Package ‘Haver’

August 17, 2018

Type Package

Title Access Haver Analytics Databases from R

Version 2.4

Date 2018-08-20

Author Haver Analytics <tech@haver.com>

Maintainer Haver Analytics <tech@haver.com>

Description Provides access to Haver Analytics databases from R

License Provided under DLX Subscription Agreement
according to the Agreement Definition of MATTER

LazyLoad no

LazyData no

Depends R (>= 3.0.2)

Imports stats

Suggests zoo

R topics documented:

`Haver_package` *Introduction to the* **Haver** *Package*

Description

This document provides an introductory overview of functions that allow you to access data stored in Haver Analytics databases from within R. It also defines terminology and conventions that are relevant for the entire documentation of the package.

The objects and functions of the **Haver** package are designed to let you conveniently and efficiently access data and metadata of Haver Analytics databases. This is their only task: They are not designed for data analysis. Once you are past the query stage and you want to do data analysis, convert the data objects created by database queries into `R` objects that are more suitable for this purpose. To do so, you can use coercion functions provided by the **Haver** package.

Getting to Know the Haver Package Quickly

The basic operations of **Haver** data queries are easy and can be learned quickly. In a nutshell, the functions that access Haver Analytics databases are called `haver.data` and `haver.metadata`. Both take as arguments specifications for databases and time series codes to be retrieved. The data and metadata are returned in objects `HaverData` and `HaverMetaData`, respectively. Data and metadata are linked closely: A time series retrieval results in a `HaverData` object and this object internally contains the corresponding metadata, which can be accessed at any time. Conversely, a `HaverMetaData` object resulting from a metadata query can be used as an input argument to `haver.data` in order to query the corresponding time series data.

This help entry and some other help entries for the **Haver** package are somewhat lengthy, but this is not because usage is complicated. Rather, they provide much detail that can be skipped by beginning users but that may become useful at a later stage.

If you want to get a very quick introduction, read (sub-)sections 'Old and New Haver Functions', 'Where to Go from here', and 'Examples' of this help entry, and do take note of the other section and subsection headings so you know where to look up information at a later point.

The other sections than the ones just mentioned define terminology and/or provide information for efficiently gaining a deeper understanding of the package. If you decide to only skim through them for now, we recommend that you do not postpone a more thorough reading for too long.

Preliminaries

DLX, Access to Databases, Operating System: 'Haver' is short for the company name Haver Analytics, whereas 'DLX' refers to a software bundle that is distributed by Haver Analytics ('DLX' is short for 'Data Link Express'). DLX comprises all software components that update and provide access to Haver Analytics databases. For the practical purposes of this help file this distinction does not matter and you can read 'Haver' and 'DLX' interchangeably. This package uses the term 'Haver' mostly when referring to databases, series codes, etc., related to Haver Analytics or the DLX software.

Accessing Haver databases requires that you have access to Haver databases on a local or on a network drive. You do not need to have any Haver desktop client software installed.

The **Haver** package is only available on Windows operating systems.

Old and New Haver Functions: As you will see, the names of the featured set of functions provided by the **Haver** package start with 'haver'. Previous versions of this package contained functions whose names started with 'dlx'. To distinguish older from newer functions, help documents will refer to the old functions as the "legacy 'dlx' family of functions" and to the new functions as the "recommended 'haver' family of functions".

As this terminology implies, usage of the legacy 'dlx' family of functions is discouraged since the recommended 'haver' family of functions provides improved functionality. However, the legacy 'dlx' family of functions is still available in the current **Haver** package for backward compatibility reasons. Any existing scripts based on these functions will continue to work without modification. If you are already familiar with the workings of the legacy 'dlx' family of functions you can jump to Haver_legacy which will tell you about the most important differences between the two function families.

Unless otherwise noted, any **Haver** help document that is not explicitly geard towards describing the legacy 'dlx' family of functions will solely be concerned with the recommended 'haver' family of functions. For example, the statements of the following section on conventions of functions and help files of the **Haver** package only apply to the recommended 'haver' family of functions, as do statements of all subsequent sections in this help file.

Conventions of Haver Functions and Objects and of Haver Help Files:

Usage of Terms 'code', 'symbol', 'variable', etc.: Codes of Haver time series are referred to as *series codes*, *Haver codes*, or *Haver series codes*. The time series themselves are referred to as *Haver series* or *Haver time series*.
To unambigously identify a Haver time series, you must indicate the series code and the database name. The term series code may or may not imply this full specification. If it should be stressed that a full specification is necessary, the word *full* is prepended to 'Haver code', 'series code', etc. If this is done using the form *database name - colon - series code*, the specification is said to be in *database:seriescode* format. The *database:seriescode* format within R corresponds to the *seriescode@database* format that is e.g. used in the DLX add-in for Excel. Conversely, if it should be stressed that a series code specification does NOT include the database name, the word *regular* is prepended to 'Haver code', 'series code', etc.
The following table summarizes this information, with examples of full series codes given in *database:seriescode* format.

term	**example**
Haver code, series code, Haver series code	gdp, usecon:gdp
full Haver code, full series code, etc.	usecon:gdp
regular Haver code, etc.	gdp

Side note: If a regular series code exists in two databases, it usually refers to the same concept, potentially with differences in frequencies (e.g. 'daily:ffed' and 'weekly:ffed'). Cases where a regular series code exists in multiple databases and refers to different concepts are very rare.

The term 'variable' is mostly avoided since it can mean many different things (a variable of a statistical model, a variable within computer code, an R symbol, Haver codes, etc.).
'variables' of R code are called *symbols* or R *symbols*. For example, x and y of the expression
`> x <- y + 7`

are R symbols.

Casing: Arguments to **Haver** functions are not case-sensitive (if the argument is of type character). In particular, you can specify Haver series codes in upper case or lower case or any mixture thereof.
Function names of **Haver** functions are always lower case (e.g. `haver.data`). Names of objects of the **Haver** package are in 'UpperCamelCase', i.e. in lower case with the starting letter of a word capitalized (e.g. `HaverMetaData`).
Almost all symbols and field names returned by **Haver** functions are lower case. In particular, Haver series codes or database names returned by **Haver** functions are always lower case. The field names of a HaverMetaData object are also lower case (see `HaverMetaData`). The exception to the 'return character values in lower case' convention of **Haver** functions are the values of most fields of HaverMetaData objects.

Example Code Symbol Names: The most important objects of the **Haver** package are `HaverData` and `HaverMetaData` objects, to be described below and in other help files. Whenever an example R code statement in a help file assigns such an object to a symbol, symbols 'hd' and 'hmd' are used, possibly with suffixes that are easily understood within the context ('hd_1', 'hd_2', 'hmd_d', etc.). Objects of type `HaverErrorReport` are called 'her'. Any other R symbols that are defined in example code start with 'ex_'.
All of these symbols are avaiable for interactive exploration after `example` concludes. Note that executing the 'Examples' section will overwrite any symbols with identical names in the global workspace. If you want to prevent this, use the `local=TRUE` option of `example`.

Date Format: Dates are always in the format yyyy-mm-dd. For example, 24 March 2014 is denoted 2014-03-24.

Missing Values: Missing values for time series data are frequently referred to as 'NAs'. In data sets missing values are recorded as (numeric) NAs.

Reinstalling the Haver Package

If you want to uninstall the **Haver** package, simply issuing

```
> remove.packages("Haver")
```

may result in an incomplete deinstallation, which may cause problems for reinstalling the package (you may have to quit R and manually delete the remnants of the old installation). If the **Haver** package is attached to your R search path, be sure to issue

```
> detach("package:Haver", unload=TRUE)
```

before calling `remove.packages`.

Preparatory Steps for R Code Examples

Example Databases: The **Haver** package comes with three example databases that are used frequently in the 'Examples' section of **Haver** help entries: 'HAVERD', 'HAVERW', and 'HAVERMQA'. They contain daily, weekly, and monthly/quarterly/annual data, respectively. These databases are subsequently referred to as the 'example databases'.

Do not use data contained in example databases for analysis. They are shipped for demonstration purposes only. The data contained in these databases is several years old.

The Haver databases that your institution subscribes to and that you can use in analysis are referred to as the 'actual databases' or 'actual Haver databases'. Keep in mind that, while the example databases contain only between 3 and 161 series, actual Haver databases contain thousands or even hundreds of thousands of series.

In rare cases, help files will refer to series in actual databases. Such references are only made outside of 'Examples' sections.

Setting the Haver Database Path to Actual and Example Databases: Before Haver database queries can be executed the correct database directory path has to be set. This is done with `haver.path`. In most cases you will not have to do this as `haver.path` tries to set the correct path to the actual Haver databases automatically.

The path to the example databases is different than the path to the actual databases. `haver.path` provides a convenient syntax shortcut for setting the path to example databases and for restoring the path to the actual databases. This is done by the statements

```
> haver.path("examples")
> haver.path("restore")
```

which you will find at the beginning and at the end (respectively) of each 'Examples' section of **Haver** help entries. As an additional safeguard there is a statement

```
> haver.path()
```

at the very top of each 'Examples' section which displays the path setting that was in place before the 'Examples' section was entered.

Depending on the configuration of R and of DLX on your machine, it may occur that calling `haver.path("examples")` and `haver.path("restore")` does not work. In this case, executing the examples using `example` will not be possible. However, you can still set the path to the example databases manually and copy & paste statements from the 'Examples' sections to the R command prompt.

To set the path to the example databases manually, first take note of the path setting to the actual databases, which is displayed by

```
> haver.path()
```

Then, execute

```
> haver.path(set=EXAMPLE_PATH)
```

where `EXAMPLE_PATH` is the subdirectory 'dat' of the installation directory of the Haver package. For example, if you had installed the Haver package to 'c:\r\r-3.0.2\library', you would have to execute

```
> haver.path(set="c:\\r\\r-3.0.2\\library\\Haver\\dat")
```

Note the use of a double backslash as directory separator. Once you are done looking at the examples, manually set the path back to the actual Haver databases.

Where to Go from Here

First, read and/or execute the 'Examples' section below (to execute the examples, use the R function `example`). It provides introductory examples to give you a first idea on how to use functions of the **Haver** package.

You can then go on to a more detailed exposition of the functions `haver.metadata` and `haver.data`. The help files for the two functions contain a lot of detail, too much detail probably for a beginning **Haver** user. You absolutely do not have to read everything, but do take note of the (sub-) section headings in these help files so you know where to look up information at a later point. You should read the files thoroughly enough to be able to understand the 'Examples' sections of both files.

You can then examine the three objects involved and learn how to manipulate them: `HaverMetaData`, `HaverData`, and `HaverErrorReport`. Along the way, you will come across functions `haver.codelists`, `haver.datamd`, and `haver.setalias`.

Finally, Haver_advanced provides an extended real-world data query example with further illustrations of how you can use the **Haver** interface in practice.

If at any point you have trouble retrieving data because of path settings, look at `haver.path`.

The **Haver** package imposes artificial limits on the size of data queries. These limits may never become relevant for you. If they do, you can change the settings for these limits. Should a **Haver** routine notify you that a query is not possible because of query size limits, `haver.limits` provides information on the customization of query limits.

Author(s)

Haver Analytics <tech@haver.com>

Maintainer: Haver Analytics <tech@haver.com>

See Also

`haver.path`
`haver.data`
`haver.metadata`
`HaverData`
`HaverMetaData`
`HaverErrorReport`
`haver.codelists`
`haver.datamd`
`haver.setalias`
`haver.limits`
`Haver_advanced`
`Haver_legacy`

Examples

```
## This section provides introductory examples.
## It's purpose is to give you an overview of the Haver package.
## Detailed explanations are skipped at this point.

## Display current path setting:
haver.path()

## Set the database path to the directory of the example databases:
haver.path("examples")

## This directory contains three databases: HAVERD, HAVERW, and HAVERMQA.
## Function 'haver.metadata' accesses metadata information of Haver databases.
## We use it here to get an overview of what is in the databases.
haver.metadata(database="haverd")
haver.metadata(database="haverw")

## Actual Haver Analytics databases have thousands or even hundreds of thousands
## of series. In such and other cases, assigning the output to a symbol
## is appropriate. As usual, this prevents printing:
hmd_mqa <- haver.metadata(database="havermqa")
```

```
## Symbol 'hmd_mqa' now holds a HaverMetaData object.
## You can view its contents in the viewer:
## Not run:
View(hmd_mqa)

## End(Not run)

## Let's pull in three exchange rate series from 'HAVERD'.
## 'haver.data' is the function to retrieve time series data.
## We specify a starting date in order to limit the output.
hd_1 <- haver.data(codes=c("fxtwb", "fxtwm", "fxtwotp"),
                   database="haverd", start=as.Date("2012-02-15", format="%Y-%m-%d"))

## 'hd_1' is of class 'HaverData':
hd_1

## The metadata has automatically been saved within this object.
## We can access it using function 'haver.datamd':
haver.datamd(hd_1)

## Alternatively, we can assign the HaverMetaData object to a symbol
## and then customize the output:
hmd_1 <- haver.datamd(hd_1)
print(hmd_1, fields=c("code", "descriptor"), desc.length=60)

## We can pull in data from more than one database using the
## 'database:seriescode' format in the 'codes' argument. Note that in the
## following query the daily series gets aggregated to weekly frequency.
## The frequency of the resulting data set is weekly.
hd_2 <- haver.data(codes=c("haverD:fxtwb", "haverW:farbn"),
                   start=as.Date("2012/01/01", format="%Y/%m/%d"))
hd_2

## Next, we pull in data from the entire 'HAVERMQA' database. To do this,
## we use the HaverMetaData object hmd_mqa which we have created earlier
## and feed it into the data retrieval.
## Since the lowest frequency of the queried series is annual, series with higher
## frequencies get aggregated. 'haver.data' also informs us that two series
## cannot be aggregated. They get dropped from the query.
hd_mqa  <- haver.data(codes=hmd_mqa)

## You can look at the data set in the viewer:
## Not run:
View(hd_mqa)

## End(Not run)

## You can specify a higher frequency if you want, say, quarterly.
## Then 'haver.data' drops annual series from the query:
hd_mqa2 <- haver.data(codes=hmd_mqa, freq="q")

## To summarize statistical information, or for plotting, you must convert
## the HaverData object to another object, for example, an R 'ts' object:
ex_ts <- as.ts(hd_mqa2)
plot(ex_ts[,1:6])

haver.path("restore")
```

`haver.data`	*Access Haver Analytics Time Series Data*

Description

`haver.data` queries time series data contained in Haver Analytics databases. The data are returned either as a `HaverData` object, as a `data.frame`, as a `ts` object, or as a `zoo` object.

`haver.hasfullcodes` is a small helper function.

This help entry assumes that you are familiar with Haver_package.

Usage

```
haver.data(codes, database, start, end, frequency,
           rtype = "HaverData", aggmode = "strict",
           eop.dates = FALSE, limits = TRUE, cbreak = 500)

haver.hasfullcodes(x)
```

Arguments

`codes`	is either a character vector or it is an object that fulfills the conditions of `haver.hasfullcodes`. If a character vector is supplied, the elements can be regular series codes or additionally supply the database where the code resides, in which case the elements must be specified in *database:seriescode* format. Examples of character vector elements are ‘gdp’ and ‘usecon:gdp’. Haver series codes may also be supplied through an object that passes `haver.hasfullcodes`. The only difference of usage of the `codes` argument in comparison to `haver.metadata` is that it is NOT optional here (see section ‘Details’ below).
`database`	is a character string. This argument is optional and specifies a default Haver database (excluding the path to the database, and excluding any file extension). If this argument is omitted, all elements of ‘codes’ must be in *database:seriescode* format. Otherwise the function errors out.
`start`	an R Date object. Specifies the starting date of the query. Even though `start` is supplied as a Date object, it is interpreted within the context of the frequency of the data set. This is an important point. See section ‘Specifying Query Starting and Ending Periods’ below for details.
`end`	an R Date object. Specifies the ending date of the query. Analogous comments as for `start` apply.
`frequency`	a string. Specifies the target frequency of the query. One of ‘daily’, ‘weekly’, ‘monthly’, ‘quarterly’, and ‘annual’, or any abbreviations thereof, down to a single character.
`rtype`	a character string. Specifies the return type of the function. One of ‘HaverData’ (the default), ‘data.frame’, ‘ts’, and ‘zoo’.
`aggmode`	the aggregation mode of the query. One of ‘strict’, ‘relaxed’, and ‘force’. See section ‘Aggregation Modes’ below for details.

`eop.dates`	logical `TRUE/FALSE`. By default (`eop.dates=FALSE`), elements of the time vector are displayed as periods (e.g. '1988-Q1'). If `eop.dates=TRUE`, end-of-period dates are used instead (e.g. '1988-03-31').
`limits`	logical `TRUE/FALSE`. If `limits=FALSE`, `haver.data` ignores settings regarding query limits.
`cbreak`	is an integer. This option is rarely needed and specifies the number of codes after which each C-level function of the **Haver** package checks for user interrupts. If you want to switch off checking for interrupts, specify a very large value, such as `cbreak=1e8`.
`x`	a `HaverMetaData` object or a `data.frame` that fulfills certain conditions. See section 'details' below.

Details

Both `haver.data` and `haver.metadata` take arguments `codes` and `database`. They allow you to conveniently specify Haver Analytics series codes. Usage of these arguments in the two functions is very similar, with just one exception: If the argument `codes` is omitted in `haver.metadata`, the function interprets the request as 'all series codes from database *database*'. In `haver.data`, such an interpretation is not made, and consequently the argument `codes` is not optional there.

The order of series records within the returned object corresponds to the order of series in the input argument `codes`.

As is generally the case with **Haver** functions, (character) values of arguments supplied are not case-sensitive.

An object that passes the test of `haver.hasfullcodes` can be used as input argument `codes` instead of a character vector. `haver.hasfullcodes` returns a logical 'TRUE' if an object satisfies the following conditions:

- the `class` is `HaverMetaData` or
- the `class` is `data.frame` and in addition the data.frame's first two columns
  - are named 'database' and 'code'
  - have entries of type 'character'. Note that entries of type 'factor' are not allowed.
  - have no empty entries
  - have only alphanumeric entries
  - have at least one entry (the `data.frame` has at least one row).

See the 'Examples' section below for more information on this technique.

Value

The return value of `haver.data`, by default, is a `HaverData` object. You can request different return types using the function argument `rtype`. Besides 'HaverData', admitted values are 'data.frame', 'ts', and 'zoo'. 'ts' will only return a `ts` object if the data set frequency is monthly, quarterly, or annual. Otherwise a warning is issued and a `HaverData` object is returned.

If you specify any other return type than the default, no metadata will be available in the returned objects. Series metadata can only be stored in HaverData objects.

In the case of invalid series code and/or database specifications, a `HaverErrorReport` is returned.

`haver.hasfullcodes` returns a logical `TRUE/FALSE`.

Temporal Aggregation

Determination of the Data Set Frequency: If option `frequency` is not specified the data set frequency corresponds to the lowest frequency of the series in the query. For example, querying a daily and a monthly series will result in a monthly data set, with daily values aggregated to monthly ones.

If you do specify option `frequency`, then this setting takes precedence over the rule laid out in the previous paragraph. For example, when querying a daily and a monthly series and specifying `frequency="weekly"`, then the daily series gets aggregated to weekly values, whereas the monthly series gets dropped from the query (**Haver** supports temporal aggregation, but not temporal disaggregation).

Temporal Aggregation Modes: A **Haver** aggregation mode can either be 'strict', 'relaxed', or 'force'. You can invoke a particular aggregation mode by setting the argument `aggmode` to one of these three possible values. The default is 'strict'.

Note the difference between the argument `aggmode` and the metadata field 'aggtype'. Examples for 'aggtype's are 'AVG' and 'SUM' (see section 'Metadata Fields' in `HaverMetaData`). They define which calculations are carried out when a series is aggregated. `aggmode`s determine how these calculations behave in the presence of NAs. `aggmode` is set for a particular query, whereas 'aggtype' is a fixed setting that an individual series has.

For many queries, the default aggregation mode 'strict' is the correct one to use. However, temporal aggregation of daily and weekly data to a lower frequency should be done under 'relaxed' or 'force' in most cases. Most economic daily time series, for example, have about 10 NAs per year because of holidays. Whenever a daily series with 'aggtype'=AVG or 'aggtype'=SUM has a missing value, its aggregated monthly value will be NA also, under the default aggregation mode 'strict'. As a result, aggregating such daily series to monthly series will produce NAs mostly. You can remedy this by specifying `aggmode="relaxed"` or `aggmode="force"`.

The following table describes the behavior that obtains with each combination of `aggmode` and 'aggtype'. The term 'aggregated span' used in the table stands for a time span *in the original frequency* that aggregates to one observation of the target frequency. For example, 1973-Jan - 1973-Mar is an aggregated span for quarterly aggregation to 1973-Q1.

aggtype	**aggmode**	**rule for returned value**
EOP	strict	Returns the value of the last period in the aggregated span. If this value is missing, it returns missing.
	relaxed	Returns the last nonmissing value of the aggregated span. Returns missing only if all values in the aggregated span are missing.
	force	Same as under "relaxed".
AVG	strict	Does not return an aggregated value as soon as one value in the aggregated span is missing.
	relaxed	Calculates an aggregated value as soon as one value in the aggregated span is nonmissing.
	force	Same as under "relaxed".
SUM	strict	Does not return an aggregated value as soon as one value in the aggregated span is missing.

relaxed	Same as under "strict".
force	Calculates an aggregated value as soon as one value in the aggregated span is nonmissing.

Specifying Query Starting and Ending Periods

Start and end dates have to be supplied as R `Date` object. If the start date is after the last observation of the requested series an error occurs and similarly for the end date. To minimize the possibility of erroneous query specifications start and end dates are restricted to the years 1900-2099.

If argument `enddate` is after the last observation of the implied data set, or if argument `startdate` is before the first observation of the implied data set, observations in the data set returned are padded with `NA`s accordingly. In other words, time periods of the data set will always start with `startdate`, if specified, and will always end with `enddate`, if specified.

Start and ending dates are supplied to `haver.data` as R `Date` objects (i.e. as calendar dates), but the interpretation of what these dates mean depends on the frequency of the data set returned. For example, if `start=as.Date("2012-01-01", format="%Y-%m-%d")` and the data set is returned in annual frequency, it is taken to mean '2012'. This can be confusing if the frequency of the returned data set is not known in advance. Let's say that you think you have a list of monthly series codes but there is actually one or more annual codes among them, so the data get aggregated to annual frequency. If you specified `start` as above and in addition `end=as.Date("2012-05-31", format="%Y-%m-%d")`, both dates get interpreted as '2012'. `haver.data` will return a value for 2012, and it will use all data points from 2012-Jan to 2012-Dec to calculated this value. If there had in addition be daily series in the query, an aggregated value for 2012 would be calculated for them too, based on all data points from 2012-Jan-01 to 2012-Dec-31. To summarize:

- Start and ending periods are supplied as R Date objects (i.e. as calendar dates), but are interpreted as periods within the context of the frequency of the data set returned.
- Any aggregation that takes place uses all underlying periods of the 'aggregated span', with no exceptions. The rules described in section 'Aggregation Modes' apply, with no exceptions.

For daily and weekly data sets it may happen that the dates of the data retrieved are slightly (by a few days) outside of the range specified by arguments `start` and `end`.

When Data Cannot Be Retrieved

In some instances, `haver.data` cannot retrieve data for a particular series or for a particular query. Dependening on the reason why this happens, `haver.data` responds in two different ways: It may either drop some series codes from the query and retrieve data for the other series codes, or it may retrieve no data at all and instead return a `HaverErrorReport` object. The paragraphs below state in which cases each behavior occurs and what you can do to fix up the query.

Dropping of Series: There are three instances where a query is successful (i.e. a `HaverData` object is returned) but some of the requested series get dropped from the query:

1. A series does not have any data points. It only occurs if the series has no data point in the database. This is very rarely the case. A series does not get dropped if you specify starting and ending dates in such a way that there are no valid data points in this time span. The series will solely consist of NAs in this case.

2. A series cannot be aggregated to the data set frequency. This occurs, for example, when a series holds some kind of fraction (e.g. the trade balance as a percentage of GDP). Then the aggregated value cannot simply be calculated as a sum or average. Such series have aggtype=NDF or aggtype=NST, see `HaverMetaData`. In these cases, Haver Analytics frequently provides separate series with aggregated values that are based on correct calculations.
3. A series cannot be disaggregated to the data set frequency. Temporal disaggregation is currently not supported by the **Haver** package.

If at least one series whose data can be retrieved remains, `haver.data` returns a `HaverData` object. If one or more series were dropped, this object contains information on the codes that failed, and on the reasons why this happened. The function `haver.codelists` enables you to get at lists of dropped series codes.

Query Errors: As soon as one series code or one database specified in a query cannot be found, `haver.data` returns a `HaverErrorReport`. This object contains details on what went wrong with the query. The `haver.codelists` method of the `HaverErrorReport` object enables you to get at lists of series codes that could not be found.

Duplicate Codes: Duplicates of *full* series codes are removed automatically. A warning is issued. For example, the query
```
> hd <- haver.data(codes=c("fxtwb", "fxtwb"), dat="haverd"))
```
will return a data set that contains 'haverd:fxtwb' just once.

Duplicate *regular* series codes are allowed. For example,
```
> hd <- haver.data(codes=c("daily:ffed", "weekly:ffed")))
```
will result in a data set that contains two series. This will, however, also be accompanied by a warning, for the following reasons: First, the `View` command will not work correctly anymore. With regards to the present example, it will display the first occurence of 'ffed' twice. Secondly, if you want to assign aliases, you may not use two different aliases for 'daily:ffed' and 'weekly:ffed'. See section 'Assigning Aliases' in `haver.setalias`. Moreover, while coercions to other objects will work correctly you will have multiple time series with identical names. In general, we advice you to avoid duplicate regular series codes within one query.

Daily and Weekly Queries

When querying a weekly series, the dates assigned to data points correspond to the release day-of-week of the series on which the data source publishes new data. When aggregating a daily series to weekly frequency, this release day is set to Friday. When querying multiple weekly series, the dates shown correspond to the release day of the first series in the data set.

If you supply arguments `start` and `end`, it may happen that the starting and ending dates of the data retrieved are slightly (by a few days) outside of the implied time span.

When aggregating daily or weekly series to a lower frequency, familiarity with Haver aggregation modes is necessary. See section 'Aggregation Modes' above.

Query Limits

To provide a safeguard against large data queries that may either take very long or that consume an excessive amount of memory, `haver.data` by default checks settings on limits regarding data queries. There are two limits imposed. The first one specifies the maximum number of series in a query (default value: 5000) and the second one the maximum number of data points (default value:

15 million, which allows e.g. for 1000 daily series that span 50 years). You can query the current state and change these settings using `haver.limits`, whose help entry discusses the issues related to query execution time and memory consumption in more detail.

See Also

`Haver_package`
`haver.path`
`haver.metadata`
`HaverData`
`HaverMetaData`
`HaverErrorReport`
`haver.codelists`
`haver.datamd`
`haver.setalias`
`haver.limits`
`Haver_advanced`
`Haver_legacy`

Examples

```
haver.path()
haver.path("examples")

## The following statements illustrate how you can specify series codes and
## database names. Argument usage of 'codes' and 'database' is identical to
## the statements of the 'Examples' section in 'haver.metadata'.
## All queries below are equivalent. To limit the output, we
## supply a start and an end date.
ex_t1 <- as.Date("2012-01-10", format="%Y-%m-%d")
ex_tN <- as.Date("2012-01-13", format="%Y-%m-%d")
haver.data(cod=c(     "fxtwb",      "fxtwm",      "fxtwotp"), dat="haverd"  , sta=ex_t1, end=ex_tN)
haver.data(cod=c(     "FXTWB",      "FXTWM",      "FXTWOTP"), dat="HAVERD"  , sta=ex_t1, end=ex_tN)
haver.data(cod=c("haverd:fxtwb",      "fxtwm",      "fxtwotp"), dat="haverd"  , sta=ex_t1, end=ex_tN)
haver.data(cod=c("haverd:fxtwb", "haverd:fxtwm", "haverd:fxtwotp")           , sta=ex_t1, end=ex_tN)
haver.data(cod=c("haverd:fxtwb", "haverd:fxtwm", "haverd:fxtwotp"), dat="havermqa", sta=ex_t1, end=ex_tN)

## Querying codes from multiple databases is possible:
ex_tN <- as.Date("2012-01-31", format="%Y-%m-%d")
hd_1  <- haver.data( cod=c("haverD:fxtwb", "haverW:lic", "fcm1") , dat="haverMQA", start=ex_t1, end=ex_tN)

## Querying all codes of a database is possible with 'haver.metadata'
## but not with 'haver.data', so the following produces an error:
## Not run:
haver.data( dat="haverd" )

## End(Not run)

## You can assign the return value of 'haver.data' to a symbol, of course:
ex_tN <- as.Date("2012-01-13", format="%Y-%m-%d")
hd_2  <- haver.data(codes=c("fxtwb", "fxtwm", "fxtwotp") , database="haverd", start=ex_t1, end=ex_tN)
```

```
## If you want to look at the data, you can use 'View':
## Not run:
View(hd_2)

## End(Not run)

## You can also invoke the 'print' method of the object by typing its name:
hd_2

## You can request a lower frequency than daily, e.g. quarterly:
## (at this frequency, we do not have to use options 'start' and 'end'
##  in order to limit output)
hd_3 <- haver.data(codes=c("fxtwb", "fxtwm", "fxtwotp") , database="haverd" , freq="q")
hd_3

## The data points are NAs because the default aggregation mode is 'strict'.
## When aggregating daily data, it is frequently appropriate to use aggregation
## modes that are less restrictive. For details, see section 'Aggregation Modes' above.
hd_4 <- haver.data(codes=c("fxtwb", "fxtwm", "fxtwotp") ,
                   database="haverd" , freq="q", aggmode="relaxed")
hd_4

## Note that the default aggregation mode remains 'strict'.

## Let's see what the database 'HAVERMQA' contains:
hmd_mqa <- haver.metadata(database="havermqa")
## Not run:
View(hmd_mqa)

## End(Not run)

## We pick and retrieve a monthly, a quarterly, and an annual series.
## The monthly and the quarterly series get aggregated to annual frequency.
hd_5 <- haver.data(codes=c("c", "fcm1", "ift"), database="havermqa",
                   start=as.Date("2008-01-13", format="%Y-%m-%d"))
hd_5

## When we explicitly request a quarterly frequency, monthly series get aggregated
## while annual ones get dropped:
hd_6 <- haver.data(codes=c("c", "fcm1", "ift"), database="havermqa",
                   start=as.Date("2008-01-13", format="%Y-%m-%d"), freq="q")
hd_6

## We can investigate what series got dropped by using 'haver.codelists':
haver.codelists(hd_6)

## Every time series data retrieval also retrieves the series metadata and
## attaches it to the 'HaverData' object.
## You can access it using 'haver.datamd':
haver.datamd(hd_6)

## The previous call returned a 'HaverMetaData' object.
## When assigning it to a symbol, you can customize its output:
hmd_6 <- haver.datamd(hd_6)
print(hmd_6, fields=c("code", "frequency"))

## You can retrieve an identical HaverData object that
## displays end-of-period dates:
hd_7 <- haver.data(codes=c("c", "fcm1", "ift"), database="havermqa",
                   start=as.Date("2008-01-13", format="%Y-%m-%d"),
                   freq="q", eop.dates=TRUE)

## If you do not need metadata attached to the data object,
## you can request other return types, e.g. the R object 'ts':
ex_ts <- haver.data(codes=c("c", "fcm1", "ift"), database="havermqa",
                    start=as.Date("2008-01-13", format="%Y-%m-%d"),
                    freq="q", rtype="ts")
ex_ts

## 'ts' object data can directly be graphed:
plot(ex_ts)

## We can use a HaverMetaData object to specify codes for 'haver.data' queries:
hmd_w <- haver.metadata(dat="haverw")
hd_w  <- haver.data(codes=hmd_w)

## We can add codes from other HaverMetaData objects, and manually
## add or delete codes once we are working with data.frames
## whose columns 1 and 2 contain database names and codes:
ex_df_d    <- as.data.frame(haver.metadata(dat="haverd"))
ex_df_dw   <- rbind(ex_df_d[1:2] , as.data.frame(hmd_w)[,1:2])
ex_df_all  <- rbind(ex_df_dw , data.frame(database="havermqa", code="c"))

## If you are not sure whether you can use an object other than a character
## vector as input for the 'codes' argument of 'haver.data', you can check with:
haver.hasfullcodes(ex_df_all)

## Since this is 'TRUE', we can pull in the data:
hd_all <- haver.data(codes=ex_df_all)
print(haver.datamd(hd_all) , fields.disp=c("database", "code", "descriptor"))

haver.path("restore")
```

`haver.metadata` *Access Haver Analytics Time Series Metadata*

Description

`haver.metadata` queries metadata information on time series contained in Haver Analytics databases. The information is returned either as a `HaverMetaData` object or as a `data.frame`.

This help entry assumes that you are familiar with Haver_package.

Usage

```
haver.metadata(codes, database, decode = TRUE, cbreak = 500)
```

Arguments

`codes`	is either a character vector or it is an object that fulfills the conditions of `haver.hasfullcodes`. If a character vector is supplied, the elements can be regular series codes or additionally supply the database where the code resides, in which case the elements must be specified in *database:seriescode* format. Examples of character vector elements are 'gdp' and 'usecon:gdp'. Haver series codes may also be supplied through an object that passes `haver.hasfullcodes`. The only difference of usage of the `codes` argument in comparison to `haver.data` is that it IS optional here (see section 'Details' below).
`database`	is a character string. This argument is optional and specifies a default Haver database (excluding the path to the database, and excluding any file extension). If this argument is omitted, all elements of 'codes' must be in *database:seriescode* format. Otherwise the function errors out.
`decode`	is a logical TRUE / FALSE. This option is rarely used. By default (`decode=TRUE`), metadata information that is numerically encoded will be translated to meaningful character strings. For example, internally the aggregation type 'AVG' (average) is encoded as '1', and the default `decode=TRUE` will re-translate this into 'AVG'.
`cbreak`	is an integer. This option is rarely needed and specifies the number of codes after which each C-level function of the **Haver** package checks for user interrupts. If you want to switch off checking for interrupts, specify a very large value, such as `cbreak=1e8`.

Details

Argument `codes` may be omitted if the argument `database` is supplied, in which case the request is interpreted as 'all series from database *database*'. Note that this is not possible with `haver.data`.

There are three possible combinations of using arguments `codes` and `database`.

1. only supply `codes`. In this case all elements of `codes` have to be in *database:seriescode* format.
2. only supply `database`. This will retrieve metadata for all series contained in *database*.

3. supply both codes and database. Metadata for all elements of codes is queried. If an element of codes is not in *database:seriescode* format, the value of database will serve as the default database.

The order of series records within the returned object corresponds to the order of series in the input argument codes.

As is generally the case with **Haver** functions, (character) values of arguments supplied are not case-sensitive.

Value

An object of class HaverMetaData. They contain various fields with information on series metadata. For a detailed description of these fields, see HaverMetaData.

In the case of invalid series code and/or database specifications, a HaverErrorReport is returned.

Query Errors

As soon as one series code or one database specified in a query cannot be found, haver.metadata returns a HaverErrorReport. This object contains details on what went wrong with the query. The function haver.codelists enables you to get at lists of series codes that could not be found.

See Also

Haver_package
haver.path
haver.data
HaverData
HaverMetaData
HaverErrorReport
haver.codelists
haver.datamd
haver.setalias
haver.limits
Haver_advanced
Haver_legacy

Examples

```
haver.path()
haver.path("examples")

## The following statements illustrate how you can specify series codes and
## database names. Argument usage of 'codes' and 'database' is identical to
## the statements of the 'Examples' section in 'haver.data'.
## All queries below are equivalent.
haver.metadata(cod=c(       "fxtwb",        "fxtwm",        "fxtwotp"), dat="haverd"  )
haver.metadata(cod=c(       "FXTWB",        "FXTWM",        "FXTWOTP"), dat="HAVERD"  )
haver.metadata(cod=c("haverd:fxtwb",        "fxtwm",        "fxtwotp"), dat="haverd"  )
haver.metadata(cod=c("haverd:fxtwb", "haverd:fxtwm", "haverd:fxtwotp")                )
haver.metadata(cod=c("haverd:fxtwb", "haverd:fxtwm", "haverd:fxtwotp"), dat="havermqa")
```

```
## Querying codes from multiple databases is possible:
hmd_1 <- haver.metadata( cod=c("haverD:fxtwb", "haverW:lic", "fcm1") , dat="haverMQA")
hmd_1

## Querying all codes of a database is not possible with 'haver.data',
## but it is with 'haver.metadata':
hmd_2 <- haver.metadata( dat="haverd" )
hmd_2

## Note that when querying an entire database, you must name the
## argument 'database' explicitly.
## Not run:
haver.metadata("haverd")

## End(Not run)

## The statement above produces an error since the first argument of
## 'haver.metadata' is 'codes', but we meant to specify 'database',
## so to accomplish what we want to do we have to run:
haver.metadata(database="haverd")

## When setting the argument 'decode=FALSE', some metadata remain encoded as
## numeric values. In rare cases it may be helpful to look at these numeric
## values. For example:
hmd_w <- haver.metadata(database="haverw", decode=FALSE)
table(hmd_w[["frequency"]])

## shows a frequency count of the series in 'HAVERW'. 9 of those series have
## a release day-of-week of 53, 12 series of 56. These values correspond to
## Wednesday and Saturday, respectively (see 'HaverMetaData'). This can
## be confirmed by looking at one of the fields 'startdate' or 'enddate':
table(weekdays(hmd_w[["enddate"]]))

haver.path("restore")
```

HaverData *HaverData Object*

Description

A HaverData object is the default return type of haver.data. It contains times series data queried from Haver Analytics databases.

If you have not done so yet, have a look at haver.data before reading this help entry.

Usage

```
## S3 method for class 'HaverData'
print(x, data=TRUE)
## S3 method for class 'HaverData'
as.matrix(x)
## S3 method for class 'HaverData'
as.data.frame(x)
## S3 method for class 'HaverData'
as.ts(x)

haver.as.zoo(x)
```

Arguments

x	a HaverData object
data	a logical TRUE / FALSE. If data=FALSE, only summary information on the data set will be printed.

Details

The print method by default displays time series data as well as some data set properties. If you are only interested in the data set properties and if you want to suppress lengthy data output, use option data=FALSE.

If you need to index into the HaverData object, we recommend that you first convert it to a a matrix using as.matrix. If you index directly into a HaverData object, the returned object will be a matrix, not a HaverData object.

Indexing replacement operations ("[<-") are not allowed.

as.matrix, as.data.frame, as.ts, and haver.as.zoo are coercion functions. Note that the conversion function to zoo objects starts with haver.

The time sequence vector will never have gaps, which implies that missing values of time series are always explicitly shown as NAs.

The metadata for all time series of a HaverData object are stored as a HaverMetaData object within the HaverData object. haver.datamd returns a copy of this object.

The `HaverData` object contains information on lists of series codes that have been successfully retrieved and on codes that were dropped from the query. You can use `haver.codelists` to extract these code lists.

You can assign aliases to Haver codes of the `HaverData` object if you prefer to use your own series names in your analysis. Use `haver.setalias` to do this.

Value

`print` prints the time series data and returns the `HaverData` object (invisibly).

`[.HaverData` *returns a matrix*, not a `HaverData` object.

`as.matrix`, `as.data.frame`, `as.ts`, and `haver.as.zoo` return the type indicated in the function name.

Coercion to Other Types

The purpose of existence of the `HaverData` object is to hold data from Haver Analytics databases. It is not designed for data analysis. With `HaverData` objects you cannot

- delete time periods
- add/prepend/append time periods
- delete or add time series
- reorder time series
- merge two or more `HaverData` objects
- plot time series

For the above operations, you either have to re-run the query using `haver.data` or you have to coerce the `HaverData` object to something else and perform the desired operation on that object. The **Haver** package offers coercion functions to four other R types.

Be aware that when using these coercion functions, the metadata does not get copied into the new objects. It is therefore recommended to generate an object *in addition* to the `HaverData` object, as in

```
> hd    <- haver.data(codes="haverd:fxrwb")
> ex_ts <- as.ts(hd)
```

instead of just

```
> ex_ts <- haver.data(codes="haverd:fxrwb", rtype="ts")
```

With the former approach, the data and metadata that you are using in your analysis are much more tractable.

See Also

`Haver_package`
`haver.path`
`haver.data`
`haver.metadata`
`HaverMetaData`
`HaverErrorReport`

haver.codelists
haver.datamd
haver.setalias
haver.limits
Haver_advanced
Haver_legacy

Examples

```
haver.path()
haver.path("examples")

## In the 'Examples' section of 'haver.data' we saw instances where series were
## dropped because temporal aggregation or disaggregation could not be performed:
hmd_mqa <- haver.metadata(database="havermqa")
hd_mqa1 <- haver.data(hmd_mqa)
hd_mqa2 <- haver.data(codes=hmd_mqa, freq="q")

## To determine which codes have been successfully queried and which ones
## got dropped, you can use the 'haver.codelists' function:
haver.codelists(hd_mqa1)
haver.codelists(hd_mqa2)

## Both 'hd_mqa1' and 'hd_mqa2' contain a lot of data so their contents is
## better looked at using 'View' rather than 'print'.
## If you want to just print summary information on the data set but
## suppress the display of time series data points, use the 'data' argument:
print(hd_mqa1, data=FALSE)
print(hd_mqa2, data=FALSE)

## If you need to index into the 'HaverData' object, it is safer to convert it
## to a matrix first:
ex_mat  <- as.matrix(hd_mqa1)
ex_mat2 <- ex_mat[67:77, c("c","cd")]
ex_mat2

haver.path("restore")
```

HaverMetaData *HaverMetaData Object*

Description

A HaverMetaData object is the return type of haver.metadata.

A second use of HaverMetaData objects is to use them as input for haver.data (see Haver_advanced).

If you have not done so yet, have a look at haver.metadata before reading this help entry.

Usage

```
## S3 method for class 'HaverMetaData'
print(x, fields.disp = DEFAULT_FIELDS, desc.length = 30)
## S3 method for class 'HaverMetaData'
dim(x)
## S3 method for class 'HaverMetaData'
x[i]
## S3 method for class 'HaverMetaData'
x[[j]]
## S3 method for class 'HaverMetaData'
as.data.frame(x)
```

Arguments

`x`	a HaverMetaData object
`fields.disp`	a character vector whose elements specify metadata fields to display. For a list of metadata fields and a description of their contents, see section 'Metadata Fields' below. In the syntax list above, DEFAULT_FIELDS stands for `c("database", "code", "startdate", "enddate", "frequency", "numobs", "descriptor")` Use `fields.disp=NA` to display all fields. Use `fields.disp=NULL` or `fields.disp=""` to only display summary information on the data set and to suppress printing of individual records.
`desc.length`	a numeric scalar in the range 2-80. You can also use `desc.length=NA` to set it to the maximum allowed length of 80.
`i`	a numeric or a logical index vector. If logical, the vector is subject to R's vector recycling rule.
`j`	a numeric scalar or string. If numeric, it must be in the range 1 to 18. If string, it must be a valid HaverMetaData field.

Details

`print` is an alternative to `View`. Using `print` you can print metadata information or a subset thereof to the R output window. The print method takes two arguments in addition to the object to be printed. The first one specifies the field list to be printed (see the field list table below). The second one specifies how many characters of the series descriptors should be printed if this field is among the fields to be displayed. Haver metadata field names can be abbreviated, as long as the abbreviation uniquely implies a metadata field name. Note that such abbreviations are not allowed in `[[` indexing operations. See section 'Examples' below for illustrations.

A quick way to determine how many code records are contained within a `HaverMetaData` object is `dim(hmd)[1]`

Single bracket `"["` indexing is restricted to row operations and therefore takes only one argument. Indexing operations allowed are very similar to the ones allowed by `[.data.frame`. One difference is that `HaverMetaData` indexing may be done with or without a trailing 'comma':
```
> hmd[1:3, ]
> hmd[1:3]
```
mean the same thing. With `data.frames`,
```
> df[1:3, ]
> df[1:3]
```
are different operations. The row operations that we are considering correspond to the first statement on `data.frames`.

Single bracket indexing using character index vectors is not allowed for `HaverMetaData` objects. If an indexing operation removes all rows, `NULL` is returned. If an indexing operation results in rows that contain `NA`s (e.g. by out-of-bounds indexes) an error is issued.

Double bracket `"[["` indexing is restricted to column operations.

Indexing replacement operations (`"[<-"` and `"[[<-"`) are not allowed.

A `HaverMetaData` object can be coerced into a `data.frame`. This is useful when you want to perform more elaborate indexing operations on the metadata than allowed by the `HaverMetaData` object. In addition, you have to coerce to `data.frames` if you want to append the contents of two or more `HaverMetaData` objects (see `rbind`). Note that you can use the resulting `data.frame` as an input to `haver.data` (an illustration of this is contained in the 'Examples' section of the `haver.data` help entry).

You can assign aliases to Haver codes of the `HaverMetaData` object if you prefer to use your own series names in your analysis. Use `haver.setalias` to do this.

Value

`print` prints the metadata and returns the `HaverMetaData` object (invisibly).

`dim` returns a numeric vector with two elements indicating the number of code records and the number of metadata fields. The latter is a fixed number for all `HaverMetaData` objects.

`[.HaverMetaData` returns a `HaverMetaData` object or `NULL`.

`[[.HaverMetaData` returns a vector whose type is either character, numeric, Date, or POSIXct.

`as.data.frame` returns a `data.frame`.

Metadata Fields

A HaverMetaData object contains 18 pieces of information for each series code. These pieces of information are referred to as 'fields' in the following.

Side note: In the Haver Analytics data browser 'DLXVG3', a subset of these fields is called 'series parameters' and can be accessed from the 'Tools' menu or through the 'Alt+F10' keyboard shortcut.

field name	**type/class**	**description**
database	character	the database name (excluding the path to the database)
code	character	the Haver series code
alias	character	a user-defined series code
startdate	Date	an R Date object corresponding to the last day of the series starting period
enddate	Date	an R Date object corresponding to the last day of the series ending period
numobs	numeric	number of periods between (and including) startdate and enddate
frequency	char/numeric	the frequency code; one of D, W, M, Q, A (daily, weekly, monthly, quarterly, annual)
datetimemod	POSIXct	a POSIXct object indicating the time the series was last modified
magnitude	numeric	the magnitude of the series (e.g. 3 for series expressed in thousands)
decprecision	numeric	the decimal precision (number of digits after the decimal point)
diftype	numeric	0/1 value: 1 if data can contain or contain non-positive values, 0 otherwise
aggtype	char/numeric	the aggregation type; one of AVG, SUM, EOP, UDF, NST (average, sum, end-of-period, undefined, not set)
datatype	character	one of US$, LocCur, US$/LC, LC/US$, INDEX, Units (where LocCur and LC stand for local currency)
geography1	character	primary Haver geography code
geography2	character	secondary Haver geography code
descriptor	character	the series description (label)
shortsource	character	the abbreviated name of the time series publisher
longsource	character	the full name of the time series publisher

When the `HaverMetaData` object is returned by `haver.metadata`, field 'alias' does not contain any entries since it is not part of the metadata provided by Haver Analytics. Rather, this field allocates space for storing user-defined series code names (aliases). For details, see `haver.setalias`.

Field 'aggtype' controls the temporal aggregation arithmetics for a particular series. `haver.data` performs temporal aggregation according to the values of this field. In some DLX software components it is possible to apply a custom setting to the aggregation type for a particular series. Within R, however, this functionality does not exist. Temporal aggregation is always performed according to the 'aggtype' field entry of a series.

Field 'diftype' is of limited interest since it is used internally in DLX software only. It controls the kinds of series transformations that are allowed for a particular series.

Internally, some fields are stored in an encoded format, i.e. as a numeric value. Changing the default behavior of `haver.metadata` by setting argument `decode=FALSE` will store numeric values in the returned `HaverMetaData` object. In rare cases, it may be useful to access the numeric values. The list of encodings is:

field name	**endoding**
frequency	10=annual, 30=quarterly, 40=monthly, 51=weekly Monday ... 57=weekly Sunday, 60=daily
aggtype	1=AVG, 2=SUM, 3=EOP, 9=UDF, 0=NST

See Also

Haver_package
haver.path
haver.data
haver.metadata
HaverData
HaverErrorReport
haver.codelists
haver.datamd
haver.setalias
haver.limits
Haver_advanced
Haver_legacy

Examples

```
haver.path()
haver.path("examples")

## We start out by pulling in all metadata for 'HAVERMQA'
## and looking at it in the viewer
hmd_mqa <- haver.metadata(dat="havermqa")

## Not run:
View(hmd_mqa)

## End(Not run)

## There are monthly series between rows 75 and 101 which
## we choose to assign to another HaverMetaData object using row-indexing.
## Note that only one index vector is supplied. It refers to rows.
hmd_2 <- hmd_mqa[75:101]

## Double bracket indexing refers to columns.
## Let's do a frequency count:
table(hmd_mqa[["frequency"]])

## More complex subsetting operations have to be done after converting
## to a data.frame.
ex_df_mqa <- as.data.frame(hmd_mqa)

## We are interested in the monthly exchange rate series starting at
## row 79 of 'hmd_mqa'. Let's build a subset of codes that start with 'fx...'.
## We utilize regular expression functions (see the help entry for 'grep')
hmd_3 <- hmd_mqa[grep("fx.*", hmd_mqa[["code"]])]

## Before we can use these series for analysis we must determine
## whether they are quoted in an unambiguous way (units of domestic currency
## per units of foreign currency, or vice versa).
## The 'datatype' metadata field can help in this.
table(hmd_3[["datatype"]])
```

```
## We see that the rates are not quoted uniformly, so we must
## do adjustments after pulling in the data and before doing analyis.

## Let's say that we need exchange rate data going back to 1970.
## We can immediately determine which series fail this criterion:
hmd_4 <- hmd_3[ hmd_3[["startdate"]] > as.Date("1970-12-31", format="%Y-%m-%d") ]
print(hmd_4, fields=c("code","start","desc"), desc.len=70)

## Note that we were able to abbreviate metadata fields names in the above statement.
## This is only possible for the 'print' method, but not for indexing.

## We invert the above filtering criterion to get the series that have the desired
## coverage. Then we feed the returned object as 'codes' argument into 'haver.data':
hmd_5 <- hmd_3[ hmd_3[["startdate"]] <= as.Date("1970-12-31", format="%Y-%m-%d") ]
hd_5  <- haver.data(codes=hmd_5)

## If we want to investigate several full databases, the best way
## is to 'rbind' data.frames:
ex_df_d <- as.data.frame(haver.metadata(dat="haverD"))

ex_df_full <- rbind(ex_df_mqa, ex_df_d)
table(ex_df_full[["database"]], ex_df_full[["frequency"]])

haver.path("restore")
```

HaverErrorReport *HaverErrorReport Object*

Description

A HaverErrorReport object is returned by `haver.data` or `haver.metadata` as soon as one series code or database cannot be found.

Usage

```
## S3 method for class 'HaverErrorReport'
print(x)
```

Arguments

x	a `HaverErrorReport` object.

Details

You can access various lists of series codes pertaining to `HaverErrorReport` objects using the function `haver.codelists`. This aides for example in fixing up queries that have failed.

Value

`print` prints information contained in the object and returns it (invisibly).

See Also

`Haver_package`
`haver.path`
`haver.data`
`haver.metadata`
`HaverData`
`HaverMetaData`
`haver.codelists`
`haver.datamd`
`haver.setalias`
`haver.limits`
`Haver_advanced`
`Haver_legacy`

Examples

```
haver.path()
haver.path("examples")

## The database 'HAVERD' is found but it does not contain series 'wrong1' and 'wrong2':
her <- haver.data(codes=c("wrong1", "wrong2", "fxtwb"), dat="haverd")
```

```
her

## The list of codes that caused the query to fail as well as the list of good codes
## can be accessed with the 'haver.codelists' function.
## Empty lists are returned as zero-length character vectors.
haver.codelists(her)

## If we can do without the codes that could not be found, we can query the
## data of the good codes immediately:
hd <- haver.data(codes=haver.codelists(her)[["codes.found"]])

haver.path("restore")
```

`haver.codelists` *Extract Lists of Haver Series Codes*

Description

`haver.codelists` extracts lists of Haver series codes from **Haver** objects of type `HaverMetaData`, `HaverData`, and `HaverErrorReport`.

Usage

```
haver.codelists(x)
```

Arguments

`x`	an object of type `HaverMetaData`, `HaverData`, or `HaverErrorReport`

Details

The objects of the **Haver** package contain information on various Haver series code lists. For the most part, this information is related to the section 'When Data Cannot Be Retrieved' of `haver.data`. `haver.codelists` accesses this information.

Value

`haver.codelists` always returns a list whose elements are character vectors, possibly of length zero. In turn, the elements of the character vectors are Haver codes in *database:seriescode* format. The returned lists have the following named elements:

argument type	**element name**	**vector contents**
`HaverMetaData`	codes.out	series that hold records in the object
`HaverData`	noobs	series that do not have any valid data points
	noagg	series that cannot be aggregated
	nodisagg	series that cannot be disaggregated
	codes.out	series whose data has been retrieved
`HaverErrorReport`	codes.found	series whose code and database were found
	database.access	series whose database could not be accessed. The most likely reason for this is that the database name is misspelled or that `haver.path` is set incorrectly.
	codes.notfound	series whose series code was not found
	metadata.access	series whose metadata could not be accessed

See Also

`Haver_package`

haver.data
haver.metadata
HaverData
HaverMetaData
HaverErrorReport
haver.datamd
haver.setalias
haver.limits
Haver_advanced
Haver_legacy

Examples

```
haver.path()
haver.path("examples")

## When applying the function to a 'HaverMetaData' object, a list
## of codes that have records in the object is returned
hmd_w <- haver.metadata(dat="haverw")
haver.codelists(hmd_w)

## Applying the function to 'HaverData' objects yields information on
## the successfully retrieved series and on the ones that got dropped:
hmd_mqa <- haver.metadata(dat="havermqa")
hd_q    <- haver.data(codes=hmd_mqa, freq="quarterly")
haver.codelists(hd_q)

## Lastly, one can apply 'haver.codelists' to 'HaverErrorReport' objects.
## The latter are returned by 'haver.data' and 'haver.metadata'
## when there is a fundamental flaw in a query.
ex_codes <- c("wrongdb:code1", "wrongdb:code2", "haverd:wrongcode")
her_hd  <- haver.data(ex_codes)
her_hd
haver.codelists(her_hd)

## The same object is returned from a correspondingly flawed metadata query:
her_hmd <- haver.metadata(ex_codes)
her_hmd
haver.codelists(her_hmd)

haver.path("restore")
```

haver.datamd *Get Metadata of an Existing HaverData Object*

Description

haver.datamd returns the metadata of series contained in a HaverData object.

Usage

```
haver.datamd(x)
```

Arguments

x	A HaverData object

Details

'md' in the function name haver.datamd stands for 'metadata'. Read 'datamd' as 'the data's metadata'.

The metadata for all time series of a HaverData object are stored as a HaverMetaData object within the HaverData object. haver.datamd returns a copy of this object.

Value

haver.datamd returns a HaverMetaData object.

See Also

Haver_package
haver.data
haver.metadata
HaverData
HaverMetaData
HaverErrorReport
haver.codelists
haver.setalias
haver.limits
Haver_advanced
Haver_legacy

Examples

```
haver.path()
haver.path("examples")

## We pull in time series data, and look at the associated metadata afterwards:
hd <- haver.data(codes=c("haverd:fxtwb", "haverw:lic")
                 , freq="quart", aggmode="relaxed")
```

```
hmd <- haver.datamd(hd)
hmd

print(hmd, fields=NA)

haver.path("restore")
```

haver.setalias *Assign Aliases to Haver Series Codes*

Description

haver.setalias enables you to define your own series codes and use them in data analysis. You can assign a user-defined series code (an alias) to any Haver series in a HaverMetaData or HaverData object. The original Haver code also remains recorded.

Usage

```
haver.setalias(x, codes, aliases, clear=FALSE)
```

Arguments

x	a HaverMetaData or a HaverData object.
codes	a character vector specifying Haver series codes that are to be given user-defined aliases (supplied by aliases). Note that codes must not contain database names.
aliases	a character vector. Specifies the aliases that are to be given to the Haver series codes in codes. Must be of the same length as codes.
clear	a logical TRUE / FALSE. If clear=TRUE, haver.setalias removes all information regarding aliases.

Value

The return value is of the same type as the first argument to the function, i.e. either HaverMetaData or HaverData.

Assigning Aliases

User-defined codes are stored in field 'alias'. Since the original Haver codes remain in field 'code', any mapping from an original Haver code to a user-defined alias is recorded in the object.

In order to define your own codes, you have to supply a mapping of original Haver series codes to the codes that you want to use. You do so through the method haver.setalias and its arguments codes (the original Haver series codes) and aliases (the codes that you want to use). The two character vectors must be of the same length. Elements of argument aliases must not exist in field code. For example, if this field contains a code 'yr', you cannot use this string as an alias for any of the codes in the object.

codes may contain codes that are not in the HaverMetaData or HaverData object, and vice versa. Among other things, this means that you can assign your own codes to any subset of the codes of the object. A requirement that always applies is that the mapping you supply must be 1:1, i.e. you may not assign an original Haver series code to two different user-defined codes, and you may not assign the same user-defined code to two different Haver codes.

You may set aliases of series codes incrementally. For example, you can issue a statement that assigns aliases for the first three codes, then issue a statement that assigns aliases for some other codes, etc. Each renaming operation requires, however, that the mapping supplied in `codes` and `aliases` is consistent with the mapping that already exists in the object. As an immediate consequence, you cannot assign an alias to a series code that already has a different alias assigned. If you want to do this, you have to clear all user-defined codes first using the `clear=TRUE` argument. You may wish to save the mapping that exists in the object before you do that. You can do this easily by accessing the 'code' and 'alias' fields of the object using double bracket indexing.

Haver series codes in `HaverMetaData` objects are always recorded in lower case. Any codes that you specify in `aliases` will also be converted to lower case. This may result in mappings that are not allowed. For example, the mapping from `codes=c("code1","code2")` to `aliases=c("mycode","myCode")` will not work since it is not a 1:1 mapping anymore once the elements in `aliases` are translated to lower case.

`haver.setalias` does not modify the `HaverMetaData` object in place but rather returns a modified object. Only statements that have some sort of an assignment like
`> hmd <- haver.setalias(hmd, codes, aliases)`
make sense.

See Also

Haver_package
haver.data
haver.metadata
HaverData
HaverMetaData
HaverErrorReport
haver.codelists
haver.datamd
haver.limits
Haver_advanced
Haver_legacy

Examples

```
haver.path()
haver.path("examples")

## We set up an example data set consisting of three series:
ex_t1 <- as.Date("2010-10-31", format="%Y-%m-%d")
hd <- haver.data(c("haverd:fxtwb", "havermqa:gdph", "havermqa:c")
                 , aggmode="relaxed", start=ex_t1)
haver.datamd(hd)

## We assign aliases for the first two codes:
ex_codes   <- c("fxtwb", "gdph")
ex_aliases <- c("fx"    , "y"   )
hd  <- haver.setalias(hd, codes=ex_codes, aliases=ex_aliases)
hmd <- haver.datamd(hd)
print(hmd, fields=c("code", "alias", "descriptor"))

## We add an alias for the third code:
hd  <- haver.setalias(hd, codes="c", aliases="pce")
```

```
## The aliases are used when coercing the object to another type:
as.matrix(hd)

haver.path("restore")
```

haver.path *Set and Query the Path to Haver Databases*

Description

`haver.path` sets and queries the path to Haver databases. A correct path setting is a prerequisite for data and metadata queries run by `haver.data` and `haver.metadata`

Usage

```
haver.path()
haver.path(set)
haver.path("auto")
haver.path("examples")
haver.path("restore")
```

Arguments

`set`	a character string specifying the directory where Haver databases reside. 'set' can also be set to one of the strings 'auto', 'examples', and 'restore', in which case the database path is automatically set to an appropriate directory.

Details

If `haver.path` is called without arguments, the current setting is returned.

If `set="auto"`, `haver.path` tries to set the proper path to Haver databases automatically. In some instances this may not work, depending on the R configuration and the DLX configuration of your machine.

Alternatively, you can set the database path manually. In this case you should supply the full (absolute) path of the directory where the Haver databases reside as the value for the `set` argument. If you supply a relative file path it is taken to be relative to the working directory. Since this is unlikely to be what you meant, a warning is issued.

`haver.path("examples")` will set the database directory to the directory where the example databases of the **Haver** package are located. Before doing so, `haver.path` will store the existing setting. You can re-enable this setting by issuing `haver.path("restore")`.

Value

When argument `set` is not supplied, a character string that holds the current setting of the database path, and `NULL` otherwise.

R Startup, Loading of Package

When **Haver** gets attached to the R search path, it first tries to restore the database path setting from the last session. If this is not successful, it tries to detect the correct setting from the DLX setup of your machine. If neither of these attempts are successful, you have to set the database path manually.

Setting the Database Path Manually

Note that when manually setting the database path, you must supply directory separators using '\\' or '/'. The function will fail if directories are separated by a single '\'.

If the supplied database path does not exist, an error occurs. If the supplied database path does not contain any Haver Analytics databases, a warning is issued.

Running 'Examples' Sections of Help Entries of the Haver Package

At the top of the examples section in each help entry the database path is set to point to the example databases:
`> haver.path("examples").`
At the end of each examples section, the previous setting is restored:
`> haver.path("restore").`

If you re-set the Haver database path manually after issuing `haver.path("examples")`, a call to `haver.path("restore")` will not change this setting anymore.

If you quit R or unload **Haver** before restoring the setting to the actual Haver Analytics databases, **Haver** will still set the path to the actual Haver Analytics databases when R is invoked (i.e. when **Haver** is attached).

See Also

`Haver_package`
`haver.data`
`haver.metadata`
`HaverData`
`HaverMetaData`
`HaverErrorReport`
`haver.codelists`
`haver.datamd`
`haver.setalias`
`haver.limits`
`Haver_advanced`
`Haver_legacy`

Examples

```
## query the current setting
ex_origpath <- haver.path()

## let 'haver.path' try to determine the correct setting automatically
haver.path("auto")

## set to examples databases
haver.path("examples")

## restore previous setting
haver.path("restore")

## manually restore to setting before running examples section
```

```
if (!is.null(ex_origpath)) haver.path(ex_origpath)

## Not run:
## manually set the database path to an explicitly given directory
haver.path("c:/dlx/data")

## End(Not run)
```

`haver.limits` *Set and Get Limits for Haver Analytics Data Queries*

Description

`haver.limits` handles the settings with regards to the limits of data queries of `haver.data`. Note that these limits have no impact on the metadata queries of `haver.metadata`.

Limits on the size of queries are imposed to protect you from unintentionally conducting very large queries that are problematic in terms of execution time or memory consumption.

Usage

```
haver.limits()
haver.limits(what)
haver.limits(what, value)
```

Arguments

`what`	a character string. One of 'numcodes', 'numdatapoints', and 'default'. `what="numcodes"` sets the maximum allowed number of codes in a query to `value`. `what="numdatapoints"` sets the maximum allowed number of data points in a query to `value`. `what="default"` ignores the value of `value`, if supplied, and restores default values for 'numcodes' and 'numdatapoints', which are 5000 and 15000000 (15 million), respectively.
`value`	a numeric non-negative scalar, specifying the limit for the 'numcodes' or 'numdatapoints' setting.

Details

The values set by `haver.limits` are taken into account by `haver.data`. It will issue an error if a query is beyond what is allowed by query limits, unless you specify its option `limits=FALSE`.

`haver.limits()`, with an empty argument list, does nothing besides returning the current settings.

`haver.limits("default")` restores **Haver** default values.

Any values set by `haver.limits` will remain in effect until the package gets detached from the R search path, or until R is restarted. Restarting R or re-loading the **Haver** package will result in the restoration of default values.

The default values for query limits allow you to conduct queries which are fairly sizeable. The default for the maximum number of codes is 5000. The default for the maximum number of data points is 15 million, which is a little bit more than a daily data set with 1000 time series, covering 50 years of data. Such a data set requires more than 100 megabyte of memory, but the memory necessary for performing the retrieval is several times as high. Before you increase values for limits, make sure these are tenable for your system.

Value

A list with named elements 'numcodes' and 'numdatapoints', containing numeric scalars, indicating the current settings of query limits.

Query Speed and Memory consumption

Large data queries can become problematic with respect to execution time and memory consumption. Whether this is the case depends on the limits of your machine, and it may also depend the speed of your institution's network. The following paragraphs provide recommendations for managing speed and memory issues, in general terms.

Query Execution Time: Should a query take too long, you can interrupt execution as with other R commands. Still, after an interrupted query the memory claimed by R is likely to be larger than before the query. The `gc` command can help to free some or all of the memory up again.

Memory Consumption: When specifying large queries, assessing a tenable level of memory consumption by `haver.data` should not be geared towards the total free memory on your computer, for the following reason:

- The memory consumed by data query operations is likely to be a multiple of the memory consumed by the `HaverData` object returned.
- Commands issued after the query may also consume a large amount of memory. For example, if you follow the recommendation in section 'Coercion to Other Types' in `HaverData` to create an object for data analysis in addition to the `HaverData` object, you are requesting the amount of memory consumed by the Haver data set for a second object. As another example, `haver.setalias` requires a copy operation of an entire `HaverData` object.

On 32-bit systems, memory limits may pose more stringent restrictions on your queries than on 64-bit systems. See `Memory-limits`.

Should a data query consume a lot of memory, there is a good change that you can reclaim some or all of it by using the `gc` command.

See Also

`Haver_package`
`haver.path`
`haver.data`
`haver.metadata`
`HaverData`
`HaverMetaData`
`HaverErrorReport`
`haver.codelists`
`haver.datamd`
`haver.setalias`
`Haver_advanced`
`Haver_legacy`

Examples

```
haver.path()
haver.path("examples")

## haver.limits() returns a list with the current settings for limits. We use
## it here to record these settings in order to be able to restore them later.
ex_storedlimits <- haver.limits()

## We reset the limit on the maximum number of codes allowed in a query:
haver.limits(what="numcodes", value=10)

## 'HAVERW' has 21 codes. The limit we just set does not affect
## metadata retrievals:
hmd <- haver.metadata(dat="haverw")

## Querying the data for 21 codes is not possible with the example settings
## for limits. The following generates an error:
## Not run:
hd <- haver.data(codes=hmd)

## End(Not run)

## A query for 10 codes still works:
hd <- haver.data(codes=hmd[1:10])

## We can also explicitly state that 'haver.data' should ignore limits:
hd <- haver.data(codes=hmd, limits=FALSE)

## With default values, the full 21 code query goes through of course:
haver.limits(what="default")
hd <- haver.data(codes=hmd)

## We restore the limits that were in place before the above examples
## were executed:
haver.limits(what="numcodes"      , value=ex_storedlimits[["numcodes"]])
haver.limits(what="numdatapoints" , value=ex_storedlimits[["numdatapoints"]])


haver.path("restore")
```

Haver_advanced — *Advanced Features and Usage of the* **Haver** *Package*

Description

This help page provides a realistic and complete work-through example of a real-world data query task, touching upon some advanced concepts of usage. In particular, the following topics are illustrated:

- finding appropriate time series
- renaming of series codes
- using `haver.metadata` to determine necessary series transformations
- conversion to non-**Haver** objects for data analysis

An Example Query Task

Let's assume that our task is to perform analysis using the following series:

- US real GDP
- a US interest rate
- real US exports and real US imports of goods and services
- US exchange rates with Canada, the EU, Japan, Switzerland, and the UK

We would like to have quarterly data back to 1970.

In the 'Examples' section below, we will use the **Haver** R package to search Haver databases for series. Please do not forget that Haver's DLXVG3 data browser has powerful search capabilities. However, once you are familiar with the workings of the **Haver** package, its functions may serve as a complement to the tools in DLXVG3 for searching for series.

Author(s)

Haver Analytics <tech@haver.com>

Maintainer: Haver Analytics <tech@haver.com>

See Also

`Haver_package`
`haver.path`
`haver.data`
`haver.metadata`
`HaverData`
`HaverMetaData`
`HaverErrorReport`
`haver.codelists`
`haver.datamd`
`haver.setalias`
`haver.limits`
`Haver_legacy`

Examples

```
haver.path()
haver.path("examples")

## In this example section, we focus on the database 'HAVERMQA'.
hmd_mqa <- haver.metadata(dat="havermqa")

## We have the presumption that our databases include series on US GDP whose
## codes contain the string 'gdp'. We look for regular expression matches
## in the 'code' field of the metadata:
hmd_mqa[grep(".*gdp.*", hmd_mqa[["code"]]), ]

## We can see that US real GDP has series code 'gdph'.
## In order to find interest rates and exports series we apply regular
## expressions again, this time to the 'descriptor' metadata field:
hmd_mqa[ grep(".*Treasury.*", hmd_mqa[["descriptor"]]), ]
hmd_mqa[ grep(".*Export.*"  , hmd_mqa[["descriptor"]]), ]

## To view the full descriptors, you can use 'View'. Here we use 'print':
print( hmd_mqa[ grep(".*Treasury.*", hmd_mqa[["descriptor"]]), ]
       , fields=c("code", "descriptor"), desc.len=80)
print( hmd_mqa[ grep(".*Export.*"  , hmd_mqa[["descriptor"]]), ]
       , fields=c("code", "descriptor"), desc.len=80)

## We opt for series 'fcm1' and 'xh'. We briefly confirm that the
## imports series has an analogous code to the exports series:
hmd_mqa[hmd_mqa[["code"]]=="mh"]

## For exchange rates, we again apply prior knowledge that 'HAVERMQA'
## contains many exchange rates series whose codes start with 'fx...':
hmd_mqa[grep(".*fx.*", hmd_mqa[["code"]]), ]

## We pick the series 'fxcan', 'fxeur', 'fxjap', 'fxsw', 'fxuk'

## We have now determined the 9 series codes for our analysis:
## 'gdph', 'fcm1', 'xh', 'mh', 'fxcan', 'fxeur', 'fxjap', 'fxsw', 'fxuk'
## so we can retrieve the data:
ex_codes <- c("gdph", "fcm1", "xh", "mh", "fxcan",
              "fxeur", "fxjap", "fxsw", "fxuk")
hd_adv  <- haver.data(codes=ex_codes, database="havermqa")
hmd_adv <- haver.datamd(hd_adv)
print(hmd_adv, fields=c("code", "startdate", "descriptor"), desc.length=NA)

## We note that series fxeur starts only in 1999 but keep it in the data set anyway.
## From the labels, we can see that the national accounts series do have the same
## magnitude (billions of chained dollars). If this had not been the case, we would
## have had to harmonize the magnitudes manually.
## We also see that exchange rates are not quoted uniformly. We opt to express
## all exchange rates in units of foreign currency per US dollar. We note that
## later we will have to invert series 'fxeur' and 'fxuk'.
## We use the metadata to confirm our conclusions:
print(hmd_adv, fields=c("code", "magnitude", "datatype"))
```

```
## As another preliminary QA check, we plot the raw data
## using a coercion function:
plot(as.ts(hd_adv))

## At this point we would like to assign series aliases that are more
## convenient to work with, possibly because these are standard codes in our
## own work flow or in the institution we work for:
ex_codes   <- c("gdph", "fcm1", "xh", "mh")
ex_aliases <- c("yr"  , "i"   , "xr", "mr")
hd_adv     <- haver.setalias(hd_adv, ex_codes, ex_aliases)

## You will notice that the series in question have been renamed:
## Not run:
View(hd_adv)

## End(Not run)

## This information has also been recorded in the metadata:
hmd_adv <- haver.datamd(hd_adv)
print(hmd_adv, fields=c("code", "alias", "descriptor"), desc.len=NA)

## In order to start data analysis, we convert our HaverData object to an R 'ts'
## object. The series aliases that we have assigned will be carried forward.
## Our custom series names are the only difference with respect to the
## previous plot:
ex_ts <- as.ts(hd_adv)
plot(ex_ts)

## In the following, we apply a few functions that are outside the Haver package
## in order to illustrate that data analysis is done elsewhere.
## In particular, we cut off part of the sample, invert some exchange rate series,
## generate log-differences and first differences, plot the series,
## and look at summary statistics.

ex_ts                       <- window(ex_ts, start=c(1974,1))
ex_ts[,c("fxeur", "fxuk")] <- 1 / ex_ts[,c("fxeur", "fxuk")]
ex_lognames                 <- c("yr", "xr", "mr", "fxcan",
                                   "fxeur", "fxjap", "fxsw", "fxuk")
ex_ts[,ex_lognames]         <- log(ex_ts[,ex_lognames])
ex_ts                       <- diff(ex_ts)
plot(ex_ts)
summary(ex_ts)

haver.path("restore")
```

Haver_legacy *Haver Legacy Functions*

Description

This document describes legacy functions to access Haver Analytics time series databases from R. It also compares the the legacy functions to the recommended haver familiy of functions. Loading the current Haver package includes the superseded legacy functions. Use of the functions described here should only be made if existing R scripts depend on them. In all other cases, we recommend using a new set of functions (see section 'Old and New Haver Functions' in Haver_package).

Details

This section gives an overview of the legacy dlx family of functions.

One or more time series are queried using dlxGetData and are returned as zoo objects indexed by Date objects. Metadata for one series is queried using dlxGetInfo. In order to use these functions you must first set the location of your Haver databases using dlxSetDbPath. Some basic examples are provided below:

```
## Load Haver library
library(Haver)

## Set the location of your Haver databases.
dlxSetDbPath("C:/DLX/data/")

## Get the series LANAGRA and GDP from database USECON.
dlxGetData(c("lanagra", "gdp"), "usecon")

## Get the series LANAGRA and GDP from database USECON and
## FFED from database WEEKLY
dlxGetData(c("lanagra", "gdp", "weekly:ffed"), "usecon")

## Get the series LANAGRA from database USECON in the date range
## 2000:1 to 2010:12
dlxGetData("lanagra", "usecon", start="2000-01-01", end="2010-12-31")

## Get the series LANAGRA from database USECON, convert to quarterly
## frequency, and get data in the range 2000:1 to 2010:4
dlxGetData("lanagra", "usecon", start="2000-01-01", end="2010-12-31",
  frequency="q")

## Get metadata about series LANAGRA in database USECON
dlxGetInfo("lanagra", "usecon")
```

You can also set the aggregation mode that will be used in calls to dlxGetData by using function dlxSetAggregation. Options are "strict", "relaxed", and "force". See ?dlxSetAggregation for details. To see the current aggregation mode, call `dlxGetAggregation`. The default is "strict".

More examples are provided in the help files of the individual `dlx` functions (see the table below for hyperlinks).

Changes in the Legacy `dlx` Familiy of Functions

Compared to the previous version of the **Haver** package, the only changes made are:

- While the return type of data queries is still `zoo`, the **zoo** package is no longer automatically installed with the **Haver** package. If you do not have **zoo** installed and want to use the legacy dlx family of functions, you have to install it yourself (see `install.packages`).
- Some of the help files have been reworked slightly.
- Path settings will be preserved from one R session to the next.

Differences to the New Functions and Interactions between Old and New Functions

This section highlights the most important differences between the legacy `dlx` family of functions and the recommended `haver` family of functions. The following table gives a quick overview of old and new function names:

legacy `dlx` family	**corresponding `haver` family function**
`dlxSetDbPath`	`haver.path`
`dlxGetInfo`	`haver.metadata`
`dlxGetData`	`haver.data`
`dlxSetAggregation`	
`dlxGetAggregation`	

The functionality of `dlxSetAggregation` is now incorporated in `haver.data`.

Path Setting: `dlxSetDbPath` has been replaced by `haver.path`. Both functions store the database path in a global variable '.dlxdbpath'. This means that setting the path using `dlxSetDbPath` will be relevant for the recommended `haver` family of functions. Conversely, setting the path using `haver.path` will also provide the setting for the legacy `dlx` family of functions.

Regardless of which function you use, path settings will be preserved from one R session to the next.

Name Changes: The `varnames` argument of `dlxGetData` has been renamed into `codes` for `haver.data`. The casing of metadata fields returned by `dlxGetData` has been changed to be strictly lower case in `haver.metadata`. The metadata field 'varname' of `dlxGetData` has been renamed to 'code' in `haver.metadata`.

Metadata: `dlxGetInfo` only lets you query metadata for one code at a time. `haver.metadata` lets you specify lists of codes, or lets you retrieve metadata for an entire database. In addition, `haver.metadata` returns the information in a more convenient fashion, namely as a `data.frame` or as a `HaverMetaData` object. Both can be filtered, and `data.frames` from different metadata queries can be appended using `rbind`. The resulting objects can be passed to `haver.data` for data queries.

Data Queries:

Return Types: haver.data does not by default return a zoo object. Instead, it returns a HaverData object. If you want haver.data to give you zoo objects, you can either set the value of the argument rtype to zoo, or you can have it return a HaverData object first and then coerce it to a zoo object using haver.as.zoo.
HaverData objects can also be coerced into a matrix or a data.frame.

Metadata Associated with Data Queries: For each data query, haver.data automatically retrieves the metadata associated with the series retrieved, as long as you use the default return type HaverData. You can access the metadata of the HaverData object simply by using haver.datamd(hd) which, among other things, gives you a listing of the series descriptors.

Data Set Frequency: The frequency argument of dlxGetData only applied if no series in the query was recorded in a lower frequency. By contrast, the frequency argument of haver.data always pins down the frequency of the returned data set. If series cannot be aggregated or disaggregated to fit this frequency they get removed from the query.

Other: With haver.data, you can feed objects created by haver.metatdata into the data query.
The time display has also improved. With haver.data, you can choose between period display and end-of-period date display.

Aggregation Mode: The recommended haver family of functions has no direct counterparts to dlxSetAggregation and dlxGetAggregation. The aggregation mode is instead determined by the aggmode argument of haver.data. If this argument is not supplied, aggmode always defaults to ‘strict’. This is independent of what you set using dlxSetAggregation.

The behavior of dlxGetData is different: Here settings invoked by dlxSetAggregation really determine under which aggregation mode data are queried.

For a description of aggregation modes, see haver.data.

Using Aliases for Haver Series Codes: In dlxGetData, you can supply an argument .colnames that renames Haver series to names of your liking. haver.data does not have a corresponding argument. Instead, the renaming feature has been expanded for HaverMetaData and HaverData objects. You can incrementally apply Haver code aliases of your liking, and you can reset names to the original Haver codes. The mapping of original Haver series codes and your aliases is always stored in the objects and can be extracted at any time.

Starting and Ending Dates of Queries: In dlxGetdata, arguments ‘start’ and ‘end’ could be supplied as different objects (character strings, R Date objects, chron objects, etc.). In haver.data, these arguments must be R Date objects.

Author(s)

Haver Analytics <tech@haver.com>

Maintainer: Haver Analytics <tech@haver.com>

See Also

Haver_package
haver.path

haver.data
haver.metadata
HaverData
HaverMetaData
HaverErrorReport
haver.codelists
haver.datamd
haver.setalias
haver.limits
Haver_advanced

dlxGetData — *Get Time Series from a Haver Database (legacy function / help entry)*

Description

Get one or more time series, convert them to the same frequency, and store them in a zoo object. Frequency aggregation is currently supported, but not disaggregation.

Usage

```
dlxGetData(varnames, database, start, end, frequency, .colnames)
```

Arguments

`varnames`	A vector of Haver variable names. You can either give just the variable name, or a combination of the database name and variable name in the format "DATABASE:VARIABLE". If a database is not given, then the series will be assumed to reside in the database given in the function's database argument.
`database`	A Haver database name (with no filename extension). Each variable name that does not explictly contain a database name will be assumed to reside in this database.
`start`	Optional starting date for the time series window, defined using either a Date object or something that can be coerced to a Date (such as a string, POSIXct, or chron object). Leave blank to start at the latest date that still captures all the series. If the provided starting date is earlier than the first value of any of the series in your query, than the zoo object will be padded with blank values.
`end`	Optional ending date for the time series window. defined using either a Date object or something that can be coerced to a Date (such as a string, POSIXct, or chron object). Leave blank to end at the earliest date which still captures all the series. If the provided ending date is later than the last value of any of the series in your query, than the zoo object will be padded with blank values.
`frequency`	Optional frequency that the time series should be converted to, IF POSSIBLE. Options are "d", "wmon", "wtue", "wwed", "wthu", "wfri", "wsat", "wsun", "m", "q", "a". If left blank, all series are converted to the frequency of the lowest-frequency series listed in varnames. For instance, if you pass in the names of monthly and quarterly series then the resulting block will come back as quarterly. The frequency option will have no effect if you specify a frequency that is higher than the lowest frequency series in your list of variables. For example, if you request a monthly and quarterly series and ask for monthly frequency then you will still get data in a quarterly frequency. On the other hand, if you asked for annual data, the resulting data would in fact be annual.
`.colnames`	Optional vector of column names to be used for the zoo object. Length must be equal to varnames. If no values are given, the zoo object will use lowercase Haver variable names.

Value

A zoo object indexed by Date objects. The Date objects are set to the last day in each period.

Examples

```
## Not run:
## Load Haver library
library(Haver)

## Set the location of your Haver databases.
dlxSetDbPath("C:/DLX/data/")

## Get the series LANAGRA and GDP from database USECON.
dlxGetData(c("lanagra", "gdp"), "usecon")

## Get the series LANAGRA and GDP from database USECON and
## FFED from database WEEKLY
dlxGetData(c("lanagra", "gdp", "weekly:ffed"), "usecon")

## Get the series LANAGRA from database USECON in the date range
## 2000:1 to 2010:12
dlxGetData("lanagra", "usecon", start="2000-01-01", end="2010-12-31")

## Get the series LANAGRA from database USECON, convert to quarterly
## frequency, and get data in the range 2000:1 to 2010:4
dlxGetData("lanagra", "usecon", start="2000-01-01", end="2010-12-31",
  frequency="q")

## End(Not run)
```

`dlxGetInfo`	*Get Detailed Information about a Haver Time Series (legacy function / help entry)*

Description

Get detailed information about a Haver time series.

Usage

```
dlxGetInfo(varname, database)
```

Arguments

`varname`	A single Haver variable name.
`database`	The Haver database in which the series you are examining resides.

Value

Returns a list with the following fields defined.

database: The full path to the database.

varname: The time series variable name.

startDate: A Date object. For weekly, monthly, quarterly, and annual series, this will be the last day in the period.

endDate: A Date object. For weekly, monthly, quarterly, and annual series, this will be the last day in the period.

numobs: The number of observations in the series.

frequency: A frequency code. 10=Annual, 30=Quarterly, 40=Monthly, 51=Weekly Monday ... 57=Weekly Sunday, 60=Daily

dateTimeMod: A POSIXct object showing the time the series was last modified.

magnitude: Magnitude

decPrecision: The decimal precision.

difType: An integer value. 1 if data can contain negative values, 0 otherwise.

aggType: An integer value. 1=Average, 2=Sum, 3=End of period, 9=Undefined, 0=not set

dataType: One of US$, LocCur, US$/LC, LC/US$, INDEX, Units,

geography1: Primary geography code.

geography2: Secondary geography code.

descriptor: The series description.

shortSource: The abbreviated name of the time series publisher.

longSource: The full name of the time series publisher.

Examples

```
## Not run:
    dlxGetInfo("gdp", "usecon")

## End(Not run)
```

`dlxSetDbPath` *Set the Location of Your Haver Databases (legacy function / help entry*

Description

Set the location of your Haver databases by assigning to global variable .dlxdbpath

Usage

```
dlxSetDbPath(path)
```

Arguments

`path`	The path to your DLX/data directory, with or without a trailing slash.

Examples

```
    ## Not run:
    dlxSetDbPath("C:/DLX/data/")

## End(Not run)
```

dlxSetAggregation *Sets the Haver Aggregation Mode (legacy function / help entry)*

Description

Sets the Haver aggregation mode.

Usage

```
dlxSetAggregation(mode = c("strict", "relaxed", "force"))
```

Arguments

mode	One of "strict", "relaxed", or "force".

Details

For a description of aggregation modes, see section ‘Aggregation Modes’ in `haver.data`.

Value

Returns "strict", "relaxed", or "force"

`dlxGetAggregation` *Get the Haver Aggregation Mode (legacy function / help entry)*

Description

Get the Haver aggregation mode.

Usage

```
dlxGetAggregation()
```

Value

Returns "strict", "relaxed", or "force"