-digit SIC).
`width`	`[integer > 0]` An integer vector that gives the width of each digit in `x`. A single value is recycled to span the longest element in `x`. This cannot contain NAs. The default assumes each digit has a width of 1, as in the NAICS, NAPCS, and SIC classifications.
`pad`	`[character(1)]` A string used to pad the shorter labels for an unbalanced classification. The default pads with NA.
`...`	`[list]` Lists of character vectors that give the codes/labels for each level of the classification, ordered so that moving down the list goes down the hierarchy (as made by `expand_classification()`).
`sep`	`[character(1)]` A character used to combine codes/labels across elements of `...`. The default uses `":"`.

Value

`expand_classification()` returns a list with a entry for each level in `x` giving the "digits" that represent each level in the hierarchy.

`interact_classfications()` returns a list of lists with the same structure as `expand_classification()`.

See Also

aggregation_structure() to make a price-index aggregation structure.

split_classification() to expand a classification by splitting along a delimiter.

combine_classifications() for combining multiple hierarchical classifications.

csh_from_digits() in the **accumulate** package for different handling of unbalanced classifications.

Examples

```
# A simple classification structure.
#            1
#      |-----+-----|
#      11          12
#  |---+---|       |
#  111     112     121

expand_classification(c("111", "112", "121"))

# Expanding more complex classifications:
# ... if last 'digit' is either TA or TS.
expand_classification(
  c("111TA", "112TA", "121TS"),
  width = c(1, 1, 1, 2)
)

# ... if first 'digit' is either 11 or 12.
expand_classification(c("111", "112", "121"), width = c(2, 1))

# ...if there are delimiters in the classification (like COICOP).
expand_classification(c("01.1.1", "01.1.2", "01.2.1"), width = 2)
```

---

geks_index *GEKS index*

Description

Calculate a generalized inter-temporal GEKS price index over a rolling window.

Usage

```
geks_index(
  price,
  quantity,
  period,
  product,
  index_formula = function(p1, p0, q1, q0) {
     nested_gmean(p1/p0, list(p0 * q0, p1 *
```

```
    q1), na.rm = TRUE)
 },
  window = nlevels(period),
  n = window - 1L,
  order = 0,
  match_method = c("all", "back-price")
)
```

Arguments

price	[numeric > 0] A numeric vector of prices, the same length as quantity.
quantity	[numeric >= 0] A numeric vector of quantities, the same length as price.
period	[factor] A factor, or something that can be coerced into one, that gives the corresponding time period for each element in price and quantity. The ordering of time periods follows the levels of period to agree with base::cut.Date().
product	[factor] A factor, or something that can be coerced into one, that gives the corresponding product identifier for each element in price and quantity.
index_formula	[function] A function giving the index-number formula in the GEKS index. Usually a Törnqvist, Fisher (the default), or Walsh index. It must have arguments p1, p0, q1, and q0, and satisfy the time-reversal test. See vignette("index-number-formulas") for details.
window	[integer(1) > 0] A positive integer giving the length of the rolling window. The default is a window that encompasses all periods in period. Non-integers are truncated towards zero.
n	[integer(1) > 0] A positive integer giving the length of the index series for each window, starting from the end of the window. For example, if there are 13 periods in window, setting n = 1 gives the index for period 13. The default gives an index for each period in window. Non-integers are truncated towards zero.
order	[numeric(1)] A finite number giving the order of the generalized mean used to average price indexes over the rolling window. The default uses a geometric mean.
match_method	[character(1)] Either "all" to match all products against each other (the default) or "back-price" to match only back prices. The later can be faster when there is lots of product imbalanced.

Value

A list with a named numeric vector giving the value of the respective period-over-period GEKS index for each window.

Note

Like back_period(), if multiple prices correspond to a period-product pair, then the back price at a point in time is always the first price for that product in the previous period. Unlike a bilateral index, however, duplicated period-product pairs can have more subtle implications for a multilateral index.

References

Balk BM (2008). *Price and Quantity Index Numbers*. Cambridge University Press. doi:10.1017/CBO9780511720758.

IMF, ILO, Eurostat, UNECE, OECD, World Bank (2020). *Consumer price index manual: Concepts and methods*. International Monetary Fund. doi:10.5089/9781484354841.069.

Ivanic I, Diewert WE, Fox KJ (2011). "Scanner data, time aggregation and the construction of price indexes." *Journal of Econometrics*, **161**(1), 24–35. doi:10.1016/j.jeconom.2010.09.003.

See Also

`splice_index()` to splice the rolling-window indexes together.

`GEKSIndex()` in the **IndexNumR** package for an implementation of the GEKS index with more options.

The **rsmatrix** package for multilateral repeat-sales indexes.

Examples

```
price <- 1:10
quantity <- 10:1
period <- rep(1:5, 2)
product <- rep(letters[1:2], each = 5)

cumprod(geks_index(price, quantity, period, product)[[1]])

# Calculate the index over a rolling window.
(geks <- geks_index(price, quantity, period, product, window = 3))

# Use a movement splice to combine the indexes in each window.
splice_index(geks, 2)

# ... or use a mean splice.
splice_index(geks)

# Make a Jevons GEKS index.
geks_index(
  price,
  quantity,
  period,
  product,
  index_formula = \(p1, p0, ...) gmean(p1 / p0, na.rm = TRUE, order = 0)
)
```

`gmean` *Generalized mean*

Description

Calculated a weighted generalized mean.

Usage

```
gmean(x, weights = NULL, order = 1, na.rm = FALSE)
```

Arguments

x	[numeric > 0] A strictly positive numeric vector.
weights	[numeric >= 0] A positive numeric vector of weights, the same length as x. The default is to equally weight each element of x.
order	[numeric(1)] A finite number giving the order (or exponent) of the generalized mean. The default calculates an arithmetic mean.
na.rm	[logical(1)] Should missing values be removed? By default, missing values are not removed.

Details

The generalized mean is also called the power mean, Hölder mean, or l_p mean; see Bullen (2003, p. 175) for details.

Both x and weights are usually strictly positive (and finite), especially for the purpose of making a price index. This is not enforced, but the results may not make sense if the generalized mean is not defined. There are two exceptions to this.

1. The convention by Hardy, Littlewood, and Pólya (1952, p. 13) is used in cases where x has zeros: the generalized mean is 0 whenever the weights are strictly positive and order < 0. The analogous convention holds whenever at least one element of x is Inf: the generalized mean is Inf whenever the weights are strictly positive and order > 0.
2. Some authors let the weights be non-negative and sum to 1. If there are zero weights then the corresponding element of x has no impact on the result whenever x is strictly positive. Unlike stats::weighted.mean(), however, zero weights are not strong zeros, so infinite values in x will propagate.

The weights are scaled to sum to 1 to satisfy the definition of a generalized mean.

Value

A numeric value for the generalized mean.

Note

The generalized mean can be defined on the extended real line, so that order = -Inf / Inf returns base::min()/base::max(), to agree with the definition by Bullen (2003). This is not implemented, and the order of the generalized mean must be finite.

References

Bullen PS (2003). *Handbook of Means and their Inequalities*. Springer Science+Business Media. doi:10.1007/9789401703994.

Hardy G, Littlewood JE, Pólya G (1952). *Inequalities*, 2nd edition. Cambridge University Press.

See Also

Other math functions: emean(), nested_gmean(), scale_weights(), transmute_weights(), transmute_weights2(), update_weights()

Examples

```
x <- 1:3
w <- c(0.25, 0.25, 0.5)

# Arithmetic mean.
gmean(x, w)

# Geometric mean.
gmean(x, w, order = 0)

# The Lehmer mean is a generalized mean with specific weights.
gmean(x, w * x)
```

---

head.piar_index *Return the first/last parts of an index*

Description

Extract the first/last parts of an index as if it were a matrix.

Usage

```
## S3 method for class 'piar_index'
head(x, n = 6L, ...)

## S3 method for class 'piar_index'
tail(x, n = 6L, ...)
```

Arguments

x	[piar_index] A price index, as made by, e.g., elementary_index().
n	[integer] See utils::head()/utils::tail(). The default takes the first/last 6 levels of x.
...	Not currently used.

Value

A price index that inherits from the same class as x.

See Also

Other index methods: [.piar_index(), aggregate.piar_index, as.data.frame.piar_index(), as.ts.piar_index(), chain(), contrib(), is.na.piar_index(), lag.piar_index(), levels.piar_index(), mean.piar_index, merge.piar_index(), split.piar_index(), stack.piar_index(), time.piar_index(), window.piar_index()

Examples

```
index <- as_index(matrix(1:9, 3))

head(index, 1)

tail(index, 1)
```

---

impute_prices *Impute missing prices*

Description

Impute missing prices using the carry forward/backward or the self-correcting overall mean method.

Usage

```
impute_prices(x, ...)

## Default S3 method:
impute_prices(x, ...)

## S3 method for class 'matrix'
impute_prices(
  x,
  period,
  product,
  ...,
  ea = NULL,
  weights = NULL,
  pias = NULL,
  order = c(0, 1),
  r = order,
  method = c("overall-mean", "carry-forward"),
  impute_rules = NULL
)

## S3 method for class 'numeric'
impute_prices(
  x,
  period,
```

```
  product,
  ...,
  ea = NULL,
  weights = NULL,
  pias = NULL,
  order = c(0, 1),
  r = order,
  method = c("overall-mean", "carry-forward", "carry-backward"),
  impute_rules = NULL
)

## S3 method for class 'data.frame'
impute_prices(x, formula, ..., ea = NULL, weights = NULL)

carry_forward(x, ...)

carry_backward(x, ...)

shadow_price(x, ...)
```

Arguments

x	[object] Either a numeric vector (or something that can be coerced into one), a data frame of prices, or a two-column matrix of current prices and back prices (in that order).
...	Further arguments passed to or used by methods.
period	[factor] A factor, or something that can be coerced into one, giving the time period associated with each price in x. The ordering of time periods follows of the levels of period, to agree with base::cut.Date().
product	[factor] A factor, or something that can be coerced into one, giving the product associated with each price in x.
ea	[factor] A factor, or something that can be coerced into one, giving the elementary aggregate associated with each price in x. This is evaluated in x for the data frame method. The default pools all data into one elementary aggregate.
weights	[numeric >= 0] A numeric vector of weights for the prices in x (i.e., product weights), or something that can be coerced into one. The default is to give each price equal weight. This is evaluated in x for the data frame method.
pias	[piar_aggregation_structure] A price index aggregation structure, or something that can be coerced into one, as made with aggregation_structure(). The default imputes from elementary indexes only (i.e., not recursively).
order	[numeric(2)] A pair of numeric values. The first gives the order of the generalized-mean price index used to calculate the elementary price indexes, defaulting to a geometric index. The second gives the order of the generalized-mean price index used to aggregate the elementary price indexes, defaulting to an arithmetic index. Other values are possible; see gmean() for details.
r	Deprecated.
method	[character(1)] Name of the imputation method, one of "overall-mean" (the default), "carry-forward", or "carry-backward".

`impute_rules` [function] (Experimental) A function that applies imputation rules to the elementary indexes in each time period prior to aggregation. It takes two arguments, the elementary indexes for a given time period and the (price updated) aggregation structure, and must return back the elementary indexes.

`formula` [formula] A two-sided formula, or something that can be coerced into one, with prices on the left-hand side and time periods and products on the right-hand side (in that order).

Details

The carry forward method replaces a missing price for a product by the price for the same product in the previous period. It tends to push an index value towards 1, and is usually avoided; see the CPI manual (IMF, ILO, Eurostat, UNECE, OECD, and World Bank 2020, paragraph 6.61). The carry backwards method does the opposite, but this is rarely used in practice.

The self-correcting overall mean method recursively imputes a missing price by the value of the price for the same product in the previous period multiplied by the value of the period-over-period elementary index for the elementary aggregate to which that product belongs. This requires computing and aggregating an index (according to `pias`, unless `pias` is not supplied) for each `period`. The index values used to do the imputations are not returned because the index needs to be recalculated to get correct percent-change contributions. It is identical to simply excluding missing price relatives in the index calculation, except in the period that a missing product returns. For this reason care is needed when using this method. It is sensitive to the assumption that a product does not change over time, and in some cases it is safer to simply omit the missing price relatives instead of imputing the missing prices.

Imputation works slightly differently depending on whether data are in a long or wide format. When `x` is a two-column matrix of current and back prices (in that order), then imputation is done separately on the current price at a point in time and the back price at the next point in time. When `x` is a numeric vector then these two prices are necessarily the same.

Value

A numeric vector or matrix of prices with missing values replaced (where possible).

References

IMF, ILO, Eurostat, UNECE, OECD, World Bank (2020). *Consumer price index manual: Concepts and methods*. International Monetary Fund. doi:10.5089/9781484354841.069.

See Also

`price_relative()` for making price relatives for the same products over time.

Examples

```
prices <- data.frame(
  price = c(1:7, NA),
  period = rep(1:2, each = 4),
  product = 1:4,
  ea = rep(letters[1:2], 4)
```

```
)

impute_prices(prices, price ~ period + product, method = "carry-forward")

impute_prices(
  prices,
  price ~ period + product,
  ea = ea,
  method = "overall-mean"
)

# Can also be done with current price-back price formulation.
prices$back_price <- with(
  prices,
  price[back_period(period, product)]
)

impute_prices(
  prices,
  cbind(price, back_price) ~ period + product,
  ea = ea,
  method = "overall-mean"
)
```

`is.na.piar_index` *Missing values in a price index*

Description

Identify missing values in a price index.

Usage

```
## S3 method for class 'piar_index'
is.na(x)

## S3 method for class 'piar_index'
anyNA(x, recursive = FALSE)
```

Arguments

`x`	`[piar_index]` A price index, as made by, e.g., `elementary_index()`.
`recursive`	`[logical(1)]` Check if `x` also has missing percent-change contributions. By default only index values are checked for missingness.

Value

`is.na()` returns a logical matrix, with a row for each level of `x` and a columns for each time period, that indicates which index values are missing.

anyNA() returns TRUE if any index values are missing, or percent-change contributions (if recursive = TRUE).

See Also

Other index methods: [.piar_index(), aggregate.piar_index, as.data.frame.piar_index(), as.ts.piar_index(), chain(), contrib(), head.piar_index(), lag.piar_index(), levels.piar_index(), mean.piar_index, merge.piar_index(), split.piar_index(), stack.piar_index(), time.piar_index(), window.piar_index()

Examples

```
index <- as_index(matrix(c(1, 2, 3, NA, 5, NA), 2))

anyNA(index)
is.na(index)

# Carry forward imputation.
index[is.na(index)] <- 1
index
```

---

is_aggregation_structure

Test if an object is an aggregation structure

Description

Test if an object is a price index aggregation structure.

Usage

```
is_aggregation_structure(x)
```

Arguments

x	[object] An object to test.

Value

Returns TRUE if x inherits from piar_aggregation_structure.

is_index *Test if an object is a price index*

Description

Test if an object is a index object or a subclass of an index object.

Usage

```
is_index(x)

is_chainable_index(x)

is_direct_index(x)
```

Arguments

x	[object] An object to test.

Value

`is_index()` returns TRUE if x inherits from `piar_index`.

`is_chainable_index()` returns TRUE if x inherits from `chainable_piar_index`.

`is_direct_index()` returns TRUE if x inherits from `direct_piar_index`.

lag.piar_index *Lag a price index*

Description

Compute a lagged version of a price index by shifting the time base back a given number of time periods.

Usage

```
## S3 method for class 'piar_index'
lag(x, k = 1L, ...)
```

Arguments

x	[piar_index] A price index, as made by, e.g., `elementary_index()`.
k	[integer(1)] An integer, or something that can be coerced into one, giving the number of lags.
...	Not currently used.

Value

A lagged version of x.

See Also

Other index methods: [.piar_index(), aggregate.piar_index, as.data.frame.piar_index(), as.ts.piar_index(), chain(), contrib(), head.piar_index(), is.na.piar_index(), levels.piar_index(), mean.piar_index, merge.piar_index(), split.piar_index(), stack.piar_index(), time.piar_index(), window.piar_index()

Examples

```
x <- as_index(matrix(1:9, 3))
lag(x)
```

---

levels.piar_aggregation_structure

Get the levels for an aggregation structure

Description

Get the hierarchical list of levels for an aggregation structure. It is an error to try and replace these values.

Usage

```
## S3 method for class 'piar_aggregation_structure'
levels(x)
```

Arguments

x	[piar_aggregation_structure] A price index aggregation structure, as made by aggregation_structure().

Value

A list of character vectors giving the levels for each position in the aggregation structure.

See Also

Other aggregation structure methods: as.matrix.piar_aggregation_structure(), cut.piar_aggregation_structure update.piar_aggregation_structure(), weights.piar_aggregation_structure()

levels.piar_index *Get the levels for a price index*

Description

Methods to get and set the levels for a price index.

Usage

```
## S3 method for class 'piar_index'
levels(x)

## S3 replacement method for class 'piar_index'
levels(x) <- value

set_levels(x, value)
```

Arguments

x	[piar_index] A price index, as made by, e.g., elementary_index().
value	[character] A character vector, or something that can be coerced into one, giving the replacement levels for x.

Value

levels() returns a character vector with the levels for a price index.

The replacement method returns a copy of x with the levels in value. (set_levels() is an alias that's easier to use with pipes.)

See Also

Other index methods: [.piar_index(), aggregate.piar_index, as.data.frame.piar_index(), as.ts.piar_index(), chain(), contrib(), head.piar_index(), is.na.piar_index(), lag.piar_index(), mean.piar_index, merge.piar_index(), split.piar_index(), stack.piar_index(), time.piar_index(), window.piar_index()

mean.piar_index *Aggregate a price index over subperiods*

Description

Aggregate an index over subperiods by taking the (usually arithmetic) mean of index values over consecutive windows of subperiods.

Usage

```
## S3 method for class 'chainable_piar_index'
mean(
  x,
  ...,
  weights = NULL,
  window = NULL,
  na.rm = FALSE,
  contrib = TRUE,
  order = 1,
  r = order,
  duplicate_contrib = c("sum", "make.unique")
)

## S3 method for class 'direct_piar_index'
mean(
  x,
  ...,
  weights = NULL,
  window = NULL,
  na.rm = FALSE,
  contrib = TRUE,
  order = 1,
  r = order,
  duplicate_contrib = c("sum", "make.unique")
)
```

Arguments

x	[piar_index] A price index, as made by, e.g., elementary_index().
...	Not currently used.
weights	[numeric >= 0] A numeric vector of weights for the index values in x, or something that can be coerced into one. The default is equal weights. It is usually easiest to specify these weights as a matrix with a row for each index value in x and a column for each time period.
window	[integer(1) > 0] A positive integer giving the size of the window used to average index values across subperiods. The default averages over all periods in x. Non-integers are truncated towards 0.
na.rm	[logical(1)] Should missing values be removed? By default, missing values are not removed. Setting na.rm = TRUE is equivalent to overall-mean imputation.
contrib	[logical(1)] Aggregate percent-change contributions in x? By default contributions are aggregated.
order	[numeric(1)] Order of the generalized mean to aggregate index values. 0 for a geometric index (the default for making elementary indexes), 1 for an arithmetic index (the default for aggregating elementary indexes and averaging indexes over subperiods), or -1 for a harmonic index (usually for a Paasche index). Other values are possible; see gmean() for details.

`r` Deprecated.

`duplicate_contrib`
: `[character(1)]` The method to deal with duplicate product contributions. Either `"make.unique"` to make duplicate product names unique with `base::make.unique()` or `"sum"` to add contributions for the same products across subperiods (the default).

Details

The `mean()` method constructs a set of non-overlapping windows of length `window`, starting in the first period of the index, and takes the mean of each index value in these windows for each level of the index. The last window is discarded if it is incomplete (with a warning), so that index values are always averaged over `window` periods. The names for the first time period in each window form the new names for the aggregated time periods.

Percent-change contributions are aggregated if `contrib = TRUE` following the same approach as `aggregate()`.

An optional vector of weights can be specified when aggregating index values over subperiods, which is often useful when aggregating a Paasche index; see Balk (2008, section 4.3) for details.

Value

A price index, averaged over subperiods, that inherits from the same class as `x`.

References

Balk BM (2008). *Price and Quantity Index Numbers*. Cambridge University Press. doi:10.1017/CBO9780511720758.

See Also

Other index methods: `[.piar_index()`, `aggregate.piar_index`, `as.data.frame.piar_index()`, `as.ts.piar_index()`, `chain()`, `contrib()`, `head.piar_index()`, `is.na.piar_index()`, `lag.piar_index()`, `levels.piar_index()`, `merge.piar_index()`, `split.piar_index()`, `stack.piar_index()`, `time.piar_index()`, `window.piar_index()`

Examples

```
index <- as_index(matrix(c(1:12, 12:1), 2, byrow = TRUE), chainable = FALSE)

# Turn a monthly index into a quarterly index.
mean(index, window = 3)
```

`merge.piar_index` *Merge price indexes*

Description

Combine two price indexes with common time periods, merging together the index values and percent-change contributions for each time period.

This is useful for building up an index when different elementary aggregates come from different sources of data, or use different index-number formulas.

Usage

```
## S3 method for class 'chainable_piar_index'
merge(x, y, ...)

## S3 method for class 'direct_piar_index'
merge(x, y, ...)
```

Arguments

`x`	`[piar_index]` A price index, as made by, e.g., `elementary_index()`.
`y`	`[piar_index]` A price index, or something that can coerced into one. If `x` is a period-over-period index then `y` is coerced into a chainable index; otherwise, `y` is coerced into a direct index.
`...`	Not currently used.

Value

A combined price index that inherits from the same class as `x`.

See Also

Other index methods: `[.piar_index()`, `aggregate.piar_index`, `as.data.frame.piar_index()`, `as.ts.piar_index()`, `chain()`, `contrib()`, `head.piar_index()`, `is.na.piar_index()`, `lag.piar_index()`, `levels.piar_index()`, `mean.piar_index`, `split.piar_index()`, `stack.piar_index()`, `time.piar_index()`, `window.piar_index()`

Examples

```
index1 <- as_index(matrix(1:6, 2))

index2 <- index1
levels(index2) <- 3:4

merge(index1, index2)
```

nested_gmean *Nested generalized means*

Description

Calculate a weighted (outer) generalized mean of two (inner) generalized means (i.e., crossing means).

Usage

```
nested_gmean(
  x,
  weights = list(NULL, NULL),
  order = c(1, -1),
  outer_weights = NULL,
  outer_order = 0,
  na.rm = FALSE
)
```

Arguments

x	[numeric > 0] A strictly positive numeric vector.
weights	[list] A list of positive numeric vector of weights, each the same length as x, for both of the inner generalized means. NULL elements of weights equally weight each element of x. The default uses equal weights for both inner generalized mean.
order	[numeric(2)] A finite numeric vector giving the order of each of the inner generalized means. The default computes an arithmetic mean and a harmonic mean.
outer_weights	[numeric(2)] A strictly positive numeric vector weights for each of the inner generalized means as used in the outer generalized mean. The default weights each inner generalized mean equally.
outer_order	[numeric(1)] A finite number giving the order of the outer generalized mean. The default uses a geometric mean.
na.rm	[logical(1)] Should missing values in x and weights be removed? By default missing values are not removed. Note that removal of missing values is balanced across x and both elements of weights.

Value

A numeric value for the nested generalized mean.

See Also

Other math functions: emean(), gmean(), scale_weights(), transmute_weights(), transmute_weights2(), update_weights()

Examples

```
x <- 1:3
w1 <- c(0.25, 0.25, 0.5)
w2 <- c(0.3, 0.3, 0.4)
# Calculate the geometric mean of the arithmetic and harmonic means
# to make a Fisher index.
nested_gmean(x, list(w1, w2))
```

`outliers` *Outlier detection for price relatives*

Description

Standard cutoff-based methods for detecting outliers with price relatives.

Usage

```
outliers(
  x,
  upper,
  lower = upper,
  method = c("quartile", "resistant-fences", "kimber", "robust-z", "tukey"),
  scale = 0,
  quantile_type = 7
)

hb_transform(x)
```

Arguments

`x`	`[numeric > 0]` A numeric vector, usually of price relatives. These can be made with, e.g., `back_period()`.
`upper, lower`	`[numeric(1) > 0]` A number giving the upper and lower cutoffs for each element of `x`. By default the lower cutoff is the same as the upper cutoff
`method`	`[character(1)]` The outlier detection method, one of `"quartile"` (the default), `"resistant-fences"`, `"kimber"`, `"robust-z"`, or `"tukey"`.
`scale`	`[0 <= numeric(1) <= 1]` A number between 0 and 1 giving the scale factor for the median to establish the minimum dispersion between quartiles for each element of `x`. The default does not set a minimum dispersion.
`quantile_type`	`[integer(1)]` See `stats::quantile()`.

Details

This function constructs an interval of the form $[b_l(x) - c_l \times l(x), b_u(x) + c_u \times u(x)]$ and assigns a value in `x` as `TRUE` if that value does not belong to the interval, `FALSE` otherwise. The different methods differ in how they construct the values $b_l(x)$, $b_u(x)$, $l(x)$, and $u(x)$. Any missing values in `x` are ignored when calculating the interval, but will return `NA`.

The quartile method and Tukey algorithm are described in the CPI manual (IMF, ILO, Eurostat, UNECE, OECD, and World Bank 2020, paragraphs 5.113–5.135). The resistant fences method is an alternative to the quartile method, and is described by Rais (2008) and Hutton (2008). The Kimber method is yet another alternative. Quantile-based methods often identify price relatives as outliers because the distribution is concentrated around 1; setting `scale > 0` puts a floor on the minimum dispersion between quantiles as a fraction of the median. See the references for more details.

	$b_l(x)$	$b_u(x)$	$l(x)$	$u(x)$
Quartile	$Q_2(x)$	$Q_2(x)$	$Q_2(x) - Q_1(x)$	$Q_3(x) - Q_2(x)$
Resistant fences	$Q_1(x)$	$Q_3(x)$	$Q_3(x) - Q_1(x)$	$Q_3(x) - Q_1(x)$
Kimber	$Q_1(x)$	$Q_3(x)$	$Q_2(x) - Q_1(x)$	$Q_3(x) - Q_2(x)$

The robust Z-score is the usual method to identify relatives in the (asymmetric) tails of the distribution, simply replacing the mean with the median, and the standard deviation with the median absolute deviation.

These methods often assume that price relatives are symmetrically distributed (if not Gaussian). As the distribution of price relatives often has a long right tail, the natural logarithm can be used to transform price relative before identifying outliers (sometimes under the assumption that price relatives are distributed log-normal). The Hidiroglou-Berthelot transformation is another approach, described in the CPI manual (IMF et al. 2020, paragraph 5.124). (Sometimes the transformed price relatives are multiplied by $\max(p_1, p_0)^u$, for some $0 \leq u \leq 1$, so that products with a larger price get flagged as outliers (paragraph 5.128).)

Value

A logical vector, the same length as `x`, that is `TRUE` if the corresponding element of `x` is identified as an outlier, `FALSE` otherwise.

References

Hutton H (2008). "Dynamic outlier detection in price index surveys." In *Proceedings of the Survey Methods Section: Statistical Society of Canada Annual Meeting*. https://ssc.ca/sites/default/files/survey/documents/SSC2008_H_Hutton.pdf.

IMF, ILO, Eurostat, UNECE, OECD, World Bank (2020). *Consumer price index manual: Concepts and methods*. International Monetary Fund. doi:10.5089/9781484354841.069.

Rais S (2008). "Outlier detection for the Consumer Price Index." In *Proceedings of the Survey Methods Section: Statistical Society of Canada Annual Meeting*. https://ssc.ca/sites/default/files/survey/documents/SSC2008_S_Rais.pdf.

See Also

`back_period()` for a simple utility function to turn prices in a table into price relatives.

The `HBmethod()` function in the **univOutl** package for the Hidiroglou-Berthelot method for identifying outliers.

Examples

```
x <- c(1, 10, 15, 100)

outliers(x, upper = 2.5, method = "quartile")

# Always identifies fewer outliers than above.
outliers(x, upper = 2.5, method = "resistant-fences")
```

`piar_index` *Price index objects*

Description

There are several classes to represent price indexes.

- All indexes inherit from the `piar_index` virtual class.
- Period-over-period indexes that can be chained over time inherit from `chainable_piar_index`.
- Fixed-base indexes inherit from `direct_piar_index`.

Details

The `piar_index` object is a list-S3 class with the following components:

index A matrix of index values with a column for each period in `time` and a row for each level in `levels`.

contrib A list-matrix containing named vectors that give the percent-change contributions for each price relative with a column for each time period in `time` and a row for each level in `levels`, or `NULL`.

levels A character vector giving the levels of the index.

time A character vector giving the time periods for the index.

The `chainable_piar_index` and `direct_piar_index` subclasses have the same structure as the `piar_index` class, but differ in the methods used to manipulate the indexes.

`price_data` *Price data*

Description

Sample price and weight data for both a match sample and fixed sample type index.

price_relative *Calculate period-over-period price relatives*

Description

Construct period-over-period price relatives from information on prices and products over time.

Usage

```
price_relative(x, ...)

## Default S3 method:
price_relative(x, period, product, ...)

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

Arguments

x	[object] Either a numeric vector (or something that can be coerced into one) or data frame of prices.
...	Further arguments passed to or used by methods.
period	[factor] A factor, or something that can be coerced into one, that gives the corresponding time period for each element in x. The ordering of time periods follows the levels of period to agree with base::cut.Date().
product	[factor] A factor, or something that can be coerced into one, that gives the corresponding product identifier for each element in x.
formula	[formula] A two-sided formula, or something that can be coerced into one, with prices on the left-hand side and time periods and products (in that order) on the right-hand side.

Value

A numeric vector of price relatives, with product as names.

See Also

back_period() to get only the back price or base price.

impute_prices() to impute missing prices.

outliers() for methods to identify outliers with price relatives.

Examples

```
price_relative(
  1:6,
  period = rep(1:2, each = 3),
  product = rep(letters[1:3], 2)
)
```

scale_weights *Scale weights*

Description

Scale a vector of weights so that they sum to 1.

Usage

```
scale_weights(x)
```

Arguments

x	[numeric >= 0] A positive numeric vector with at least one non-zero element.

Value

A numeric vector that sums to 1. If there are NAs in x then the result sums 1 to if these values are removed.

See Also

Other math functions: emean(), gmean(), nested_gmean(), transmute_weights(), transmute_weights2(), update_weights()

Examples

```
scale_weights(1:5)

scale_weights(c(1:5, NA))
```

splice_index *Splice an index series*

Description

Splice a collection of index series computed over a rolling window into one index series. Splicing on multiple points combines the results with a geometric mean.

Usage

```
splice_index(x, periods = NULL, initial = NULL, published = FALSE)
```

Arguments

x	[list] A list of equal-length numeric vectors giving the period-over-period indexes for each window.
periods	[integer] An integer vector giving the splice points for each window. The default splices on each point in the window.
initial	[numeric] A numeric vector giving an initial period-over-period index series onto which the elements of x are spliced. The default uses the first element of x.
published	[logical(1)] Should the splice be done against the published series? The default splices using the recalculated index series.

Value

A numeric vector giving the spliced (fixed-base) index series.

References

Chessa AG (2019). "A Comparison of Index Extension Methods for Multilateral Methods." In *Proceedings of the 16th Meeting of the Ottawa Group on Price Indices*. https://stats.unece.org/ottawagroup/download/f540.pdf.

Krsinich F (2016). "The FEWS index: Fixed effects with a window splice." *Journal of Official Statistics*, **32**(2), 375–404. doi:10.1515/jos20160021.

Examples

```
# Make an index series over a rolling window.
x <- list(c(1.1, 0.9, 1.2), c(0.8, 1.3, 1.4), c(1.3, 1.3, 0.8))

# Mean splice.
splice_index(x)

# Movement splice.
splice_index(x, 3)

# Window splice.
splice_index(x, 1)

# Splicing on the published series preserves the within-window
# movement of the index series.
splice_index(x, 1, published = TRUE)
```

---

split.piar_index *Split an index into groups*

Description

Split an index into groups of indexes according to a factor, along either the levels or time periods of the index.

Usage

```
## S3 method for class 'piar_index'
split(x, f, drop = FALSE, ..., along = c("levels", "time"))

## S3 replacement method for class 'piar_index'
split(x, f, drop = FALSE, ..., along = c("levels", "time")) <- value
```

Arguments

x	[piar_index] A price index, as made by, e.g., elementary_index().
f	[factor | list] A factor or list of factors to group elements of x.
drop	[logical(1)] Should levels that do not occur in f be dropped? By default all levels are kept.
...	Further arguments passed to base::split.default().
along	[character(1)] Either "levels" to split over the levels of x (the default), or "time" to split over the time periods of x.
value	[list] A list of values compatible with the splitting of x, or something that can be coerced into one, recycled if necessary.

Value

split() returns a list of index objects for each level in f. The replacement method replaces these values with the corresponding element of value.

See Also

Other index methods: [.piar_index(), aggregate.piar_index, as.data.frame.piar_index(), as.ts.piar_index(), chain(), contrib(), head.piar_index(), is.na.piar_index(), lag.piar_index(), levels.piar_index(), mean.piar_index, merge.piar_index(), stack.piar_index(), time.piar_index(), window.piar_index()

Examples

```
index <- as_index(matrix(1:6, 2))

split(index, 1:2)

split(index, c(1, 1, 2), along = "time")
```

---

split_classification *Split a hierarchical classification*

Description

Expand a character representation of a hierarchical classification to make a price index aggregation structure by splitting along a delimiter.

Usage

```
split_classification(x, split, ..., sep = ".", pad = NA)
```

Arguments

x	[character] A character vector, or something that can be coerced into one, of codes/labels for a specific level in a classification (e.g., 5-digit COICOP).
split	[character(1)] A regular expression to delineate and split the levels in x. See base::strsplit().
...	Additional argument to pass to base::strsplit().
sep	[character(1)] A character used to delineate levels in x in the result. The default separates levels by ".".
pad	[character(1)] A string used to pad the shorter labels for an unbalanced classification. The default pads with NA.

Value

A list with a entry for each level in x giving the "digits" that represent each level in the hierarchy.

See Also

aggregation_structure() to make a price-index aggregation structure.

expand_classification() to expand a classification by the width of the levels.

combine_classifications() for combining multiple hierarchical classifications.

Examples

```
#' # A simple classification structure.
#            1
#      |-----+-----|
#      11          12
#  |---+---|       |
#  111     112     121

split_classification(c("111", "112", "121"), "")

# Useful if there are delimiters in the classification (like COICOP).
split_classification(c("01.1.1", "01.1.2", "01.2.1"), ".", fixed = TRUE)
```

---

stack.piar_index *Stack price indexes*

Description

`stack()` combines two price indexes with common levels, stacking index values and percent-change contributions for one index after the other.

`unstack()` breaks up a price index into a list of indexes for each time period.

These methods can be used in a map-reduce to make an index with multiple aggregation structures (like a Paasche index).

Usage

```
## S3 method for class 'chainable_piar_index'
stack(x, y, ...)

## S3 method for class 'direct_piar_index'
stack(x, y, ...)

## S3 method for class 'chainable_piar_index'
unstack(x, ...)

## S3 method for class 'direct_piar_index'
unstack(x, ...)
```

Arguments

`x`	`[piar_index]` A price index, as made by, e.g., `elementary_index()`.
`y`	`[piar_index]` A price index, or something that can coerced into one. If `x` is a period-over-period index then `y` is coerced into a chainable index; otherwise, `y` is coerced into a direct index.
`...`	Not currently used.

Value

`stack()` returns a combined price index that inherits from the same class as `x`.

`unstack()` returns a list of price indexes with the same class as `x`.

Note

It may be necessary to use `rebase()` prior to stacking fixed-based price indexes to ensure they have the same base period.

See Also

Other index methods: `[.piar_index()`, `aggregate.piar_index`, `as.data.frame.piar_index()`, `as.ts.piar_index()`, `chain()`, `contrib()`, `head.piar_index()`, `is.na.piar_index()`, `lag.piar_index()`, `levels.piar_index()`, `mean.piar_index`, `merge.piar_index()`, `split.piar_index()`, `time.piar_index()`, `window.piar_index()`

Examples

```
index1 <- as_index(matrix(1:6, 2))

index2 <- index1
time(index2) <- 4:6

stack(index1, index2)

# Unstack does the reverse.
all.equal(
  c(unstack(index1), unstack(index2)),
  unstack(stack(index1, index2))
)
```

`time.piar_index` *Get the time periods for a price index*

Description

Methods to get and set the time periods for a price index.

Usage

```
## S3 method for class 'piar_index'
time(x, ...)

time(x) <- value

## S3 replacement method for class 'piar_index'
time(x) <- value

set_time(x, value)

## S3 method for class 'piar_index'
start(x, ...)

## S3 method for class 'piar_index'
end(x, ...)

ntime(x)
```

Arguments

`x`	`[piar_index]` A price index, as made by, e.g., `elementary_index()`.
`...`	Not currently used.
`value`	`[character]` A character vector, or something that can be coerced into one, giving the replacement time periods for `x`.

Value

`time()` returns a character vector with the time periods for a price index. `start()` and `end()` return the first and last time period.

`ntime()` returns the number of time periods, analogous to `nlevels()`.

The replacement method returns a copy of `x` with the time periods in `value`. (`set_time()` is an alias that's easier to use with pipes.)

See Also

Other index methods: `[.piar_index()`, `aggregate.piar_index`, `as.data.frame.piar_index()`, `as.ts.piar_index()`, `chain()`, `contrib()`, `head.piar_index()`, `is.na.piar_index()`, `lag.piar_index()`, `levels.piar_index()`, `mean.piar_index`, `merge.piar_index()`, `split.piar_index()`, `stack.piar_index()`, `window.piar_index()`

---

transmute_weights *Transmute weights for a generalized mean*

Description

Transmute weight to turn a generalized mean of a given order into a generalized mean of any other order. Useful for calculating additive and multiplicative decompositions for generalized-mean indexes. See `vignette("decomposing-indexes")` for more details.

Usage

```
transmute_weights(x, weights = NULL, order = 0, to = 1, mean = NA)
```

Arguments

`x`	`[numeric > 0]` A strictly positive numeric vector.
`weights`	`[numeric >= 0]` A positive numeric vector of weights, the same length as `x`. The default is to equally weight each element of `x`.
`order`	`[numeric(1)]` A finite number giving the order of the generalized mean. The default transmutes the weights for a geometric mean.
`to`	`[numeric(1)]` A finite number giving the order of the target generalized mean. The default computes weights for an arithmetic mean.
`mean`	`[numeric(1)]` A finite number giving the generalized mean of `x` and `weights`, if known. The default computes this values.

Details

This function generalizes the additive and multiplicative decompositions for arithmetic and geometric indexes by Balk (2008, chapter 4). It returns a value such that

```
gmean(x, w, r) == gmean(x, transmute_weights(x, w, r, s), s)
```

Transmuting weights returns a value that is the same length as `x`, so any missing values in `x` or `weights` will return `NA`. Unless all values are `NA`, however, the result will still satisfy the above identity when `na.rm = TRUE`.

Value

A numeric vector, the same length as `x`, that sums to 1.

References

Balk BM (2008). *Price and Quantity Index Numbers*. Cambridge University Press. doi:10.1017/CBO9780511720758.

See Also

Other math functions: `emean()`, `gmean()`, `nested_gmean()`, `scale_weights()`, `transmute_weights2()`, `update_weights()`

Examples

```
x <- 1:3
w <- 3:1

# Calculate the geometric mean as an arithmetic mean.
gmean(x, order = 0)
gmean(x, transmute_weights(x, order = 0, to = 1), order = 1)
```

`transmute_weights2` *Transmute weights for a nested generalized mean*

Description

Transmute weights to turn a nested generalized mean of a given order into a generalized mean of any order. Useful for calculating additive and multiplicative decompositions for an index made of nested generalized means (e.g., Fisher index). See `vignette("decomposing-indexes")` for details.

Usage

```
transmute_weights2(
  x,
  weights = list(NULL, NULL),
  order = c(1, -1),
  outer_weights = NULL,
  outer_order = 0,
  to = 1,
  pivot = outer_order
)
```

Arguments

x	[numeric > 0] A strictly positive numeric vector.
weights	[list] A list of positive numeric vector of weights, each the same length as x, for both of the inner generalized means. NULL elements of weights equally weight each element of x. The default uses equal weights for both inner generalized mean.
order	[numeric(2)] A finite numeric vector giving the order of each of the inner generalized means. The default computes an arithmetic mean and a harmonic mean.
outer_weights	[numeric(2)] A strictly positive numeric vector weights for each of the inner generalized means as used in the outer generalized mean. The default weights each inner generalized mean equally.
outer_order	[numeric(1)] A finite number giving the order of the outer generalized mean. The default uses a geometric mean.
to	[numeric(1)] A finite number giving the order of the target generalized mean for the transmuted weights. The default constructs weights for an arithmetic mean.
pivot	[numeric(1)] A finite number giving the pivot value for the transmuted weights. The default uses the order of the outer generalized mean, otherwise to is common alternative.

Details

This function generalizes the additive and multiplicative decompositions for the Fisher index by Balk (2008, chapter 4). It returns a value such that

```
nested_gmean(x, list(w1, w2), c(r1, r2)) ==
    gmean(x, transmute_weights2(x, list(w1, w2), c(r1, r2), to = s), s)
```

Transmuting weights returns a value that is the same length as x, so any missing values in x or weights will return NA. Unless all values are NA, however, the result will still satisfy the above identity when na.rm = TRUE.

Value

A numeric vector, the same length as x, that sums to 1.

References

Balk BM (2008). *Price and Quantity Index Numbers*. Cambridge University Press. doi:10.1017/CBO9780511720758.

See Also

Other math functions: emean(), gmean(), nested_gmean(), scale_weights(), transmute_weights(), update_weights()

Examples

```
x <- 1:3
w1 <- 3:1
w2 <- c(1, 2, 1)

# Calculate the geometric mean of the arithmetic and harmonic means
# as an arithmetic mean.
nested_gmean(x, list(w1, w2))
gmean(x, transmute_weights2(x, list(w1, w2), to = 1))
```

`update.piar_aggregation_structure`
Update an aggregation structure

Description

Price update the weights in a price index aggregation structure.

Usage

```
## S3 method for class 'piar_aggregation_structure'
update(object, index, ..., period = NULL, order = 1, r = order)
```

Arguments

`object`	`[piar_aggregation_structure]` A price index aggregation structure, as made by `aggregation_structure()`.
`index`	`[piar_index]` A fixed-base (direct) price index, or something that can be coerced into one. Usually an aggregate price index as made by `aggregate()`.
`...`	Not currently used.
`period`	`[character(1)]` The time period used to price update the weights. The default uses the last period in `index`.
`order`	`[numeric(1)]` Order of the generalized mean to update the weights. The default is 1 for an arithmetic index.
`r`	Deprecated.

Value

A copy of `object` with price-updated weights using the index values in `index`.

See Also

`aggregate()` to make an aggregated price index.

Other aggregation structure methods: `as.matrix.piar_aggregation_structure()`, `cut.piar_aggregation_structure`
`levels.piar_aggregation_structure()`, `weights.piar_aggregation_structure()`

Examples

```
# A simple aggregation structure.
#            1
#      |-----+-----|
#      11          12
#  |---+---|       |
#  111     112     121
#  (1)     (3)     (4)
aggregation_weights <- data.frame(
  level1 = c("1", "1", "1"),
  level2 = c("11", "11", "12"),
  ea     = c("111", "112", "121"),
  weight = c(1, 3, 4)
)

pias <- as_aggregation_structure(aggregation_weights)

index <- as_index(
  matrix(1:9, 3, dimnames = list(c("111", "112", "121"), NULL))
)

weights(pias, ea_only = FALSE)

weights(update(pias, index), ea_only = FALSE)
```

`update_weights` *Update weights*

Description

Factor weights to turn the generalized mean of a product into the product of generalized means. Useful for price-updating the weights in a generalized-mean index.

Usage

```
update_weights(x, weights = NULL, order = 1)
```

Arguments

`x`	`[numeric > 0]` A strictly positive numeric vector.
`weights`	`[numeric >= 0]` A positive numeric vector of weights, the same length as `x`. The default is to equally weight each element of `x`.
`order`	`[numeric(1)]` A finite number giving the order (or exponent) of the generalized mean. The default calculates an arithmetic mean.

Details

This function generalizes the result in the PPI Manual (ILO, IMF, OECD, UNECE, and World Bank 2004, chapter 9, section C.5) for chaining the Young index, and gives a way to chain generalized-mean price indexes over time. It returns a value such that

```
gmean(x * y, w) ==
    gmean(x, w) * gmean(y, update_weights(x, w))
```

Factoring weights returns a value that is the same length as `x`, so any missing values in `x` or `weights` will return `NA`. Unless all values are `NA`, however, the result will still satisfy the above identity when `na.rm = TRUE`.

Value

A numeric vector the same length as `x`.

References

ILO, IMF, OECD, UNECE, World Bank (2004). *Producer price index manual: Theory and practice*. International Monetary Fund. doi:10.5089/9781589063044.069.

See Also

Other math functions: `emean()`, `gmean()`, `nested_gmean()`, `scale_weights()`, `transmute_weights()`, `transmute_weights2()`

Examples

```
x <- 1:3
y <- 4:6
w <- 3:1

# Factor the arithmetic mean by chaining the calculation.
gmean(x * y, w)
gmean(x, w) * gmean(y, update_weights(x, w))

# In cases where x and y have the same order, Chebyshev's
# inequality implies that the chained calculation is too small.
gmean(x * y, w) > gmean(x, w) * gmean(y, w)
```

`weights.piar_aggregation_structure`

Get the weights for an aggregation structure

Description

Get and set the weights for a price index aggregation structure.

Usage

```
## S3 method for class 'piar_aggregation_structure'
weights(object, ..., ea_only = TRUE, na.rm = FALSE)

weights(object) <- value

## S3 replacement method for class 'piar_aggregation_structure'
weights(object) <- value

set_weights(object, value)
```

Arguments

object	[piar_aggregation_structure] A price index aggregation structure, as made by aggregation_structure().
...	Not currently used.
ea_only	[logical(1)] Should weights be returned for only the elementary aggregates (the default)? Setting to FALSE gives the weights for the entire aggregation structure.
na.rm	[logical(1)] Should missing values be removed when aggregating the weights (i.e., when ea_only = FALSE)? By default, missing values are not removed.
value	[numeric >= 0] A numeric vector of weights for the elementary aggregates of object.

Value

weights() returns a named vector of weights for the elementary aggregates. The replacement method replaces these values without changing the aggregation structure. (set_weights() is an alias that's easier to use with pipes.)

If ea_only = FALSE then the return value is a list with a named vector of weights for each level in the aggregation structure.

See Also

Other aggregation structure methods: as.matrix.piar_aggregation_structure(), cut.piar_aggregation_structure levels.piar_aggregation_structure(), update.piar_aggregation_structure()

Examples

```
# A simple aggregation structure.
#            1
#      |-----+-----|
#      11          12
#  |---+---|       |
#  111     112     121
#  (1)     (3)     (4)
aggregation_weights <- data.frame(
  level1 = c("1", "1", "1"),
  level2 = c("11", "11", "12"),
```

```
  ea     = c("111", "112", "121"),
  weight = c(1, 3, 4)
)

pias <- as_aggregation_structure(aggregation_weights)

# Extract the weights.
weights(pias)

# ... or update them.
weights(pias) <- 1:3
weights(pias)
```

`window.piar_index` *Window a price index*

Description

Extract and replace index values over a window of time periods.

Usage

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

## S3 replacement method for class 'piar_index'
window(x, start = NULL, end = NULL, ...) <- value
```

Arguments

`x`	`[piar_index]` A price index, as made by, e.g., `elementary_index()`.
`start`	`[character(1)]` The time period to start the window. The default in the first period of `x`.
`end`	`[character(1)]` The time period to end the window. The default is the last period of `x`.
`...`	Not currently used.
`value`	`[numeric > 0 | piar_index]` A numeric vector or price index of replacement values.

Value

`window()` extracts a price index over a window of time periods that inherits from the same class as `x`. The replacement method replaces these with `value`.

See Also

Other index methods: `[.piar_index()`, `aggregate.piar_index`, `as.data.frame.piar_index()`, `as.ts.piar_index()`, `chain()`, `contrib()`, `head.piar_index()`, `is.na.piar_index()`, `lag.piar_index()`, `levels.piar_index()`, `mean.piar_index`, `merge.piar_index()`, `split.piar_index()`, `stack.piar_index()`, `time.piar_index()`

Examples

```
x <- as_index(matrix(1:9, 3))

window(x, "2")

window(x, "2") <- 1
x
```

Index

∗ **aggregation structure methods**
  as.matrix.piar_aggregation_structure, 10
  cut.piar_aggregation_structure, 22
  levels.piar_aggregation_structure, 40
  update.piar_aggregation_structure, 59
  weights.piar_aggregation_structure, 61
∗ **index methods**
  [.piar_index, 3
  aggregate.piar_index, 4
  as.data.frame.piar_index, 9
  as.ts.piar_index, 12
  chain, 17
  contrib, 20
  head.piar_index, 33
  is.na.piar_index, 37
  lag.piar_index, 39
  levels.piar_index, 41
  mean.piar_index, 41
  merge.piar_index, 44
  split.piar_index, 51
  stack.piar_index, 53
  time.piar_index, 55
  window.piar_index, 63
∗ **math functions**
  emean, 26
  gmean, 31
  nested_gmean, 45
  scale_weights, 50
  transmute_weights, 56
  transmute_weights2, 57
  update_weights, 60
∗ **price index functions**
  splice_index, 50
[.piar_index, 3
[.piar_index(), *7*, *10*, *13*, *19*, *21*, *34*, *38*, *40*, *41*, *43*, *44*, *52*, *54*, *56*, *63*
[<-.piar_index *([.piar_index)*, 3

aggregate(), *8*, *25*, *43*, *59*
aggregate.chainable_piar_index *(aggregate.piar_index)*, 4
aggregate.direct_piar_index *(aggregate.piar_index)*, 4
aggregate.piar_index, 4, *4*, *10*, *13*, *19*, *21*, *34*, *38*, *40*, *41*, *43*, *44*, *52*, *54*, *56*, *63*
aggregation_structure, 7
aggregation_structure(), *5*, *11*, *13*, *22*, *29*, *35*, *40*, *53*, *59*, *62*
anyNA.piar_index *(is.na.piar_index)*, 37
as.data.frame(), *8*, *14–16*, *25*
as.data.frame.piar_aggregation_structure *(as.matrix.piar_aggregation_structure)*, 10
as.data.frame.piar_index, 9
as.data.frame.piar_index(), *4*, *7*, *13*, *19*, *21*, *34*, *38*, *40*, *41*, *43*, *44*, *52*, *54*, *56*, *63*
as.matrix(), *6*, *8*, *14–16*, *25*
as.matrix.piar_aggregation_structure, 10
as.matrix.piar_aggregation_structure(), *22*, *40*, *59*, *62*
as.matrix.piar_index *(as.data.frame.piar_index)*, 9
as.ts.piar_index, 12
as.ts.piar_index(), *4*, *7*, *10*, *19*, *21*, *34*, *38*, *40*, *41*, *43*, *44*, *52*, *54*, *56*, *63*
as_aggregation_structure, 13
as_aggregation_structure(), *8*, *11*
as_index, 14
as_index(), *10*, *18*, *25*

back_period, 16
back_period(), *30*, *46*, *47*, *49*
base::all.equal(), *27*

base::as.data.frame(), *10, 11*
base::as.data.frame.list(), *10, 11*
base::cut.Date(), *16*, *24*, *30*, *35*, *49*
base::make.unique(), *5*, *24*, *43*
base::max(), *32*
base::min(), *32*
base::split.default(), *52*
base::strsplit(), *53*

carry_backward *(*impute_prices*)*, 34
carry_forward *(*impute_prices*)*, 34
chain, 17
chain(), *4*, *7*, *10*, *13*, *21*, *25*, *34*, *38*, *40, 41*, *43, 44*, *52*, *54*, *56*, *63*
chainable_piar_index, *16*, *19*, *25*, *39*
chainable_piar_index *(*piar_index*)*, 48
combine_classifications, 19
combine_classifications(), *29*, *53*
contrib, 20
contrib(), *4*, *7*, *10*, *13*, *19*, *34*, *38*, *40, 41*, *43, 44*, *52*, *54*, *56*, *63*
contrib2DF *(*contrib*)*, 20
contrib<- *(*contrib*)*, 20
cut.piar_aggregation_structure, 22
cut.piar_aggregation_structure(), *11*, *40*, *59*, *62*

direct_piar_index, *16*, *19*, *25*, *39*
direct_piar_index *(*piar_index*)*, 48

elemental_index *(*elementary_index*)*, 23
elementary_index, 23
elementary_index(), *3*, *5*, *8*, *10*, *12*, *18*, *20*, *33*, *37*, *39*, *41, 42*, *44*, *52*, *54, 55*, *63*
emean, 26
emean(), *33*, *45*, *50*, *57, 58*, *61*
end.piar_index *(*time.piar_index*)*, 55
expand_classification, 28
expand_classification(), *8*, *19*, *53*

fs_prices *(*price_data*)*, 48
fs_weights *(*price_data*)*, 48

geks_index, 29
gmean, 31
gmean(), *5, 6*, *24*, *27*, *35*, *42*, *45*, *50*, *57, 58*, *61*

hb_transform *(*outliers*)*, 46
head.piar_index, 33
head.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *38*, *40, 41*, *43, 44*, *52*, *54*, *56*, *63*

impute_prices, 34
impute_prices(), *25*, *49*
interact_classifications *(*expand_classification*)*, 28
is.na.piar_index, 37
is.na.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *34*, *40, 41*, *43, 44*, *52*, *54*, *56*, *63*
is_aggregation_structure, 38
is_chainable_index *(*is_index*)*, 39
is_direct_index *(*is_index*)*, 39
is_index, 39

lag.piar_index, 39
lag.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *34*, *38*, *41*, *43, 44*, *52*, *54*, *56*, *63*
levels.piar_aggregation_structure, 40
levels.piar_aggregation_structure(), *11*, *22*, *59*, *62*
levels.piar_index, 41
levels.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *34*, *38*, *40*, *43, 44*, *52*, *54*, *56*, *63*
levels<-.piar_index *(*levels.piar_index*)*, 41

mean.chainable_piar_index *(*mean.piar_index*)*, 41
mean.direct_piar_index *(*mean.piar_index*)*, 41
mean.piar_index, *4*, *7*, *10*, *13*, *19*, *21*, *34*, *38*, *40*, 41, *41*, *44*, *52*, *54*, *56*, *63*
merge(), *6*
merge.chainable_piar_index *(*merge.piar_index*)*, 44
merge.direct_piar_index *(*merge.piar_index*)*, 44
merge.piar_index, 44
merge.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *34*, *38*, *40, 41*, *43*, *52*, *54*, *56*, *63*
ms_prices *(*price_data*)*, 48
ms_weights *(*price_data*)*, 48

nested_gmean, 45
nested_gmean(), *27*, *33*, *50*, *57, 58*, *61*
ntime *(*time.piar_index*)*, 55

outliers, 46

outliers(), *17*, *49*

piar_aggregation_structure, *14*, *22*, *38*
piar_aggregation_structure
(aggregation_structure), 7
piar_index, *16*, *25*, *39*, 48
price_data, 48
price_relative, 49
price_relative(), *24*, *25*, *36*

rebase (chain), 17
rebase(), *25*, *54*

scale_weights, 50
scale_weights(), *27*, *33*, *45*, *57*, *58*, *61*
set_contrib (contrib), 20
set_contrib_from_index (contrib), 20
set_levels (levels.piar_index), 41
set_time (time.piar_index), 55
set_weights
(weights.piar_aggregation_structure), 61
shadow_price (impute_prices), 34
splice_index, 50
splice_index(), *31*
split.piar_index, 51
split.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *34*, *38*, *40*, *41*, *43*, *44*, *54*, *56*, *63*
split<-.piar_index (split.piar_index), 51
split_classification, 52
split_classification(), *19*, *29*
stack.chainable_piar_index
(stack.piar_index), 53
stack.direct_piar_index
(stack.piar_index), 53
stack.piar_index, 53
stack.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *34*, *38*, *40*, *41*, *43*, *44*, *52*, *56*, *63*
start.piar_index (time.piar_index), 55
stats::quantile(), *46*
stats::ts, *12*
stats::ts(), *12*
stats::weighted.mean(), *32*

tail.piar_index (head.piar_index), 33
time.piar_index, 55
time.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *34*, *38*, *40*, *41*, *43*, *44*, *52*, *54*, *63*
time<- (time.piar_index), 55
transmute_weights, 56
transmute_weights(), *6*, *24*, *27*, *33*, *45*, *50*, *58*, *61*
transmute_weights2, 57
transmute_weights2(), *6*, *25*, *27*, *33*, *45*, *50*, *57*, *61*

unchain (chain), 17
unstack.chainable_piar_index
(stack.piar_index), 53
unstack.direct_piar_index
(stack.piar_index), 53
update(), *8*
update.piar_aggregation_structure, 59
update.piar_aggregation_structure(), *11*, *22*, *40*, *62*
update_weights, 60
update_weights(), *6*, *27*, *33*, *45*, *50*, *57*, *58*
utils::head(), *33*
utils::tail(), *33*

weights(), *8*
weights.piar_aggregation_structure, 61
weights.piar_aggregation_structure(), *11*, *22*, *40*, *59*
weights<-
(weights.piar_aggregation_structure), 61
window.piar_index, 63
window.piar_index(), *4*, *7*, *10*, *13*, *19*, *21*, *34*, *38*, *40*, *41*, *43*, *44*, *52*, *54*, *56*
window<-.piar_index
(window.piar_index), 63