Package 'hicp'

July 28, 2025

Type Package

Title Harmonised Index of Consumer Prices

Version 1.0.0

Description The Harmonised Index of Consumer Prices (HICP) is the key economic figure to measure inflation in the euro area.
The methodology underlying the HICP is documented in the HICP Methodological Manual (<https://ec.europa.eu/eurostat/web/products-manuals-and-guidelines/w/ks-gq-24-003>).
Based on the manual, this package provides functions to access and work with HICP data from Eurostat's public database (<https://ec.europa.eu/eurostat/data/database>).

License EUPL

Encoding UTF-8

LazyData true

Depends R (>= 3.5.0)

Imports restatapi (>= 0.24.0), data.table (>= 1.16.0)

Suggests knitr, rmarkdown, testthat (>= 3.0.0)

Config/testthat/edition 3

VignetteBuilder knitr

NeedsCompilation no

URL https://github.com/eurostat/hicp

BugReports https://github.com/eurostat/hicp/issues

Author Sebastian Weinand [aut, cre]

Maintainer Sebastian Weinand <sebastian.weinand@ec.europa.eu>

Repository CRAN

Date/Publication 2025-07-28 11:40:02 UTC

Contents

`chaining` *Chain-linking, rebasing and index conversion*

Description

Function `unchain()` unchains a chained index series. These unchained index series can be aggregated into higher-level indices using `aggregate()`. To obtain a long-term index series, the higher-level indices must be chained using function `chain()`. Function `rebase()` sets the index reference period. Monthly indices can be converted into quarterly and yearly indices or 12-month moving averages using function `convert()`.

Usage

```
unchain(x, t, by=12, settings=list())

chain(x, t, by=12, settings=list())

rebase(x, t, t.ref="first", settings=list())

convert(x, t, type="year", settings=list())
```

Arguments

`x`	numeric vector of index values.
`t`	date vector in format `YYYY-MM-DD` with monthly frequency, that is, one observation per month. Quarterly and yearly frequencies are also supported.
`by`	for one-month or one-quarter overlap a single integer between 1 and 12 specifying the price reference period; for annual overlap using a full calendar year `NULL`.
`t.ref`	character specifying the index reference period either in format `YYYY` for a calendar year or `YYYY-MM` for a specific month or quarter. Can also be `first` or `last` to use the first or last available period. If `t.ref` contains multiple entries, these are processed in the order provided, and the first match is used for the rebasing.

`type` type of converted index. Either `year` (for annual average), `quarter` (for quarterly average), or `12mavg` (for a 12-month moving average).

`settings` list of control settings to be used. The following settings are supported:

- `chatty` : logical indicating if package-specific warnings and info messages should be printed or not. The default is `getOption("hicp.chatty")`.
- `freq` : character specifying the frequency of `t`. Allowed values are `month`, `quarter`, `year`, and `auto` (the default). For `auto`, the frequency is internally derived from `t`.
- `na.rm` : logical indicating if averages for calendar years should also be computed when there are `NA`s and less than 12 months (or 4 quarters) present (for `na.rm=TRUE`). For the 12-month moving average in `convert()`, the calculations are always based on the last 12 months (or 4 quarters), meaning that only `NA`s are excluded. The default is `na.rm=FALSE`.

Details

Function `unchain()` sets the value of the first price reference period to `NA` although the value could be set to 100 (if `by` is not `NULL`) or 100 divided by the average of the year (if `by=NULL`). This is wanted to avoid aggregation of these values. Function `chain()` finally sets the values back to 100.

Value

Functions `unchain()`, `chain()`, `rebase()`, and `convert(..., type="12mavg")` return numeric values of the same length as `x`.

For `type="year"` and `type="quarter"`, function `convert()` returns a named numeric vector of the length of quarters or years available in `t`, where the names correspond to the last month of the year or quarter.

Author(s)

Sebastian Weinand

References

European Commission, Eurostat, *Harmonised Index of Consumer Prices (HICP) - Methodological Manual - 2024 edition*, Publications Office of the European Union, 2024, https://data.europa.eu/doi/10.2785/055028.

See Also

`aggregate`

Examples

```
### EXAMPLE 1

t <- seq.Date(from=as.Date("2021-12-01"), to=as.Date("2024-12-01"), by="1 month")
p <- rnorm(n=length(t), mean=100, sd=5)
```

```
# rebase index to new reference period:
rebase(x=p, t=t, t.ref=c("1996","2022")) # 1996 not present so 2022 is used
rebase(x=p, t=t, t.ref=c("1996","first")) # 1996 not present so first period is used

# convert into quarterly index:
convert(x=p, t=t, type="q") # first quarter is not complete so NA

# unchaining and chaining gives initial results:
100*p/p[1]
chain(unchain(p, t, by=12), t, by=12)

# use annual overlap:
100*p/mean(p[1:12])
(res <- chain(unchain(p, t, by=NULL), t, by=NULL))
# note that for backwards compability, each month in the first
# year receives an index value of 100. this allows the same
# computation again:
chain(unchain(res, t, by=NULL), t, by=NULL)

### EXAMPLE 2: Working with published HICP data

library(data.table)
library(restatapi)
options(restatapi_cores=1) # set cores for testing on CRAN
options(hicp.chatty=FALSE) # suppress package messages and warnings

# get hicp index values for euro area with base 2015:
dt <- hicp::data(id="prc_hicp_midx", filter=list(unit="I15", geo="EA"))
dt[, "time":=as.Date(paste0(time, "-01"))]
setkeyv(x=dt, cols=c("unit","coicop","time"))

# unchain, chain, and rebase all euro area indices by coicop:
dt[, "dec_ratio" := unchain(x=values, t=time), by="coicop"]
dt[, "chained_index" := chain(x=dec_ratio, t=time), by="coicop"]
dt[, "index_own" := rebase(x=chained_index, t=time, t.ref="2015"), by="coicop"]

# convert all euro area indices into annual averages:
dta <- dt[, as.data.table(
              x=convert(x=values, t=time, type="year"),
              keep.rownames=TRUE), by="coicop"]
setnames(x=dta, c("coicop","time","index"))
plot(index~as.Date(time), data=dta[coicop=="CP00",], type="l") # plot all-items index
```

---

coicop *COICOP codes, bundles and relatives*

Description

Function `is.coicop()` checks if the input is a valid COICOP code while `level()` returns the level (e.g. division or subclass).

For HICP data, COICOP codes are sometimes merged into bundles (e.g. `08X`, `0531_2`), deviating from the usual code structure. Function `is.bundle()` flags if a COICOP code is a bundle or not, while `unbundle()` resolves the bundles into the underlying valid codes.

Functions `parent()` and `child()` derive the higher-level parents or lower-level children of a COICOP code.

Usage

```
is.coicop(id, settings=list())

level(id, label=FALSE, settings=list())

is.bundle(id, settings=list())

unbundle(id, settings=list())

child(id, usedict=TRUE, closest=TRUE, k=1, settings=list())

parent(id, usedict=TRUE, closest=TRUE, k=1, settings=list())
```

Arguments

`id`	character vector of COICOP codes.
`label`	logical indicating if the number of digits or the labels (e.g., division, subclass) should be returned.
`usedict`	logical indicating if parents or children should be derived from the full code dictionary defined by `settings$coicop.version` (if set to `TRUE`) or only from the codes present in `id`.
`closest`	logical indicating if the closest parents or children should be derived or the *k*-th ones defined by `k`. For example, if set to `TRUE`, the closest parent could be the direct parent for one code (e.g. `031->03`) and the grandparent for another (e.g. `0321->03`).
`k`	integer specifying the *k*-th relative (e.g., `1` for direct parents or children, `2` for grandparents and grandchildren, ...). Multiple values allowed, e.g., `k=c(1,2)`. Only relevant if `closest=FALSE`.
`settings`	list of control settings to be used. The following settings are supported: • `coicop.version` : character specifying the COICOP version to be used by `is.coicop()`, `level()`, `child()`, and `parent()` for flagging valid COICOP codes. See details for the allowed values. The default is `getOption("hicp.coicop.version")`. • `all.items.code` : character specifying the code internally used by `level()`, `child()`, and `parent()` for the all-items index. The default is taken from `getOption("hicp.all.items.code")`. • `coicop.bundles` : named list specifying the COICOP bundle code dictionary used for unbundling any bundle codes in `id`. The default is `getOption("hicp.coicop.bundles` • `simplify` : logical indicating if the output of `child()`, `parent()` or `unbundle()` should be simplified into a vector if possible. The default is `FALSE`.

- For `child()` and `parent()`: If both a COICOP bundle code and the underlying codes are the parent or child, only the latter codes are kept (e.g., `c(08X,082)->082`). Note that simplification usually only works for `parent()`.
- For `unbundle()`: All codes underlying the bundle code are kept, meaning that the resulting vector length can exceed the length of `id`.

Details

The following COICOP versions are supported:

- Classification of Individual Consumption According to Purpose (COICOP-1999): `coicop1999`
- European COICOP (version 1, ECOICOP): `ecoicop`
- ECOICOP adopted to the needs of the HICP (version 1, ECOICOP-HICP): `ecoicop.hicp`
- COICOP-2018: `coicop2018`
- ECOICOP (version 2, ECOICOP 2): `ecoicop2`

The COICOP version can be set temporarily in the function settings or globally via `options(hicp.coicop.version)`. The package default is `ecoicop.hicp`.

None of the COICOP versions include a code for the all-items index. By default, the internal package code for the all-items index is defined by `options(hicp.all.items.code="00")` but can be changed by the user. The level is always 1.

Although bundle codes (e.g. `08X`, `0531_2`) are no valid COICOP codes, they are internally resolved into their underlying codes and processed in that way if they can be found in the bundle code dictionary (see `getOption("hicp.coicop.bundles")`). If bundle codes should not be processed, the dictionary can be cleared by `options("hicp.coicop.bundles"=list())`.

Value

Functions `is.coicop()` and `is.bundle()` return a logical vector, function `level()` an integer vector, and functions `child()`, `parent()`, and `unbundle()` a list. All function outputs have the same length as `id`.

Author(s)

Sebastian Weinand

See Also

`tree`

Examples

```
### EXAMPLE 1

# check if coicop codes are valid:
is.coicop(id=c("00","CP00","01","011","13","08X"))

# get the coicop level or label:
```

```
level(id=c("00","05","053","0531_2"))
level(id=c("00","05","053","0531_2"), label=TRUE)

# derive children and parents

# no children of 01 present in ids:
child(id=c("01"), usedict=FALSE)

# still no direct child present:
child(id=c("01","0111"), usedict=FALSE, closest=FALSE, k=1)

# but a grandchild of 01 is found:
child(id=c("01","0111"), usedict=FALSE, closest=TRUE)

# derive the children from the code dictionary:
child(id=c("01"), usedict=TRUE)

# two parents found for 05311 due to presence of bundle code:
parent(id=c("0531","0531_2","05311","05321"), usedict=FALSE)

# simplification removes bundle code:
parent(id=c("0531","0531_2","05311","05321"), usedict=FALSE, settings=list(simplify=TRUE))

### EXAMPLE 2: Working with published HICP data

library(data.table)
library(restatapi)
options(restatapi_cores=1) # set cores for testing on CRAN
options(hicp.chatty=FALSE) # suppress package messages and warnings

# load hicp item weights of euro area:
coicops <- hicp::data(id="prc_hicp_inw", filter=list(geo="EA"))
coicops <- coicops[grepl("^CP", coicop),]
coicops[, "coicop":=gsub("^CP", "", coicop)]

# show frequency of coicop levels over time:
coicops[, .N, by=list(time, "lvl"=level(coicop))]

# get coicop parent from the data:
coicops[, "parent":=parent(id=coicop, usedict=FALSE, settings=list(simplify=TRUE)), by="time"]

# flag if coicop has child available in the data:
coicops[, "has_child":=lengths(child(id=coicop, usedict=FALSE))>0, by="time"]
coicops[has_child==FALSE, sum(values, na.rm=TRUE), by="time"]
# coicop bundles and their component ids are both taken into
# account. this double counting explains some differences
```

---

countries *Country metadata*

Description

This data set contains metadata for the euro area, EU, EFTA, and candidate countries that submit(ted) HICP data on a regular basis.

Usage

```
# country metadata:
countries
```

Format

A data.table with metadata on the individual euro area (EA), EU, EFTA, and candidate countries producing the HICP.

- `code`: the country code
- `name_[en|fr|de]`: the country name in English, French, and German
- `protocol_order`: the official protocol order of countries
- `is_eu`, `is_ea`, `is_efta`, `is_candidate`: a logical indicating if a country belongs to the EU, the euro area, or if it's an EFTA or candidate country, respectively
- `eu_since`, `eu_until`: date of joining and leaving the European Union
- `ea_since`: the date of introduction of the euro as the official currency
- `index_decimals`: the number of index decimals used for dissemination

Author(s)

Sebastian Weinand

Examples

```
# subset to euro area countries:
countries[is_ea==TRUE, ]
```

---

hicp.data *Download HICP data*

Description

These functions are simple wrappers of functions in the `restatapi` package.

The function `datasets()` lists all available HICP data sets in Eurostat's public database, while `datafilters()` gives the allowed values that can be used for filtering a data set. The function `data()` downloads a specific data set with filtering on key parameters and time, if supplied.

Usage

```
datasets(pattern="^prc_hicp", ...)

datafilters(id, ...)

data(id, filters=list(), date.range=NULL, flags=FALSE, ...)
```

Arguments

pattern	character for pattern matching on data set identifier. See also grepl.
id	data set identifier, which can be obtained from datasets().
filters	named list of filters to be applied to the data request. Allowed values for filtering can be retrieved from datafilters(). For HICP data, typical filter variables are the index reference period (unit: I96, I05, I15), the country (geo: EA, DE, FR, ...), or the COICOP code (coicop: CP00, CP01, SERV, ...).
date.range	vector of start and end date used for filtering on time dimension. These must follow the pattern YYYY(-MM)?. An open interval can be defined by setting one date to NA.
flags	logical indicating if data flags should be returned or not.
...	further arguments passed to functions: • get_eurostat_toc for datasets() • get_eurostat_dsd for datafilters() • get_eurostat_data for data()

Value

A data.table.

Author(s)

Sebastian Weinand

Source

See Eurostat's public database at https://ec.europa.eu/eurostat/web/main/data/database.

Examples

```
# set cores for testing on CRAN:
library(restatapi)
options(restatapi_cores=1)

# view available HICP data sets:
datasets()

# get allowed filters for item weights:
datafilters(id="prc_hicp_inw")
```

```
# download item weights since 2015 for euro area:
data(id="prc_hicp_inw", filters=list("geo"="EA"), date.range=c("2015", NA))
```

`index.aggregation` *Index number functions and aggregation*

Description

Lower-level price indices can be aggregated into higher-level indices in a single step using the bilateral index formulas below or gradually following the COICOP tree with the function `aggregate.tree()`.

The functions `aggregate()` and `disaggregate()` can be used for the calculation of user-defined aggregates (e.g., HICP special aggregates). For `aggregate()`, lower-level indices are aggregated into the respective total. For `disaggregate()`, they are deducted from the total to receive a subaggregate.

Usage

```
# bilateral price index formulas:
jevons(x)
carli(x)
harmonic(x)
laspeyres(x, w0)
paasche(x, wt)
fisher(x, w0, wt)
toernqvist(x, w0, wt)
walsh(x, w0, wt)

# aggregation into user-defined aggregates:
aggregate(x, w0, wt, id, formula=laspeyres, agg=list(), settings=list())

# disaggregation into user-defined aggregates:
disaggregate(x, w0, id, agg=list(), settings=list())

# gradual aggregation following the COICOP tree:
aggregate.tree(x, w0, wt, id, formula=laspeyres, settings=list())
```

Arguments

`x`	numeric vector of price relatives between two periods, typically obtained by unchaining some HICP index series.
`w0`, `wt`	numeric vector of weights in the base period `w0` (e.g., for the Laspeyres index) or current period `wt` (e.g., for the Paasche index).
`id`	character vector of aggregate codes. For `aggregate.tree()`, only valid COICOP codes or bundle codes are processed.
`formula`	a function or named list of functions specifying the index formula(s) used for aggregation. Each function must return a scalar and have the argument `x`. For weighted index formulas, the arguments `w0` and/or `wt` must be available as well.

agg — list of user-defined aggregates to be calculated. For disaggregate(), the list must have names specifying the aggregate from which indices are deducted. Each list element is a vector of codes that can be found in id. See settings$exact for further specification of this argument.

settings — list of control settings to be used. The following settings are supported:

- chatty : logical indicating if package-specific warnings and info messages should be printed or not. The default is getOption("hicp.chatty").
- coicop.version : character specifying the COICOP version to be used for flagging valid COICOP codes. See coicop for the allowed values. The default is getOption("hicp.coicop.version").
- all.items.code : character specifying the code internally used for the all-items index. The default is taken from getOption("hicp.all.items.code").
- coicop.bundles : named list specifying the COICOP bundle code dictionary used for unbundling any bundle codes in id. The default is getOption("hicp.coicop.bundles
- exact : logical indicating if the codes in agg must all be present in id for aggregation or not. If FALSE, aggregation is carried out using the codes present in agg. If TRUE and some codes cannot be found in id, NA is returned. The default is TRUE.
- names : character of names for the aggregates in agg. If not supplied, the aggregates are numbered.

Details

The bilateral index formulas currently available are intended for the aggregation of (unchained) price relatives x. The Dutot index is therefore not implemented.

Value

The functions jevons(), carli(), harmonic(), laspeyres(), paasche(), fisher(), toernqvist(), and walsh() return a single aggregated value.

The functions aggregate(), disaggregate() and aggregate.tree() return a data.table with the sum of weights w0 and wt (if supplied) and the computed aggregates for each index formula specified by formula.

Author(s)

Sebastian Weinand

References

European Commission, Eurostat, *Harmonised Index of Consumer Prices (HICP) - Methodological Manual - 2024 edition*, Publications Office of the European Union, 2024, https://data.europa.eu/doi/10.2785/055028.

See Also

unchain, chain, rebase

Examples

```
library(data.table)

### EXAMPLE 1

# example data with unchained prices and weights:
dt <- data.table("coicop"=c("0111","0112","012","021","022"),
                 "price"=c(102,105,99,109,115),
                 "weight"=c(0.2,0.15,0.4,0.2,0.05))

# aggregate directly into overall index:
dt[, laspeyres(x=price, w0=weight)]

# same result at top level with gradual aggregation:
(dtagg <- dt[, aggregate.tree(x=price, w0=weight, id=coicop)])

# compute user-defined aggregates by disaggregation:
dtagg[, disaggregate(x=laspeyres, w0=w0, id=id,
                     agg=list("00"=c("01"), "00"=c("022")),
                     settings=list(names=c("A","B")))]

# which can be similarly derived by aggregation:
dtagg[, aggregate(x=laspeyres, w0=w0, id=id,
                  agg=list(c("021","022"), c("011","012","021")),
                  settings=list(names=c("A","B")))]

# same aggregates by several index formulas:
dtagg[, aggregate(x=laspeyres, w0=w0, id=id,
                  agg=list(c("021","022"), c("011","012","021")),
                  formula=list("lasp"=laspeyres, "jev"=jevons, "mean"=mean),
                  settings=list(names=c("A","B")))]

# no aggregation if one index is missing:
dtagg[, aggregate(x=laspeyres, w0=w0, id=id,
                  agg=list(c("01","02","03")),
                  settings=list(exact=TRUE))]

# or just use the available ones:
dtagg[, aggregate(x=laspeyres, w0=w0, id=id,
                  agg=list(c("01","02","03")),
                  settings=list(exact=FALSE))]

### EXAMPLE 2: Index aggregation using published HICP data

library(restatapi)
options(restatapi_cores=1)  # set cores for testing on CRAN
options(hicp.chatty=FALSE)  # suppress package messages and warnings

# import monthly price indices:
prc <- hicp::data(id="prc_hicp_midx", filter=list(unit="I15", geo="EA"))
prc[, "time":=as.Date(paste0(time, "-01"))]
prc[, "year":=as.integer(format(time, "%Y"))]
```

```
setnames(x=prc, old="values", new="index")

# unchaining indices:
prc[, "dec_ratio" := unchain(x=index, t=time), by="coicop"]

# import item weights:
inw <- hicp::data(id="prc_hicp_inw", filter=list(geo="EA"))
inw[, "time":=as.integer(time)]
setnames(x=inw, old=c("time","values"), new=c("year","weight"))

# derive coicop tree at lowest possible level:
inw[grepl("^CP",coicop),
   "tree":=tree(id=gsub("^CP","",coicop), w=weight, flag=TRUE, settings=list(w.tol=0.1)),
    by=c("geo","year")]

# except for rounding, we receive total weight of 1000 in each period:
inw[tree==TRUE, sum(weight), by="year"]

# merge price indices and item weights:
hicp.data <- merge(x=prc, y=inw, by=c("geo","coicop","year"), all.x=TRUE)
hicp.data <- hicp.data[year <= year(Sys.Date())-1 & grepl("^CP\\d+", coicop),]
hicp.data[, "coicop" := gsub(pattern="^CP", replacement="", x=coicop)]

# compute all-items HICP in one step using only lowest-level indices:
hicp.own <- hicp.data[tree==TRUE,
                      list("laspey"=laspeyres(x=dec_ratio, w0=weight)),
                      by="time"]
setorderv(x=hicp.own, cols="time")
hicp.own[, "chain_laspey" := chain(x=laspey, t=time, by=12)]
hicp.own[, "chain_laspey_15" := rebase(x=chain_laspey, t=time, t.ref="2015")]

# compute all-items HICP gradually through all higher-levels:
hicp.own.all <- hicp.data[, aggregate.tree(x=dec_ratio, w0=weight, id=coicop), by="time"]
setorderv(x=hicp.own.all, cols="time")
hicp.own.all[, "chain_laspey" := chain(x=laspeyres, t=time, by=12), by="id"]
hicp.own.all[, "chain_laspey_15" := rebase(x=chain_laspey, t=time, t.ref="2015"), by="id"]

# compare all-items HICP from direct and gradual aggregation:
agg.comp <- merge(x=hicp.own.all[id=="00", list(time, "index_stpwse"=chain_laspey_15)],
                  y=hicp.own[, list(time, "index_direct"=chain_laspey_15)],
                  by="time")

# no differences -> consistent in aggregation:
head(agg.comp[abs(index_stpwse-index_direct)>1e-4,])
```

`linking` *Linking-in new index series*

Description

Function `link()` links a new index series (`x.new`) to an existing one (`x`) using the overlap periods in `t.overlap`. In the resulting linked index series, the new index series starts after the existing one.

Function `lsf()` computes the level-shift factors for linking via the overlap periods in `t.overlap` in comparison to the one-month overlap method using December of year `t-1`. The level-shift factors can then be used to shift the index level of a HICP index series.

Usage

```
link(x, x.new, t, t.overlap=NULL, settings=list())

lsf(x, x.new, t, t.overlap=NULL, settings=list())
```

Arguments

`x`, `x.new`	numeric vector of index values. `NA`-values in the vectors indicate when the index series discontinues (for `x`) or starts (for `x.new`).
`t`	date vector in format `YYYY-MM-DD` with monthly frequency, that is, one observation per month. Quarterly and yearly frequencies are also supported.
`t.overlap`	character specifying the overlap period either in format `YYYY` for a calendar year or `YYYY-MM` for a specific month or quarter. Multiple periods can be provided. If `NULL`, all intersecting periods in `x` and `x.new` are used.
`settings`	list of control settings to be used. The following settings are supported: • `chatty` : logical indicating if package-specific warnings and info messages should be printed or not. The default is `getOption("hicp.chatty")`. • `freq` : character specifying the frequency of `t`. Allowed values are `month`, `quarter`, `year`, and `auto` (the default). For `auto`, the frequency is internally derived from `t`. • `na.rm` : logical indicating if averages for calendar years should also be computed when there are `NA`s and less than 12 months (or 4 quarters) present (for `na.rm=TRUE`).

Value

Function `link()` returns a numeric vector or a matrix of the same length as `t`, while `lsf()` provides a named numeric vector of the same length as `t.overlap`.

Author(s)

Sebastian Weinand

See Also

`chain`

Examples

```
# input data:
set.seed(1)
t <- seq.Date(from=as.Date("2015-01-01"), to=as.Date("2024-05-01"), by="1 month")
x.new <- rnorm(n=length(t), mean=100, sd=5)
x.new <- rebase(x=x.new, t=t, t.ref="2019-12")
x.old <- x.new + rnorm(n=length(x.new), sd=5)
x.old <- rebase(x=x.old, t=t, t.ref="2015")
x.old[t>as.Date("2021-12-01")] <- NA # current index discontinues in 2021
x.new[t<as.Date("2020-01-01")] <- NA # new index starts in 2019-12

# linking in new index in different periods:
matplot(x=t,
        y=link(x=x.old, x.new=x.new, t=t, t.overlap=c("2021-12","2020","2021")),
        col=c("red","blue","green"), type="l", lty=1,
        xlab=NA, ylab="Index", ylim=c(80,120))
lines(x=t, y=x.old, col="black")
abline(v=as.Date("2021-12-01"), lty="dashed")
legend(x="topleft",
       legend=c("One-month overlap using December 2021",
                "Annual overlap using 2021",
                "Annual overlap using 2020"),
       fill=c("red","green","blue"), bty = "n")

# compute level-shift factors:
lsf(x=x.old, x.new=x.new, t=t, t.overlap=c("2020","2021"))

# level-shift factors can be applied to already chain-linked index series
# to obtain linked series using another overlap period:
x.new.chained <- link(x=x.old, x.new=x.new, t=t, t.overlap="2021-12")

# level-shift adjustment:
x.new.adj <- ifelse(test=t>as.Date("2021-12-01"),
                    yes=x.new.chained*lsf(x=x.old, x.new=x.new, t=t, t.overlap="2020"),
                    no=x.new.chained)

# compare:
all.equal(x.new.adj, link(x=x.old, x.new=x.new, t=t, t.overlap="2020"))
```

`rates` *Change rates and contributions*

Description

Function `rates()` derives monthly, quarterly and annual rates of change from an index series.

Function `contrib()` computes the contributions of a subcomponent (e.g., food, energy) to the change rate of the overall index (for chained indices with price reference period December of the previous year).

Usage

```
rates(x, t, type="year", settings=list())

contrib(x, w, t, x.all, w.all, type="year", settings=list())
```

Arguments

x, x.all	numeric vector of index values of the subcomponent (x) and the overall index (x.all).
w, w.all	numeric vector of weights of the subcomponent (w) and the overall index (w.all).
t	date vector in format YYYY-MM-DD with monthly frequency, that is, one observation per month. Quarterly and yearly frequencies are also supported.
type	character specifying the type of change rate. Allowed values are month for monthly change rates, quarter for quarterly change rates, and year for annual change rates. See also details.
settings	list of control settings to be used. The following settings are supported:

- chatty : logical indicating if package-specific warnings and info messages should be printed or not. The default is getOption("hicp.chatty").
- freq : character specifying the frequency of t. Allowed values are month, quarter, year, and auto (the default). For auto, the frequency is internally derived from t.
- method : character specifying the method for decomposing the change rates. Allowed values are ribe (the default) and kirchner.

Details

For monthly frequency, the change rates show the percentage change of x in the current month compared to the previous month (monthly change rates, *m-1*), compared to three months ago (quarterly change rates, *m-3*), or compared to the same month one year before (annual change rates, *m-12*).

For quarterly frequency, the change rates show the percentage change of x in the current quarter compared to the previous quarter (quarterly change rates, *q-1*) or compared to the same quarter one year before (annual change rates, *q-4*).

For yearly frequency, the change rates show the percentage change of x in the current year compared to the previous year (annual change rates, *y-1*). If x is an annual index produced by convert(), the annual change rates correspond to annual average change rates.

Value

A numeric vector of the same length as x.

Author(s)

Sebastian Weinand

References

European Commission, Eurostat, *Harmonised Index of Consumer Prices (HICP) - Methodological Manual - 2024 edition*, Publications Office of the European Union, 2024, https://data.europa.eu/doi/10.2785/055028.

Examples

```
### EXAMPLE 1

p <- rnorm(n=37,mean=100,sd=5)
t <- seq.Date(from=as.Date("2020-12-01"), by="1 month", length.out=length(p))

# compute change rates:
rates(x=p, t=t, type="month") # one month to the previous month
rates(x=p, t=t, type="year")  # month to the same month of previous year

# compute annual average rate of change:
pa <- convert(x=p, t=t, type="y") # now annual frequency
rates(x=pa, t=as.Date(names(pa)), type="year")

# compute 12-month average rate of change:
pmvg <- convert(x=p, t=t, type="12mavg") # still monthly frequency
rates(x=pmvg, t=t, type="year")

### EXAMPLE 2: Ribe contributions using published HICP data

library(data.table)
library(restatapi)
options(restatapi_cores=1)  # set cores for testing on CRAN
options(hicp.chatty=FALSE)  # suppress package messages and warnings

# import monthly price indices:
prc <- hicp::data(id="prc_hicp_midx", filter=list(unit="I15", geo="EA"))
prc[, "time":=as.Date(paste0(time, "-01"))]
prc[, "year":=as.integer(format(time, "%Y"))]
setnames(x=prc, old="values", new="index")

# import item weights:
inw <- hicp::data(id="prc_hicp_inw", filter=list(geo="EA"))
inw[, "time":=as.integer(time)]
setnames(x=inw, old=c("time","values"), new=c("year","weight"))

# merge price indices and item weights:
hicp.data <- merge(x=prc, y=inw, by=c("geo","coicop","year"), all.x=TRUE)

# add all-items hicp:
hicp.data <- merge(x=hicp.data,
                   y=hicp.data[coicop=="CP00", list(geo,time,index,weight)],
                   by=c("geo","time"), all.x=TRUE, suffixes=c("","_all"))

# ribe decomposition:
hicp.data[, "ribe" := contrib(x=index, w=weight, t=time,
```

```
                              x.all=index_all, w.all=weight_all,
                              type="year", settings=list(method="ribe")), by="coicop"]

# plot annual change rates over time:
plot(rates(x=index, t=time, type="year")~time,
     data=hicp.data[coicop=="CP00",],
     type="l", ylim=c(-2,12))

# add contribution of energy to plot:
lines(ribe~time, data=hicp.data[coicop=="NRG"], col="red")
```

`spec.aggs` *Special aggregates*

Description

This dataset contains the special aggregates and their composition of COICOP codes valid since 2017.

Usage

```
# special aggregates:
spec.aggs
```

Format

A data.table with the following variables.

- `code`: the special aggregate code
- `name_[en|fr|de]`: the special aggregate description in English, French, and German
- `composition`: a list of the COICOP product codes forming the special aggregate

Author(s)

Sebastian Weinand

Examples

```
# subset to services:
spec.aggs[code=="SERV", composition[[1]]]
```

`tree` *Derive and fix COICOP tree*

Description

Function `tree()` derives the COICOP tree at the lowest possible level. In HICP data, this can be done separately for each country and year. Consequently, the COICOP tree can differ across space and time. If needed, however, specifying the argument `by` in `tree()` allows to merge the COICOP trees at the lowest possible level, e.g. to obtain a unique composition of COICOP codes over time.

Usage

```
tree(id, by=NULL, w=NULL, flag=FALSE, settings=list())
```

Arguments

`id`	character vector of COICOP codes.
`by`	vector specifying the variable to be used for merging the tree, e.g. vector of dates for merging over time or a vector of countries for merging across space. If `by=NULL` (the default), no merging is performed.
`w`	numeric weight of `id`. If supplied, it is checked that the weights of children add up to the weight of their parent (allowing for tolerance `w.tol`). If `w=NULL` (the default), no checking of weight aggregation is performed.
`flag`	logical specifying the function output. For `FALSE` (the default), a list with the codes defining the COICOP tree at each level. For `TRUE`, a logical vector of the same length as `id` indicating which elements in `id` define the lowest level of the COICOP tree.
`settings`	list of control settings to be used. The following settings are supported:

- `chatty` : logical indicating if package-specific warnings and info messages should be printed or not. The default is `getOption("hicp.chatty")`.
- `coicop.version` : character specifying the COICOP version to be used for flagging valid COICOP codes. See `coicop` for the allowed values. The default is `getOption("hicp.coicop.version")`.
- `all.items.code` : character specifying the code internally used for the all-items index. The default is taken from `getOption("hicp.all.items.code")`.
- `coicop.bundles` : named list specifying the COICOP bundle code dictionary used for unbundling any bundle codes in `id`. The default is `getOption("hicp.coicop.bundles`
- `max.lvl` : integer specifying the maximum depth or deepest COICOP level allowed. If `NULL` (the default), the deepest level found in `id` is used.
- `w.tol` : numeric tolerance for checking of weights. Only relevant if `w` is not `NULL`. The default is 1/100.

Details

The derivation of the COICOP tree follows a top-down-approach. Starting from the top level (usually the all-items code), it is checked if

1. the code in `id` has children,
2. the children's weights correctly add up to the weight of the parent (if `w` provided),
3. all children can be found in all the groups in `by` (if `by` provided).

Only if all three conditions are met, the children are stored and further processed. Otherwise, the parent is kept and the processing stops in the respective node. This process is followed until the lowest level of all codes is reached.

If `by` is provided, function `tree()` first subsets all codes in `id` to the intersecting levels. This ensures that the derivation of the COICOP tree does not directly stops if, for example, the all-items code is missing in one of the groups in `by`. For example, assume the codes `(00,01,02,011,012,021)` for `by=1` and `(01,011,012,021)` for `by=2`. In this case, the code `00` would be dropped internally first because its level is not available for `by=2`. The other codes would be processed since their levels intersect across `by`. However, since `(01,02)` do not fulfill the third check, the derivation would stop and no merged tree would be available though codes `(011,012,021)` seem to be a solution.

Value

Either a list (for `flag=FALSE`) or a logical vector of the same length as `id` (for `flag=TRUE`).

Author(s)

Sebastian Weinand

See Also

`unbundle`, `parent`

Examples

```
### EXAMPLE 1

# derive COICOP tree from top to bottom:
tree(id=c("01","011","012","0111","0112")) # (0111,0112,012) at lowest level

# or just flag lowest level of COICOP tree:
tree(id=c("01","011","012","0111","0112"), flag=TRUE)

# still same tree because weights add up:
tree(id=c("01","011","012","0111","0112"), w=c(0.2,0.08,0.12,0.05,0.03))

# now (011,012) because weights do not correctly add up at lower levels:
tree(id=c("01","011","012","0111","0112"), w=c(0.2,0.08,0.12,0.05,0.01))

# again (011,012) because maximum (or deepest) coicop level to 3 digits:
tree(id=c("01","011","012","0111","0112","01121"),
     w=c(0.2,0.08,0.12,0.02,0.06,0.06),
```

```
     settings=list(max.lvl=3))

# coicop bundles are used if their underlying codes are not all present:
tree(id=c("08","081","082","082_083"), w=c(0.25,0.05,0.15,0.2))
# (081,082_083) where 082 is dropped because 083 is missing

# merge (or fix) coicop tree over groups:
tree(id=c("00","01","011","012", "00","01","011"), by=c(1,1,1,1,2,2,2))
# 01 is present in both by=(1,2) while 012 is missing in by=2

### EXAMPLE 2: Working with published HICP data

library(data.table)
library(restatapi)
options(restatapi_cores=1) # set cores for testing on CRAN
options(hicp.chatty=FALSE) # suppress package messages and warnings

# load HICP item weights:
coicops <- hicp::data(id="prc_hicp_inw",
                      filter=list(geo=c("EA","DE","FR")),
                      date.range=c("2005", NA))
coicops <- coicops[grepl("^CP", coicop),]
coicops[, "coicop":=gsub("^CP", "", coicop)]

# derive seperate trees for each time period and country:
coicops[, "t1" := tree(id=coicop, w=values,
                       flag=TRUE, settings=list(w.tol=0.1)), by=c("geo","time")]
coicops[t1==TRUE,
        list("n"=uniqueN(coicop),           # varying coicops over time and space
             "w"=sum(values, na.rm=TRUE)),  # weight sums should equal 1000
        by=c("geo","time")]

# derive merged trees over time, but not across countries:
coicops[, "t2" := tree(id=coicop, by=time, w=values,
                       flag=TRUE, settings=list(w.tol=0.1)), by="geo"]
coicops[t2==TRUE,
        list("n"=uniqueN(coicop),           # same selection over time in a country
             "w"=sum(values, na.rm=TRUE)),  # weight sums should equal 1000
        by=c("geo","time")]

# derive merged trees over countries and time:
coicops[, "t3" := tree(id=coicop, by=paste(geo,time), w=values,
                       flag=TRUE, settings=list(w.tol=0.1))]
coicops[t3==TRUE,
       list("n"=uniqueN(coicop),          # same selection over time and across countries
             "w"=sum(values, na.rm=TRUE)),  # weight sums should equal 1000
        by=c("geo","time")]
```

Index