Haver Python Reference

Release 1.0.0

Haver Analytics

May 10, 2017

CONTENTS

CHAPTER

ONE

THE HAVER PACKAGE

Query data and metadata from Haver Analytics databases

Author: Haver Analytics

If you are new to the Haver package, we recommend consulting Section *Introduction to the Haver Package for Python* of the Haver Python Reference available in the client section of http://www.haver.com. If you want to jump straight to function documentation, see the help entries for *Haver.data* and *Haver.metadata*.

See also:

All functions and objects from the Haver package:

- *Haver.data()* : query time series data from Haver Analytics databases
- *Haver.metadata()* : query time series metadata from Haver Analytics databases
- *Haver DataFrame* : return type of `Haver.data()`
- *Haver Meta-DataFrame* : return type of `Haver.metadata()`
- *Haver ErrorReport dictionary* : return type for incorrectly specified queries
- *Haver.path()* : set and query the path to Haver databases
- *Haver.databases()* : print or return list of Haver Analytics databases
- *Haver.limits()* : set and get limits for Haver Analytics data queries
- *Haver.codelists()* : extract lists of Haver series codes
- *Haver.verbose()* : set verbose message mode of Haver functions to `True` or `False`

CHAPTER

TWO

INTRODUCTION TO THE HAVER PACKAGE FOR PYTHON

2.1 Description

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

2.2 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. Data and metadata are returned in pandas DataFrames. These DataFrames will be referred to as Haver DataFrames and Haver Meta-DataFrames. Data and metadata are linked closely: A time series retrieval results in a DataFrame object that has a Meta-DataFrame attached to it, which can be accessed at any time. Conversely, a Haver Meta-DataFrame 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 *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.

2.3 Preliminaries

2.3.1 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.

2.3.2 Requirements

The Haver package is only available on Windows operating systems.

Python dependencies are pandas and numpy.

2.3.3 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 unambiguously 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 Python 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 never used for Haver series in order to avoid ambiguity. It is only used to refer to Python workspace variables and to the columns of a DataFrame object.

Casing

Function names of Haver functions are always lower case. As a convention in all of the documentation the Haver package is imported by the statement

```
>>> import Haver
```

Since the Python language is case-sensitive, the function `data()` of the Haver package is called by

```
>>> Haver.data([...])
```

If you do not like the capital 'H' in 'Haver' in the above statement, you may import the package as

```
>>> import Haver as haver
```

You could also import the package under an entirely different alias, e.g.

```
>>> import Haver as hv
```

We recommend that you do not do so, however. The import aliases 'Haver' and 'haver' are consistent with function names of DLX interfaces to other analytical software, so using the aliases 'Haver' and 'haver' produces statements that are more portable.

Arguments to Haver functions are never case-sensitive. In particular, you can specify Haver series codes in upper case or lower case or any mixture thereof.

Almost all character-based return values of 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 Haver Meta-DataFrame are also lower case. The exception to the 'return character values in lower case' convention of Haver functions are the values of some fields of Haver Meta-DataFrames, as well as the database list returned by `Haver.databases()`.

Argument Abbreviations

Most values given as arguments to Haver functions can be abbreviated. For example, in order to specify the output frequency of a data set returned by `Haver.data()`, you can use `frequency='q'` instead of `frequency='quarterly'`.

Example Code Variable Names

The most important objects of the Haver package are *Haver DataFrame* and *Haver Meta-DataFrame*, to be described below and in other help files. Whenever an example Python code statement in a help file assigns such an object to a workspace variable, the names 'hd' and 'hmd' are used, possibly with suffixes that are easily understood within the context ('hd_1', 'hd_2', 'hmd_d', etc.). Dictionaries that help files refer to as *Haver ErrorReport dictionaries* are called 'her'. Any other Python variables that are defined in example code start with 'ex_'.

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 'NaN' ('not a number'). In data sets missing values are recorded as `numpy.nan`.

2.4 Concurrent Execution

Warning: The Haver package for Python is not thread-safe.

Do not use the Haver package in combination with e.g. the `threading` module in order to run multiple threads. If you try to do so, Python will respond with non-informative errors, or may even crash. If you want to run multiple queries in parallel, use multiple processes, e.g. via the `multiprocessing` module.

2.5 Preparatory Steps for Python Code Examples

2.5.1 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.

2.5.2 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')
```

These statements enclose 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, if executed, displays the path setting that was in place before the 'Examples' section was entered.

Depending on the configuration of Python and of DLX on your machine, it may occur that calling `Haver.path('examples')` and `Haver.path('restore')` does not work. However, you can still set the path to the example databases manually in order to execute example statements.

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

```
>>> Haver.path()
```

Then, execute

```
>>> Haver.path(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:\users\me\Python\lib', you would have to execute

```
>>> Haver.path('c:\users\me\Python\lib\Haver\dat')
```

Once you are done looking at the examples, manually set the path back to the actual Haver databases.

2.6 Where to Go from Here

First, read and/or execute the Examples section below. 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 package 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.

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.

2.7 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.

```
Haver.path()
Haver.path('examples')
```

Display current path setting:

```
Haver.path()
```

Set the database path to the directory of the example databases and check contents:

```
Haver.path('examples')
Haver.databases()
```

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 variable is appropriate. As usual, when assigning to a variable no output is printed to the screen:

```
hmd_mqa = Haver.metadata(database='havermqa')
```

Variable `hmd_mqa` now holds a Haver Meta-DataFrame. Its data type is a pandas DataFrame. You can display the contents of the DataFrame and of a subset thereof in the Python console. pandas DataFrames provide many ways of indexing into the data. See the pandas documentation for details.

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(['fxtwb', 'fxtwm', 'fxtwotp'],
       'haverd', startdate='2012-02-15')
```

hd_1 is a pandas DataFrame:

```
type(hd_1)
```

The metadata has automatically been saved within this object. It is accessible through the haverinfo member (dict) of the Haver DataFrame:

```
hd_1.haverinfo['metadata']
```

Alternatively, we can assign the Haver Meta-DataFrame to a variable and then customize the output:

```
hmd_1 = hd_1.haverinfo['metadata']
hmd_1[['code', 'descriptor']]
```

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 DataFrame is weekly.

```
hd_2 = Haver.data(['haverD:fxtwb', 'haverW:farbn'],
            startdate='2012-01-01')
hd_2
```

Next, we pull in data from the entire 'HAVERMQA' database. To do this, we use the Haver Meta-DataFrame stored in 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(hmd_mqa)
hd_mqa
```

You can specify a higher frequency if you want, say, quarterly. Then Haver.data() drops annual series from the query. We look at the last 10 observations of the first 5 codes in the DataFrame:

```
hd_mqa2 = Haver.data(hmd_mqa, frequency='q')
hd_mqa2.iloc[-10:, 0:4]
```

You can plot the data from the Haver DataFrame (requires Matplotlib):

```
import matplotlib.pyplot as plt
hd_mqa2.loc[:,['mm', 'ms']].plot()
plt.show()
```

```
>>> Haver.path('restore')
```

CHAPTER

THREE

FUNCTIONS AND OBJECTS

3.1 Haver.data

`Haver.data`(*codes*, *database=None*, *startdate=None*, *enddate=None*, *frequency=None*, *aggmode='strict'*, *eop=True*, *dates=False*, *uselimits=True*)

Description

`Haver.data()` queries time series data contained in Haver Analytics databases. The data are returned as a pandas DataFrame.

`Haver.hasfullcodes()` is a small helper function.

This help entry assumes that you are familiar with the Haver Python Reference Section *Introduction to the Haver Package for Python*.

Syntax

```
hd = Haver.data(codes)
hd = Haver.data(codes, database)
hd = Haver.data(codes, database, [...])
hd = Haver.data(codes, None    , [...])

Haver.hasfullcodes(x)
```

Parameters

- **codes** (*str or tuple of str or list of str*) – codes is typically a list of str, or it is an object that fulfills the conditions of `Haver.hasfullcodes()`. If a list of str 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 list of str 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 'Notes' below). A single series code can also be supplied as str.

- **database** (*str or list or tuple of str with one element*) – 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.

- **`startdate`** (*`str or datetime.date`*) – Specifies the starting date of the query. If is is specified as str, it is converted to a datetime.date object using the input format yyyy-MM-dd, e.g. ('2005-12-31). Even though `startdate` is supplied as a datetime.date object (or date string), it is interpreted within the context of the frequency of the DataFrame. This is an important point. See section *Specifying Query Starting and Ending Periods* in the Haver Python Reference for details.
- **`enddate`** (*`str or datetime.date`*) – Specifies the ending date of the query. Analogous comments as for `startdate` apply.
- **`frequency`** (*`str or list or tuple of str with one element`*) – Specifies the target frequency of the query. One of 'daily', 'weekly', 'monthly', 'quarterly', 'annual', or 'yearly', or any abbreviation thereof, down to a single character.
- **`aggmode`** (*`str or list or tuple of str with one element`*) – The aggregation mode of the query. One of 'strict', 'relaxed', or 'force', or any abbreviation therof. See section *Temporal Aggregation Modes* in the Haver Python Reference for details. Default: 'strict'.
- **`dates`** (*`bool`*) – Determines whether time periods are shown as dates (e.g. '2000-03-31' instead of '2000Q1'). Data with frequencies other than weekly are displayed with respect to time periods. Weekly data are shown as dates. The date shown for weekly data corresponds to the anchor weekday of the series or DataFrame. See *Daily and Weekly Queries* in the Haver Python Reference for more details. Default: `True` for weekly DataFrames, `False` for other frequencies.
- **`eop`** (*`bool`*) – Determines whether dates are recorded beginning-of-period or end-of-period. Only relevant if `dates=True`. Default: `True`.
- **`uselimits`** (*`bool`*) – If `False`, `Haver.data()` ignores settings regarding query limits. Default: `True`.

Returns

The return value of `Haver.data()` is a pandas DataFrame object. In the context of the Haver Package, this is referred to as a Haver DataFrame.

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

`Haver.hasfullcodes()` returns a `bool`.

Return type pandas.core.frame.DataFrame

Notes

(Syntax Detail)

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 referring to all series codes contained in the database specified. In `Haver.data()`, such an interpretation is not made, and consequently the argument `codes` is not optional here.

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, (str) 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 list (tuple) of str. `Haver.hasfullcodes()` returns a logical `True` if an object satisfies the following conditions:

- it is a pandas DataFrame
- whose first two columns are named 'database' and 'code'
- and have admissible data types, as described above under 'Input Arguments'
- with no empty entries
- and have only alphanumeric entries
- and have at least one entry (the DataFrame has at least one row).

See the `Haver.data()` 'Examples' section in the Haver Python Reference for more information on this technique.

The Haver Python Reference provides much more detail on `Haver.data()`, including examples.

3.1.1 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 DataFrame, 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. This behavior occurs because the Haver package supports temporal aggregation, but not temporal disaggregation. pandas DataFrames provide such functionality, however.

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 *Haver Meta-DataFrame*). They define which calculations are carried out when a series is aggregated. aggmodes determine how these calculations behave in the presence of NaNs. `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 NaNs 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 NaN also, under the default aggregation mode 'strict'. As a result, aggregating such daily series to monthly series will produce NaNs 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	agg-mode	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.

3.1.2 Specifying Query Starting and Ending Periods

Start and end dates have to be supplied either as Python `datetime.date` object or as str. The latter are converted to `datetime.date` objects using the date mask 'yyyy-MM-dd' (e.g. 2005-12-31'). 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 the argument `enddate` is after the last observation of the implied data set, or if the argument `startdate` is before the first observation of the implied data set, observations in the data set returned are padded with NaNs 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 calendar dates (in the form of Python `datetime.date` objects), but the interpretation of what these dates mean depends on the frequency of the data set returned. For example, for `startdate='2012-01-01'` 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 `startdate` as above and in addition `enddate='2012-05-31'`, 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 were in addition a daily series in the query, an aggregated value for 2012 would be calculated for it too, based on all data points from 2012-01-01 to 2012-12-31. To summarize:

- Start and ending periods are supplied as calendar dates (in the form of as Python `datetime.date` objects), 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 *Temporal 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 `startdate` and `enddate`.

3.1.3 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 Haver ErrorReport dictionary. 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 Haver DataFrame 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 NaNs 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 section *Metadata Fields* in *Haver Meta-DataFrame*). 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 Haver DataFrame. 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 Haver ErrorReport dictionary. This structure contains details on what went wrong with the query. The `Haver.codelists()` method can be applied to the Haver ErrorReport dictionary and returns lists of series codes that could not be found.

Duplicate Codes, Python Keywords

Duplicates of full series codes are removed automatically. A warning is issued. For example, the query

```
>>> hd = Haver.data(['fxtwb', 'fxtwb'], 'haverd')
```

will return a data set that contains 'haverd:fxtwb' just once.

Duplicate regular series codes are allowed. In this case, variable names in the output DataFrames will be prepended with the database names. Full series codes are also used as variable names if at least one of the regular series codes is identical to a Python keyword (e.g. `for`). For example,

```
>>> hd = Haver.data(['daily:ffed', 'usecon:gdp'])
```

will use 'ffed' and 'gdp' as DataFrame column names, whereas

```
>>> hd = Haver.data(['daily:ffed', 'weekly:ffed'])
```

and

```
>>> hd = Haver.data(['daily:ffed', 'usecon:for'])
```

will use 'daily_ffed', 'weekly_ffed' and 'daily_ffed', 'usecon_for', respectively. In all cases, the database origin of the series is tractable via the `haverinfo` member dict of a Haver DataFrame.

3.1.4 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.

Weekly is the only frequency for which data are associated with a date instead of a period (i.e. the resulting DataFrame's index is of type `DateTimeIndex` instead of `PeriodIndex`). The reason is to avoid confusion as to what the exact release day-of-week is. The default option combinations for weekly data `dates=True` and `eop=True` show the correct dates. If you wish to show periods for weekly data, explicitly supply `dates=False` to `Haver.data()`. For other frequencies, the default for option `dates` is `True`, i.e. periods are shown.

If you supply arguments `startdate` and `enddate` to daily or weekly queries, 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 *Temporal Aggregation Modes* above.

3.1.5 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.

3.1.6 Interrupting Execution

If you have specified a large query that is taking a long time to conclude, you can cancel it in the usual way. See *Query Execution Time*.

3.1.7 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 = '2012-01-10'
ex_tN = '2012-01-13'
Haver.data([       'fxtwb',        'fxtwm',        'fxtwotp'], 'haverd'  ,␣
↪startdate=ex_t1, enddate=ex_tN)
Haver.data([       'FXTWB',        'FXTWM',        'FXTWOTP'], 'HAVERD'  ,␣
↪startdate=ex_t1, enddate=ex_tN)
Haver.data(['haverd:fxtwb',        'fxtwm',        'fxtwotp'], 'haverd'  ,␣
↪startdate=ex_t1, enddate=ex_tN)
Haver.data(['haverd:fxtwb', 'haverd:fxtwm', 'haverd:fxtwotp'],     None  ,␣
↪startdate=ex_t1, enddate=ex_tN)
Haver.data(['haverd:fxtwb', 'haverd:fxtwm', 'haverd:fxtwotp'], 'havermqa',␣
↪startdate=ex_t1, enddate=ex_tN)
```

Querying codes from multiple databases is possible:

```
ex_tN = '2012-01-31'
hd_1  = Haver.data(['haverD:fxtwb', 'haverW:lic', 'fcm1'] , 'haverMQA', startdate=ex_
↪t1, enddate=ex_tN)
```

Querying all codes of a database is possible with `Haver.metadata()` but not with `Haver.data()`, so the following produces an error:

```
>>> Haver.data(database='haverd')     # <= produces an error!
```

You can assign the return value of `Haver.data()` to a variable, of course:

```
ex_tN = '2012-01-13'
hd_2  = Haver.data(['fxtwb', 'fxtwm', 'fxtwotp'] , 'haverd', startdate=ex_t1,␣
↪enddate=ex_tN)
hd_2
```

You can request a lower frequency than daily, e.g. quarterly (at this frequency, we do not have to use options 'startdate' and 'enddate' in order to limit output):

```
hd_3 = Haver.data(['fxtwb', 'fxtwm', 'fxtwotp'] , 'haverd' , frequency='q')
hd_3
```

The data points are NaNs 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 *Temporal Aggregation Modes* above.

```
hd_4 = Haver.data(['fxtwb', 'fxtwm', 'fxtwotp'] , 'haverd' , frequency='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')
hmd_mqa.iloc[0:10, 0:5]
```

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(['c', 'fcm1', 'ift'], 'havermqa', startdate='2008-01-13')
hd_5
```

When we explicitly request a quarterly frequency, monthly series get aggregated while annual ones get dropped:

```
hd_6 = Haver.data(['c', 'fcm1', 'ift'], 'havermqa', startdate='2008-01-13', frequency=
↪'q')
hd_6
```

We can investigate what series got dropped by using `Haver.codelists()`:

```
cl = Haver.codelists(hd_6)
cl.keys()
cl.values()
cl['nodisagg']
```

Every time series data retrieval also retrieves the series metadata and attaches it to the Haver DataFrame object. You can access it through the `haverinfo` member dict of Haver DataFrames:

```
hd_6.haverinfo['metadata']
```

The previous call returned a Haver Meta-DataFrame. When assigning it to a variable, you can customize its output:

```
hmd_6 = hd_6.haverinfo['metadata']
hmd_6[['code', 'frequency']]
```

You can retrieve an identical Haver DataFrame that displays dates instead of period values:

```
hd_7 = Haver.data(['c', 'fcm1', 'ift'], 'havermqa',
                  startdate='2008-01-13', frequency='q', eopdates=True)
```

We can use a Haver Meta-DataFrame to specify codes for `Haver.data()` queries:

```
hmd_w = Haver.metadata(database='haverw')
hd_w  = Haver.data(hmd_w)
```

We can add codes from other Haver Meta-DataFrames:

```
ex_tbl_d    = Haver.metadata(database='haverd')
ex_tbl_dw   = ex_tbl_d.iloc[:,0:2].append(hmd_w.iloc[:,0:2])
ex_tbl_dw.iloc[0:6,:]

import pandas as pd
ex_manual = pd.DataFrame({'database':['havermqa', 'havermqa'], 'code':['c', 'cd']})
ex_tbl_all  = ex_tbl_dw.append(ex_manual)
```

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_tbl_all)
```

Since this is `True`, we can pull in the data:

```
hd_all = Haver.data(ex_tbl_all)
hmd_all = hd_all.haverinfo['metadata']
hmd_all[['database', 'code', 'descriptor']]

>>> Haver.path('restore')
```

3.2 Haver.metadata

`Haver.`**`metadata`**(*codes=None*, *database=None*, *decode=True*)

Description

`Haver.metadata()` queries metadata information on time series contained in Haver Analytics databases. The information is returned in a pandas DataFrame.

This help entry assumes that you are familiar with the Haver Python Reference Section *Introduction to the Haver Package for Python*.

Syntax

```
Haver.metadata(codes)
Haver.metadata(codes, database)
Haver.metadata(codes, database, [...])
Haver.metadata(None , database)
Haver.metadata(None , database, [...])
```

Parameters

- **codes** (*str or tuple of str or list of str*) – codes is typically a list of str, or it is an object that fulfills the conditions of `Haver.hasfullcodes()`. The elements of the list (tuple) can be regular series codes or additionally supply the database where the code resides, in which case they must be specified in *database:seriescode* format. Valid examples are 'gdp' and 'usecon:gdp'.

  The only difference of usage of the codes argument in comparison to `Haver.metadata()` is that it is IS optional here (see section 'Details' below). A single series code can also be supplied as str.

- **database** (*str or list or tuple of str with one element*) – 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** (*bool*) – This argument is rarely used. By default `decode is True`, i.e. 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` value of `True` will re-translate this into 'AVG'.

Returns

- *pandas.core.frame.DataFrame* – The return value of `Haver.metadata()` is a pandas DataFrame object. In the context of the Haver package, this is refered to as a *Haver MetaDataFrame*.

- *Haver ErrorReport dictionary* – In the case of invalid series code and/or database specifications, a *Haver ErrorReport dictionary* is returned.

Notes

(Syntax Detail)

Argument `codes` may be omitted if the argument `database` is supplied, in which case the function interprets the request as referring to all series codes contained in the database specified. 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 the database specified.
3. supply both `codes` and `database` . Metadata for all elements of `codes` are 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, (str) values of arguments supplied are not case-sensitive.

The Haver Python Reference provides much more detail on `Haver.metadata()`, including examples.

3.2.1 Query Errors

As soon as one series code or one database specified in a query cannot be found, `Haver.metadata()` returns a Haver ErrorReport dictionary. This structure 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.

3.2.2 Interrupting Execution

If you have specified a large query that is taking a long time to conclude, you can cancel it in the usual way. See *Query Execution Time*.

3.2.3 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([       'fxtwb',        'fxtwm',        'fxtwotp'], 'haverd')
Haver.metadata([       'FXTWB',        'FXTWM',        'FXTWOTP'], 'HAVERD')
Haver.metadata(['haverd:fxtwb',        'fxtwm',        'fxtwotp'], 'haverd')
Haver.metadata(['haverd:fxtwb', 'haverd:fxtwm', 'haverd:fxtwotp']          )
Haver.metadata(['haverd:fxtwb', 'haverd:fxtwm', 'haverd:fxtwotp'], 'havermqa')
```

Querying codes from multiple databases is possible:

```
hmd_1 = Haver.metadata(['haverD:fxtwb', 'haverW:lic', 'fcm1'] , 'haverMQA')
```

Querying all codes of a database is not possible with `Haver.data()`, but it is with `Haver.metadata()`:

```
hmd_2 = Haver.metadata(database='haverd' )
```

Note that when querying an entire database, you must name the argument `database` explicitly.

```
Haver.metadata('haverd')     # <= produces an error!
```

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')
```

or

```
Haver.metadata(None, 'haverd')
```

When specifying `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)
hmd_w['frequency'].value_counts()
```

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 *Haver Meta-DataFrame*). This can be confirmed by looking at one of the fields 'startdate' or 'enddate':

```
from datetime import datetime
ex_daynums = [datetime.isoweekday(x) for x in hmd_w['enddate'].tolist()]
ex_daynums
```

The help for `datetime.isoweekday()` confirms that a 3 in the output list corresponds to a Wednesday, and 6 corresponds to a Saturday.

```
>>> Haver.path('restore')
```

3.3 Haver DataFrame

3.3.1 Description

A Haver DataFrame is the return type of `Haver.data()`. It contains times series data queried from Haver Analytics databases.

To learn about the many possibilities that pandas DataFrames provide for indexing into the data and for data analysis in general, consult the pandas documentation.

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

3.3.2 Details

Shape and Time Recording

If you query K variables, the resulting Haver DataFrame will have K columns. The time vector will be stored as the DataFrame's index. The time vector of a Haver DataFrame will never have gaps, which implies that missing values of time series are always explicitly shown as NaNs in the DataFrame.

`haverinfo` Member Dictionary

A Haver DataFrame has a special member dictionary called 'haverinfo' that stores additional information about the query. Available keys are:

Dict Key	Dict Contents
description	query date and time (str)
vardescription	series labels (list of str)
metadata	series metadata (Haver Meta-DataFrame)
frequency	DataFrame frequency (str)
codelists	various lists of codes related to the query (dict)

Note the `metadata` key: It contains all the metadata for all time series of a Haver DataFrame as a Haver Meta-DataFrame. You can access them trough *Haver_DataFrame_name*.haverinfo['metadata'].

Warning: The `haverinfo` member is not included in copy operations.

For example, the Haver DataFrame `hd2` in

```
hd2 = hd.ix[0:10].copy()
```

will not have a member called `haverinfo`.

The `codelists` field 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.

Coercion to Other Types

See the pandas documentation on how to coerce pandas DataFrames into other types.

3.3.3 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(hmd_mqa, frequency='q')
```

To determine which codes have been successfully queried and which ones got dropped, you can use the `Haver.codelists()` function:

```
ex_cl = Haver.codelists(hd_mqa1)
ex_cl.keys()
ex_cl.values()
ex_cl['noagg']
```

Both `hd_mqa1` and `hd_mqa2` contain a lot of data:

```
hd_mqa1.shape
```

You do have the option of printing a subset of the data set to the screen by indexing into the DataFrame, of course. For example:

```
hd_mqa1.iloc[-10:,[0:3 4:7]]
```

```
>>> Haver.path('restore')
```

3.4 Haver Meta-DataFrame

3.4.1 Description

A Haver Meta-DataFrame is the return type of `Haver.metadata()`. It contains metadata of time series stored in Haver Analytics databases.

To learn about the many possibilities that pandas DataFrames provide for indexing into the data and for data analysis in general, consult the pandas documentation.

A second use of Haver Meta-DataFrames is to use them as input for `Haver.data()`.

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

3.4.2 Details

Metadata Fields

A Haver Meta-DataFrame contains 18 pieces of information for each series code. These pieces of information are referred to as 'fields' in the following. The table below lists the available fields.

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	Field description
database	str	the database name (excluding the path to the database)
code	str	the Haver series code
startdate	datetime.date	last day of the series starting period
enddate	datetime.date	last day of the series ending period
frequency	str	the frequency code; one of D, W, M, Q, A (daily, weekly, monthly, quarterly, annual)
descriptor	str	the series description (label)
numobs	numpy.int32	number of periods between (and including) startdate and enddate
datetime-mod	pandas Timestamp	the time the series was last modified
magni-tude	numpy.int8	the magnitude of the series (e.g. 3 for series expressed in thousands)
decpreci-sion	numpy.int8	the decimal precision (number of digits after the decimal point)
diftype	numpy.int8	0/1 value: 1 if data can contain or contain non-positive values, 0 otherwise
aggtype	str	the aggregation type; one of AVG, SUM, EOP, UDF, NST (average, sum, end-of-period, undefined, not set)
datatype	str	one of US$, LocCur, US$/LC, LC/US$, INDEX, Units (where LocCur and LC stand for local currency)
group	str	the DLX update group
geo-graphy1	str	primary Haver geography code
geo-graphy2	str	secondary Haver geography code
short-source	str	the abbreviated name of the time series publisher
long-source	str	the full name of the time series publisher

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 the Haver package for Python, 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` to `False` will store numeric values in the returned Haver Meta-DataFrame. In rare cases, it may be useful to access the numeric values. The list of encodings is:

Field name	Encoding
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

`haverinfo` Member Dictionary

A Haver Meta-DataFrame has a special member dictionary called 'haverinfo' that stores additional information about the query. Available keys are:

Dict Key	Dict Contents
description	query date and time (str)
vardescription	DataFrame column labels (list of str)

Warning: The `haverinfo` member is not included in copy operations.

For example, the Haver DataFrame `hmd2` in

```
hmd2 = hmd.ix[0:10].copy()
```

will not have a member called `haverinfo`.

3.4.3 Examples

```
>>> Haver.path()
>>> Haver.path('examples')
```

We start out by pulling in all metadata for 'HAVERMQA'.

```
hmd_mqa = Haver.metadata(database='havermqa')
```

When printing the data

```
hmd_mqa[['code', 'frequency']]
```

we see that there are monthly series between rows 75 and 101 (not all rows may be displayed on your screen due to pandas' `display.max_rows` option). We choose to assign the rows in question to another Haver Meta-DataFrame using row-indexing:

```
hmd_2 = hmd_mqa.iloc[74:101,:].copy()
```

Let's do a frequency count:

```
hmd_mqa['frequency'].value_counts()
```

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 expressions (see e.g. `re`)).

```
hmd_3 = hmd_mqa[hmd_mqa['code'].str.contains('^fx')].copy()
```

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.

```
hmd_3['datatype'].value_counts()
```

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 look at another example. Imagine 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'] > datetime.strptime('1970-12-31', '%Y-%m-%d').
↪date()].copy()
hmd_4[['code','startdate','descriptor']]
```

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'] <= datetime.strptime('1970-12-31', '%Y-%m-%d').
↪date()].copy()
hd_5  = Haver.data(hmd_5)
hd_5.ix[:, 0:3]
```

If we want to investigate several full databases, the best way is to stack DataFrames. Below we create a cross tabulation of the frequency and database fields of a Haver Meta-DataFrame that contains data from two Haver databases.

```
hmd_d = Haver.metadata(database='haverD')
hmd_full = hmd_mqa.append(hmd_d).copy()

import pandas as pd
pd.crosstab(hmd_full.database, hmd_full.frequency)
```

```
>>> Haver.path('restore')
```

3.5 Haver ErrorReport dictionary

3.5.1 Description

A Haver ErrorReport dictionary is returned by `Haver.data()` and `Haver.metadata()` if the query is incorrectly specified, i.e. as soon as one series code or database cannot be found.

3.5.2 Details

The dictionary key-value pairs provide information on the database path used, the time of the query, and, most importantly, lists of codes that caused the query to fail. You can access these lists of series codes either directly or using the function *Haver.codelists*. The code lists are helpful for fixing up queries that have failed.

3.5.3 Examples

```
>>> Haver.path()
>>> Haver.path('examples')
```

The database 'HAVERD' is found but it does not contain series 'wrong1' and 'wrong2', so a Haver ErrorReport dictionary is returned.

```
her = Haver.data(['wrong1', 'wrong2', 'fxtwb'], 'haverd', startdate='2012-02-20')
```

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. The lists are stored as entries of a Python dict:

```
ex_cl = Haver.codelists(her)
ex_cl.keys()
```

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(ex_cl['codesfound'], start='2012-02-20')
```

```
>>> Haver.path('restore')
```

3.6 Haver.path

`Haver.`**`path`**(*vset=None*, *save=False*)

Description

Set, query, and save the path to Haver databases. A correct path setting is a prerequisite for data and metadata queries run by *Haver.data* and *Haver.metadata*.

Note that `Haver.path()` is in no way related to the Python module search path.

Syntax

```
P = Haver.path()
Haver.path(vset)
Haver.path(*keyword*)
Haver.path(save=True)
```

Parameters

- **vset** (*str*) – Specifies the directory where Haver databases reside.
- **save** (*bool*) – Save current path setting to file.

Returns When no argument is supplied, a str that holds the current setting of the database path. If no path is set, an empty str is returned.

Return type str

Notes

(Syntax Detail)

`Haver.path()` returns the current path setting.

`Haver.path(vset)` sets the database path to *vset*. *vset* must be supplied as an absolute file path. 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.

`Haver.path(*keyword*)` takes different actions. *keyword* can be one of 'auto', 'examples', 'restore', and 'saved', or any abbreviation thereof.

- 'auto' tries to automatically detect the correct database path if it is unknown to the user. In some instances this may not work, depending on the Python configuration and the DLX configuration of your machine.
- 'examples' sets the database path to the example DLX databases that come with the Haver package. Before doing so, `Haver.path()` will store the existing setting. You can re-enable this setting by issuing `Haver.path('restore')`.
- 'restore' restores the database path setting that was in place when `Haver.path('examples')` was invoked.
- 'saved' sets the path according to a previously saved state.

`Haver.path(save=True)` saves the current path setting to the Haver package settings file. This setting gets automatically restored upon import of the Haver package, or when reloading the Haver package. Note: Saving the Haver path affects all users that access this particular package installation.

(Package Import)

When the Haver package is imported (or reloaded via `importlib`), 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.

3.6.1 Path Setting Details

Preservation of the Path Setting between Python Sessions

Normally you never have to worry about setting the Haver path, as the Haver package will typically apply the correct path setting automatically. Should this mechanism fail, however, you have to set the database path manually. After doing this, you can use `Haver.path(save=True)` to save this setting. That way it will get automatically restored upon package import and you do not have to manually set that Haver path repeatedly, e.g. in new Python sessions.

> **Warning:** Be aware that, in an environment where multiple users share the same Haver package installation, saving a path setting affects all users. This is not true for any other path operations: They only affect a single user.

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.

3.6.2 Examples

Note: executing the following example statements will change the Haver database path setting.

Query the current setting (the returned path will most likely differ on your computer):

```
ex_origpath = Haver.path()
```

Let `Haver.path()` try to determine the correct setting automatically:

```
Haver.path('auto')
Haver.path()
```

Set path to examples databases:

```
Haver.path('examples')
Haver.path()
```

Restore the previous setting:

```
Haver.path('restore')
Haver.path()
```

Restore to setting that was in place before the examples section was run:

```
Haver.path(ex_origpath)
```

Manually set the database path to an explicitly given directory:

```
Haver.path('c:/dlx/data')
```

3.7 Haver.databases

`Haver.`**`databases`**(*disp=False*)

Description

Print or return list of Haver Analytics databases.

`Haver.databases(disp=True)` displays a list of Haver Analytics database files ('.DAT' extension files) that are accessible under the current setting of `Haver.path()`.

`dblist = Haver.databases()` returns the same list of databases in a list. All elements of the list have been stripped of the file extension and are in upper case.

Syntax

```
Haver.databases(disp=True)
dblist = Haver.databases()
```

Parameters **disp** (*bool*) –

Returns

- *None* – returned by `Haver.databases(disp=True)`.
- *list of str* – returned by `Haver.databases()`. List of Haver Analytics database files found. The entries are in upper case and the '.DAT' has been removed. The list is build with respect to the current setting of `Haver.path()`.

Examples

```
>>> Haver.path()
>>> Haver.path('examples')
```

Let's look at the example databases that are shipped with the Haver package:

```
Haver.path()
Haver.path('examples')
Haver.databases(disp=True)
```

We can save the database names in a list of str:

```
ex_dbs = Haver.databases()
ex_dbs
```

```
>>> Haver.path('restore')
```

3.8 Haver.limits

`Haver.`**`limits`**(*parameter=None*, *value=None*)

Description

Set and get limits for Haver Analytics data queries.

`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.

Syntax

```
S = Haver.limits()
Haver.limits('default')
Haver.limits(parameter, value)
```

Parameters

- **parameter** (*str*) – one of 'numcodes', 'numdatapoints', or 'default', or any unambiguous abbreviation thereof.

  `parameter='numcodes'` sets the maximum allowed number of codes in a query to `value`.

  `parameter='numdatapoints'` sets the maximum allowed number of data points in a query to `value`.

  `parameter='default'` ignores `value`, if supplied, and restores default values for 'numcodes' and 'numdatapoints', which are 1,000 and 15,000,000, respectively.

- **value** (*numeric non-negative scalar*) – specifies the limit for the 'numcodes' or 'numdatapoints' setting.

Returns The numbers in the first and second position indicate the limits on the number of codes and the number of data points, respectively.

Return type list

Notes

(Syntax Detail)

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 `uselimits='False'`.

`Haver.limits()`, with an empty argument list, returns the current settings.

`Haver.limits('default')` restores Haver default values.

Any values set by `Haver.limits()` will remain in effect until Python is restarted. A restart 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.

3.8.1 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 Python commands (CTRL+C). Still, after an interrupted query the memory claimed by Python is likely to be larger than before the query. If you have interrupted a large query, it may be advisable to save your work and restart the Python console.

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:

1. The memory consumed by data query operations is likely to be a multiple of the memory consumed by the Haver DataFrame returned.
2. Commands issued after the query may also consume a large amount of memory. For example, if you create a copy of the returned DataFrame, you are requesting the amount of memory consumed by the Haver data set for a second object.

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

3.8.2 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('numcodes', 10)
Haver.limits()
```

HAVERW has 21 codes. The limit we just set does not affect metadata retrievals:

```
hmd = Haver.metadata(database='haverw')
hmd.iloc[0:3,:]
```

Querying the data for 21 codes is not possible with the example settings for limits. The following generates an error:

```
>>> hd = Haver.data(hmd);    # <- generates error!
```

A query for 10 codes still works:

```
hd = Haver.data(hmd.iloc[0:10,:])
```

We can also explicitly state that `Haver.data()` should ignore limits:

```
hd = Haver.data(hmd, uselimits=False)
```

With default values, the full 21 code query goes through of course:

```
Haver.limits('default')
Haver.limits()
hd = Haver.data(hmd)
```

We restore the limits that were in place before the above examples were executed:

```
Haver.limits('numcodes'      , ex_storedlimits[0])
Haver.limits('numdatapoints' , ex_storedlimits[1])
Haver.limits()

>>> Haver.path('restore')
```

3.9 Haver.codelists

`Haver.`**`codelists`**`(`*objin*`)`

Description

Extract lists of Haver series codes.

`Haver.codelists()` extracts lists of Haver series codes from Haver Meta-DataFrames, Haver DataFrames, and Haver ErrorReport dictionaries.

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

Syntax

```
L = Haver.codelists(objin)
```

Parameters **objin** (*Haver ErrorReport dictionary or Haver DataFrame or Haver Meta-DataFrame*) –

Returns See the Haver Python Reference for details.

Return type dict of length 3 or 4

3.9.1 Details

If the input argument to `Haver.codelists()` is a Haver DataFrame, the return value is a `dict`. Associated with each key is a list of str, whose elements are Haver codes in *database:seriescode* format. The following table lists the keys and describes the associated code lists:

argument type	element name	vector contents
HaverData	noobs	series that do not have any valid data points
	noagg	series that cannot be aggregated
	nodisagg	series that cannot be disaggregated
	codesout	series whose data has been retrieved
HaverError dict	codesfound	series whose code and database were found
	databaseaccess	series whose database could not be accessed. The most likely reason for this is that the database name is misspelled or that the Haver database path is set incorrectly.
	codesnotfound	series whose series code was not found
	metadataaccess	series whose metadata could not be accessed

In addition to the above, if the input argument is a Haver Meta-DataFrame, `Haver.codeslists()` returns a list of str instead of a dict. Its elements hold the full series codes of series with records in the table.

3.9.2 Examples

```
>>> Haver.path()
>>> Haver.path('examples')
```

When applying the function to a Haver Meta-DataFrame, a list of codes that have records in the DataFrame is returned.

```
hmd_w = Haver.metadata(database='haverw')
Haver.codelists(hmd_w)
```

Applying the function to Haver DataFrames yields information on the successfully retrieved series and on the ones that got dropped:

```
hmd_mqa = Haver.metadata(database='havermqa')
hd_q    = Haver.data(hmd_mqa, frequency='quarterly')
Haver.codelists(hd_q)
```

Lastly, one can apply `Haver.codelists()` to Haver ErrorReport dictionaries. These are returned by `Haver.data()` and `Haver.metadata()` when there is a fundamental flaw in a query. The code lists are returned as a Python `dict`.

```
ex_codes = ['wrongdb:code1', 'wrongdb:code2', 'haverd:wrongcode']
her_hd   = Haver.data(ex_codes)

ex_cl = Haver.codelists(her_hd)
ex_cl.keys()
ex_cl.values()
ex_cl['databaseaccess']
```

The same object is returned from a correspondingly flawed metadata query:

```
her_hmd = Haver.metadata(ex_codes)
Haver.codelists(her_hmd)
```

```
>>> Haver.path('restore')
```

3.10 Haver.verbose

Haver.**verbose**(*vset=None*)

Description

Set verbose message mode of Haver functions to `True` or `False`.

`Haver.verbose()` sets the message display mode of Haver functions to be on or off. When you set the message display mode to non-verbose, the Haver functions will not generate any output if you terminate statements with a semi-colon, except for warnings and error messages.

Syntax

```
M = Haver.verbose()
Haver.verbose(vset)
```

Parameters **vset** (*bool or numeric 0/1*) – A numeric 0 (1) or boolean `False` (`True`) turn the verbose mode off (on).

Returns A logical False (True) indicates that the verbose mode is off (on).

Return type bool

Notes

(Syntax Detail)

`M = Haver.verbose()` returns a bool indicating whether the mode of displaying messages is on (True) or off (False).

`Haver.verbose(vset)` sets the display mode of messages on or off. `vset` must be a bool or a numeric 0/1 value.

The verbose setting concerns regular messages issued via the `print()` function only. Warnings and error messages are not affected. You can handle the display of warnings and errors as usual in Python.

Unless there is a specific reason to set the verbose mode to `False`, we recommend not doing so.

Examples

```
>>> Haver.path()
>>> Haver.path('examples')
```

We first save the current setting for the verbose mode:

```
ex_verb = Haver.verbose()
```

Verbose mode settings affect data queries, among other things:

```
hmd = Haver.metadata(database='havermqa')   # preparatory statement
Haver.verbose(1)
hd  = Haver.data(hmd)

Haver.verbose(0)
hd  = Haver.data(hmd)
```

After verbose mode was turned off, we were not notified that two series got dropped from the query.

```
Haver.verbose(ex_verb)
```

```
>>> Haver.path('restore')
```

CHAPTER

FOUR

INDICES AND TABLES

- genindex
- modindex
- search