DataMatrix Documentation

Release 0.9

Luca Beltrame and Giovanni Marco Dall'Olio

January 06, 2010

CONTENTS

1 Overview **3**

2 Requirements and installation **5**

3 Usage **7**
- 3.1 Invocation . . . 7
- 3.2 Basic operations . . . 7
- 3.3 Row and column manipulation . . . 8
- 3.4 Subsetting . . . 9
- 3.5 Further manipulation of DataMatrix objects . . . 9
- 3.6 Saving DataMatrix objects . . . 9

4 DataMatrix variants - EmptyMatrix **11**

5 Credits **13**

6 License **15**

7 Module reference **17**
- 7.1 DataMatrix . . . 17
- 7.2 EmptyMatrix . . . 18
- 7.3 Functions . . . 19

8 Indices and tables **21**

Module Index **23**

Index **25**

Contents:

CHAPTER

ONE

OVERVIEW

DataMatrix is a Python module that tries to emulate the behavior of the `data.frame` data structure of the R programming language. Data.frames are essentially tables that can be queried either by row or columns, or, in the presence of a header, even by column names.

DataMatrix emulates this behavior by reading a text file (or file-like object), and returning an object where rows and columns can be accessed by a variety of methods.

Row names are a way to identify a row precisely (R uses it because some of its operations return the names of the rows rather than their values) and are either read from a specific column or a progressive numeric value otherwise.

DataMatrix objects can be queried with a dictionary-like syntax (e.g., `matrix["column name"]`), or by other methods.

CHAPTER

TWO

REQUIREMENTS AND INSTALLATION

DataMatrix requires Python, at least version 2.5. Python 2.6 is also known to work. Python 3.0 compatibility has not been tested. It makes no use ofi third-party modules so it should work on a standard Python install. Being a pure Python module, it will work reliably on Windows, Linux, BSD and OS X.

The latest version can be downloaded from

http://www.dennogumi.org/projects/datamatrix

both as a source distribution and a Windows installer. Windows users just need to run the installer and follow the on-screen instructions. Other operating system users should download the source distribution, unpack it (`tar xvzf filename`) and then install it by issuing:

```
python setup.py install
```

as root.

CHAPTER

THREE

USAGE

3.1 Invocation

`DataMatrix` requires a file, or file-like object. A typical invocation is:

```
import datamatrix
matrix = datamatrix.DataMatrix(open("somefile"), header=True)
```

Aside the file object, which is mandatory, there are a number of parameters that can be used. First of all, the `header` parameters tells `DataMatrix` if the file to read has a header or not, and if so, the header will be used to assign names to the columns. Otherwise, it will just be a number for each column. To specify the column where row names are located, the `row_names` parameter is used:

```
matrix = datamatrix.DataMatrix(open("somefile", header=True, row_names=1))
```

In this case, row names are obtained from the first column in the file.

If you are loading a file with an empty first element on the header (that is the case with files saved by R) you must set the `fixR` parameter to `True`, which will work around this issue, otherwise you will obtain unpredictable results. `DataMatrix` uses the `csv` module to do its parsing, so you can specify additional parameters to define the format of your data, such as delimiter (the separator between fields), lineterminator and quoting (how to deal with non-numeric fields). See the csv module documentation for additional details.

Notice that since the `csv` module does not support Unicode input, using Unicode text with `DataMatrix` may give unpredictable results.

Lastly, you can tell the initializer to skip a certain numbers of lines using the `skip` parameter. New in version 0.9.

See Also:

Module `csv` Documentation of the `csv` standard module.

3.2 Basic operations

If you print a `DataMatrix` instance, you'll get some basic information:

```
>>> print matrix
    File name:
    Column with identifier names: None (numeric)
    No. of rows: 2
```

```
    No. of columns: 2
    Columns: Name, surname
```

With the columns attribute you can view the columns as a list:

```
>>> print matrix.columns
    ['Name', 'surname']
```

Row names can be printed intead with the `rownames` attribute.

You can access specific rows with the `getRow` method:

```
>>> matrix.getRow(1)
    ['1', 'Albert', 'Einstein']
```

Or specific columns with a dictionary-like syntax::

```
>>> matrix["surname"]
    ['Einstein', 'Marx']
```

Changed in version 0.8. In `DataMatrix` versions prior to 0.8, the `getColumn` method was used. This is no longer the case: the method has been marked as deprecated and will be removed in future versions.

To get a representation of your data, there is the `view` method:

```
>>> matrix.view()
    1 Albert Einstein
    2 Groucho Marx
```

3.3 Row and column manipulation

Rows and columns can be appended with the `append` and `appendRow` methods, respectively. In both cases, the item to be appended needs to be a sequence (list or tuple) and must be as long as the other columns (when appending columns) or cover all the columns (when appending rows):

```
>>> profession = ["scientist", "comedian"] # new column
>>> matrix.append(profession, "Job")

>>> entry = ["Isaac", "Asimov", "writer"] # new row
>>> matrix.appendRow(entry,"3")
```

Notice that when you append a row and a column you must specify a column or a row name to the methods, as the examples above show. Also, the rows and columns you are apppending need to be of the same length of the rows (or columns) already present in the `DataMatrix` instance.

Alternatively, you can insert rows and columns at a specified position using the `insert` (for columns) and `insertRow` (for rows). They behave exactly like the `append*` methods, with the difference that you must supply an integer argument (1 or greater than 1) representing the column or row number:

```
>>> matrix.insert(profession,"Job",2)
>>> matrix.inserRow(entry,"3",1)
```

New in version 0.7. If the number is greater than the number of columns or rows available, the method automatically defaults to the append variant. Again, rows and columns must be of the same length as the ones already present in the instance. New in version 0.9: You can bind multiple DataMatrix instances by rows and columns, using the `cbind` and `rbind` functions, which join matrices by columns and rows. An example:

```
>>> new_matrix = datamatrix.cbind(matrix1, matrix2)
>>> new_matrix = datamatrix.rbind(matrix1, matrix2)
```

Attempting to bind matrices of unequal lengths (rows or columns depending on the used function) will raise a `ValueError` exception.

3.4 Subsetting

New in version 0.9. You can generate subsets of your matrices using the `subset` function and using a list of columns as a parameter. The result is a new DataMatrix instance:

```
new_matrix = datamatrix.subset(old_matrix, ["Supplier", "Price"])
```

3.5 Further manipulation of DataMatrix objects

New in version 0.8. For some special uses, a number of functions have been provided. `elementApply` applies a function to the whole matrix, `matrixApply` applies a function to either rows or columns, giving a single result, while `filterMatrix` can be used to filter rows depending on the content of a specific column. For further information, refer to the documentation strings of those functions.

You can also transpose the matrix (invert the rows and the columns) with the help of the `transpose` function.

Also, two conveinence functions have been provided to quickly calculate the mean of columns or rows: they are `meanRows` and `meanColumns`, respectively. New in version 0.9: The `apply` function can be used to apply a function to a specific column, either to each element, or to the column as a whole.

3.6 Saving DataMatrix objects

You can write `DataMatrix` objects to files or file-like objects with the `writeMatrix` function present in the module:

```
fh = open("somefile.txt","w")
datamatrix.writeMatrix(matrix,fh)
```

Output formatting is again set via options to the csv module. Optionally you can save only part of the columns, specified as a list:

```
datamatrix.writeMatrix(matrix, fh, columns = ["Name","Job"])
```

If you want the header (column names) to be included, you need to set the `header` parameter to `True`:

```
datamatrix.writeMatrix(matrix, fh, header = True)
```

CHAPTER

FOUR

DATAMATRIX VARIANTS - EMPTYMATRIX

`EmptyMatrix` is a `DataMatrix` variant that does not use a file. Like the name says, it's an empty instance, which only has column names and row names. A typical instantiation of an `EmptyMatrix` instance is like this:

```
>>> matrix = EmptyMatrix(row_names=["1","2","3"],column_names=["Name","Surname"])
```

in this case, we create an `EmptyMatrix` instance consisting of three rows and two columns. Columns contain only empty lists when the instance is created: we can add column data by using assignments (e.g. `matrix["Name"] = XXX`) or by inserting or appending rows using the `append*` and `insert*` methods (which are inherited from `DataMatrix`).

Aside those details, `EmptyMatrix` instances behave exactly like their `DataMatrix` equivalents.

CHAPTER

FIVE

CREDITS

DataMatrix was started by me (Luca Beltrame) and Giovanni joined up later, with code fixes and most importantly with unit tests. Bug reports, feature requests and comments should be either sent via email (einar at heavensinferno dot net) or by leaving a comment at http://www.dennogumi.org/projects-2/datamatrix .

CHAPTER

SIX

LICENSE

This program is distributed under the terms of the GNU General Public License (GPL), version 2, or (at your option), any later version. This means that you can freely modify, copy and distribute the program, under the terms of said license. The COPYING file gives a good overview of what you can and can't do.

Although we hope that this program will be useful, it is without ANY WARRANTY, without even the implied warranty of MERCHANTABILITY or FITNESS FOR A PARTICULAR PURPOSE.

CHAPTER

SEVEN

MODULE REFERENCE

7.1 DataMatrix

class **DataMatrix** (*fh=None, row_names=None, header=True, delimiter='t', quoting=0, quotechar="'", fixR=False, skip=0*)

Pythonic implementation of R's data.frame structure. This class loads a file or a file-like structure then it transforms it into a dictionary of (row name, value) tuples for each column. Optionally only column values can be retrieved and at the same time single lines can be queried. Missing values are inputted as NAs.

The column used for row names (row_names) can be specified, otherwise rows are numbered sequentially. If the file has an header (default True), it is used to name the fields, otherwise the fields are numbered sequentially, with the first field being "x" (like R does).

If you are loading a text object saved by R using row.names=TRUE, the topmost, leftmost record will be blank. To parse such files, specify fixR as True in the initializer options.

Other options include the delimiter, line terminator and quoting, and they are passed directly to the csv DictReader instance which will read the file. See the csv module documentation for more details.

Notable methods are:

- getRow - returns a specific row
- **view - Outputs a tab-delimited view of a number of** lines. Start line and how much to show are configurable.
- **append - adds a column (of the same length and with the same** identifiers) to the matrix, kind of equivalent to R's cbind.
- **appendRow - adds a row at the end of the matrix, of the same length as** the columns. It can be seen as similar to R's rbind.
- insert - inserts a column at a specified index
- insertRow - similar to insert, but works with rows
- iterRows - cycle through rows
- getRowByID - get a row with a specified row name

append (*other, column_name*)

Method to append a column. It needs a sequence (tuple or list) and a column name to be supplied. The sequence must be of the same length as the other columns.

appendRow (*other, row_name*)

Appends a row to the end of the matrix. The row must encompass all the columns (i.e., it should be as long as to cover all the columns). The row name is specified in the mandatory parameter row_name.

getColumn (*key, column_name=False*)
Gets a specific column, without the identifier. The result is returned as a list. Optionally the column name can be printed. DEPRECATED: Use datamatrix[colname] instead.

getRow (*row_number, columns='all', row_name=True*)
Returns a specific row, identified from the row number, as a list. You can specify how many columns are outputted (default: all) with the columns parameter.

getRowByID (*rowId, **kwargs*)
Fetches a specific row basing on the identifier. If there is no match, a ValueError is raised.

insert (*other, column_name, column_no*)
Method to insert a column at a specified column index.

insertRow (*other, row_name, lineno*)
Method that inserts a row at a specified line number.

iterRows (***kwargs*)
Iterate over a matrix's rows.

```
>>> from StringIO import StringIO
>>> matrixfile = StringIO(
... '''a b c d
... 3 3 3 3
... 2 2 2 2''')


>>> matrix =  DataMatrix(matrixfile)
>>> for row in matrix.iterRows():
...     print row
['1', '3 3 3 3']
['2', '2 2 2 2']
```

pop (*index=-1*)
Method analogous to the pop method of lists, with the difference that this one removes rows and returns the removed item. If no index (a.k.a. row number) is supplied, the last item is removed.

rename (*oldColumn, newColumn*)
Renames a column.

replace (*other, colName*)
Replace a column with another.

view (*lines=10, start_at=1, *args, **kwargs*)
Method used to print on-screen the table. The number of lines, and the starting line can be configured via the start_at and lines parameters. Optional parameters can be sent to getRow to select which columns are printed.

7.2 EmptyMatrix

class EmptyMatrix (*identifier=None, row_names=None, columns=None*)
DataMatrix variant that once instantiated generates an empty matrix with specified columns and row names. Does not depend upon reading a file. Rows and columns, after initialization, can be added with insertRows, insert, appendRows and append methods, respectively.

7.3 Functions

writeMatrix(*data_matrix, fh=None, delimiter='t', lineterminator='n', quoting=0, header=False, row_names=True, *args, **kwargs*)
Function that saves DataMatrix objects to files or file-like objects. A file handle is a mandatory parameter, along with the data matrix object you want to use. You can optionally pass more parameters to getRows to select which columns are saved.

elementApply(*matrix, func, columns=None*)
Applies a function to each column for each row, and outputs a new matrix as result. If the function requires any type conversion, that must be done by the user. If the columns parameter is not None, the function is applied only to specific columns.

matrixApply(*matrix, func, what='rows', resultName='Function result'*)
Apply a user-specified function to all rows or all columns. If the function requires any type conversion, that must be done by the user. The function must process the row (or the column) and return a single value. The final result is a DataMatrix instance containing one row (or one column) with the function results. The name of the column (or row) can be changed with the resultName parameter.

filterMatrix(*matrix, func, column*)
Function which returns a DataMatrix with a column that satisfies specified criteria. In particular, "func" must be a function applied to each row (on the column of interest) and should return True if the row needs to be included, and False otherwise.

meanRows(*sourceMatrix*)
Convenience function to calculate the mean of the rows of a DataMatrix instance.

meanColumns(*sourceMatrix*)
Convenience function to calculate the mean of the columns of a DataMatrix instance.

transpose(*matrix, identifier='x'*)
Transposes a DataMatrix object: rows become columns and vice versa. The optional parameter `identifier` is passed as the resulting matrix's identifer name.

apply(*matrix, func, column, whole=False*)
Applies a function to a specific column. Rows are not supported (yet). The operation is performed in-place. If the parameter "whole" is specified, the function is applied to all the column at once, rather than to each of its members.

cbind(**args*)
Joins a list of matrices by their columns. They must have the same length. Additionally, the two matrices must not have equal column names. Returns a DataMatrix instance of the joined result.

rbind(*first_matrix, second_matrix*)
Joins two matrices by their rows. The number of columns of the two matrices must be identical. Returns a new DataMatrix instance with the joined result.

subset(*matrix, column_list*)
Creates a new matrix with only a subset of the columns present, as specified by the columns_list parameter.

CHAPTER

EIGHT

INDICES AND TABLES

- *Index*
- *Module Index*
- *Search Page*

MODULE INDEX

D

datamatrix, 17

INDEX

A

append() (datamatrix.DataMatrix method), 17
appendRow() (datamatrix.DataMatrix method), 17
apply() (in module datamatrix), 19

C

cbind() (in module datamatrix), 19

D

DataMatrix (class in datamatrix), 17
datamatrix (module), 17

E

elementApply() (in module datamatrix), 19
EmptyMatrix (class in datamatrix), 18

F

filterMatrix() (in module datamatrix), 19

G

getColumn() (datamatrix.DataMatrix method), 17
getRow() (datamatrix.DataMatrix method), 18
getRowByID() (datamatrix.DataMatrix method), 18

I

insert() (datamatrix.DataMatrix method), 18
insertRow() (datamatrix.DataMatrix method), 18
iterRows() (datamatrix.DataMatrix method), 18

M

matrixApply() (in module datamatrix), 19
meanColumns() (in module datamatrix), 19
meanRows() (in module datamatrix), 19

P

pop() (datamatrix.DataMatrix method), 18

R

rbind() (in module datamatrix), 19
rename() (datamatrix.DataMatrix method), 18
replace() (datamatrix.DataMatrix method), 18

S

subset() (in module datamatrix), 19

T

transpose() (in module datamatrix), 19

V

view() (datamatrix.DataMatrix method), 18

W

writeMatrix() (in module datamatrix), 19